# Table Of Contents

## API Documentation

- **2.1 Tutorial** ................................................................. 5
- **2.2 How Tutorials** ......................................................... 440
- **2.3 Infrastructure** ......................................................... 536
- **2.4 Build Manual** ............................................................. 542
- **2.5 SeqAn Style Guides** .................................................... 560
- **2.6 Glossary** ................................................................. 597
- **2.7 References** ............................................................... 598

---

**Bibliography** ................................................................. 601
This is the manual of the SeqAn library. SeqAn is the C++ template library for the analysis of biological sequences. It contains algorithms and data structures for

- string representation and their manipulation,
- online and indexed string search,
- efficient I/O of bioinformatics file formats,
- sequence alignment, and
- much more.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tutorial</strong></td>
<td>Each tutorial takes 30 to 60 minutes of your time for learning how to use SeqAn. Jump right into using SeqAn using our tutorials!</td>
</tr>
<tr>
<td></td>
<td>• Detailed installation descriptions are available in the <em>Getting Started</em> Tutorial.</td>
</tr>
<tr>
<td><strong>How Tos</strong></td>
<td>Use these short and target-oriented articles to solve common, specific problems.</td>
</tr>
<tr>
<td><strong>Build Manual</strong></td>
<td>These articles describe how to use the SeqAn build system and integrate into your Makefiles, for example.</td>
</tr>
<tr>
<td><strong>Infrastructure</strong></td>
<td>These pages describe the SeqAn infrastructure and are mostly interesting to SeqAn developers.</td>
</tr>
<tr>
<td><strong>SeqAn Style Guides</strong></td>
<td>Please follow these style guides for SeqAn library and application code.</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td>These pages contain definitions of various terms.</td>
</tr>
</tbody>
</table>
The API documentation can be found on here.

## 2.1 Tutorial

### 2.1.1 Getting Started

This chapter gives you the necessary steps to get started with SeqAn:

- Necessary Prerequisites
- Installing SeqAn from Subversion
- Creating a first build.
- Creating your own first application.

Use the following links to select your target operating system and IDE/build system. The bold items show the recommended build system for the given platforms.

**Linux using**
- *Makefiles*
- *Eclipse*

**Mac Os X**
- *Makefiles*
- *Xcode*

**Windows**
- *Visual Studio*

Click “more” for details on the supported development platforms.

---

**Note:**

**In-Depth Information:** Supported OS, Build Systems, and Compilers

The content of this box is meant as additional information. You do not need to understand it to use SeqAn or follow the tutorials.

There are three degrees of freedom when selecting a SeqAn development platform. The degrees of freedom are:
1. The **operating system**. We support Linux, Mac Os X and Windows.

2. The **build system**. This is partially orthogonal to the operating system, although each build system is only available on some platforms (e.g. Visual Studio is only supported on Windows). We use CMake to generate the actual build files and the build system maps to “CMake generators”. A CMake generator creates either build files for a build system (e.g. GNU Make) or a project file for an IDE (e.g. for Visual Studio 2008).

3. The **compiler**. This is partially orthogonal to the operating system and build system, although only some combinations of each are possible. For example, Visual Studio projects of a particular version can only use the Visual Studio compiler of the same version.

The SeqAn team offers support for the following operating systems, build systems, and compilers:

- **Operating System**: Linux, Mac Os X, Windows.
- **Build System**: Makefiles, Visual Studio projects, XCode projects, Eclipse CDT projects.
- **Compilers**: GNU g++ from version 4.1, LLVM/Clang from version 3.0, Visual C++ from Version 8.

We are told that SeqAn also works on FreeBSD. It should work with all generators available in CMake that work with the supported compilers (e.g. the CodeBlocks generator will probably work as long as you use it on a operating system with a supported compiler, although we cannot offer any support for CodeBlocks).

### Relevant How-Tos

Although slightly more advanced than “getting started”, the following How-Tos apply to setting up your build environment:

- **Using Parallel Build Directories**
- **Installing Contribs On Windows**
- **Integration with your own Build System**

### ToC

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting Started With SeqAn On Linux Using Makefiles</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Install</td>
</tr>
<tr>
<td>A First Build</td>
</tr>
<tr>
<td>Hello World!</td>
</tr>
<tr>
<td>Further Steps</td>
</tr>
</tbody>
</table>

### Getting Started With SeqAn On Linux Using Makefiles

This tutorial explains how to get started with SeqAn on Linux using Makefiles.

We assume that you are using the Debian or a Debian-like Linux distributions such as Ubuntu. The only difference to other distributions is the name of the packages and the package management system in the Prerequisites section. It should be very simple for you to tailor these instructions to your requirements.
Prerequisites Use the following command line to install the required dependencies: the Subversion client, the GNU C++ compiler, CMake for the build system and the Python script interpreter for running helper scripts.

```bash
~ # sudo apt-get install subversion g++ cmake python
```

The following command line installs optional dependencies: developer versions of zlib and libbz2 (for compressed I/O support) and the Boost library (required by a few apps).

```bash
~ # sudo apt-get install zlib1g-dev libbz2-dev libboost-dev
```

Install Now, go to the directory you want to keep your SeqAn install in (e.g. Development in your home folder).

```bash
~ # cd $HOME/Development
```

Then, use Subversion to retrieve the current SeqAn trunk:

```bash
Development # svn co https://github.com/seqan/seqan/branches/master seqan-trunk
```

You can now find the whole tree with the SeqAn library and applications in $HOME/Development/seqan-trunk.

A First Build Next, we will use CMake to create Makefiles for building the applications, demo programs (short: demos), and tests. For this, we create a separate folder seqan-trunk-build on the same level as the folder seqan-trunk.

```bash
Development # mkdir seqan-trunk-build
```

When using Makefiles, we have to create separate Makefiles for debug builds (including debug symbols with no optimization) and release builds (debug symbols are stripped, optimization is high). Thus, we create a subdirectory for each build type.

We start with debug builds since this is best for learning: debug symbols are enabled and assertions are active.

```
Warning: Compiling debug mode yields very slow binaries since optimizations are disabled. Compile your programs in release mode if you want to run them on large data sets.
The reason for disabling optimizations in debug mode is that the compiler performs less inlining and does not optimize variables away. This way, debugging your programs in a debugger becomes much easier.
```

```bash
Development # mkdir seqan-trunk-build/debug
Development # cd seqan-trunk-build/debug
```

The resulting directory structure will look as follows.

```
~/Development
 - seqan-trunk source directory
 - seqan-trunk-build
   - debug build directory with debug symbols
```

Within the build directory debug, we call CMake to generate Makefiles in Debug mode.

```bash
debug # cmake ../../seqan-trunk -DCMAKE_BUILD_TYPE=Debug
```

We can then build one application, for example RazerS 2:

```bash
debug # make razers2
```

Optionally, we could also use “make” instead of “make razers2”. However, this can take a long time and is not really necessary.
Hello World!  Now, let us create a sandbox for you. This sandbox will be your local workspace and you might want to have it versionized on your own Subversion repository at a later point. All of your development will happen in your sandbox.

We go back to the source directory and then use the SeqAn code generator to create a new sandbox.

```
debug # cd ..../seqan-trunk
seqan-trunk # ./util/bin/skel.py repository sandbox/my_sandbox
```

Now that you have your own working space, we create a new application `first_app`.

```
seqan-trunk # ./util/bin/skel.py app first_app sandbox/my_sandbox
```

Details about the code generator are explained in Using The Code Generator.

Now, we go back into the build directory and call CMake again:

```
seqan-trunk # cd ../seqan-trunk-build/debug
debug # cmake .
```

Tip:  When and where do you have to call CMake?

CMake is a cross-platform tool for creating and updating build files (IDE projects or Makefiles). When you first create the build files, you can configure things such as the build mode or the type of the project files.

Whenever you add a new application, a demo or a test or whenever you make changes to CMakeLists.txt you need to call CMake again. Since CMake remembers the settings you chose the first time you called CMake in a file named CMakeCache.txt, all you have to do is to switch to your debug or release build directory and call “cmake .” in there.

```
~ # cd $HOME/Development/seqan-trunk-build/debug
debug # cmake .
```

Do not try to call “cmake .” from within the seqan-trunk directory but only from your build directory.

The step above creates the starting point for a real-world application, including an argument parser and several other things that are a bit too complicated to fit into the Getting Started tutorial. Therefore, we will replace the program of the app first_app with a very simple example program.

Open the file `sandbox/my_sandbox/apps/first_app/first_app.cpp` (in your seqan-trunk directory) with a text editor and replace its contents with the following:

```
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h>    // to stream a CharString into cout

int main(int, char const **) {
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```

Afterwards, you can simply compile and run your application:

```
debug # make first_app
debug # ./bin/first_app
```

On completion, you should see the following output:
Hello World!
Hello SeqAn!

Congratulations, you have successfully created your first application within the SeqAn build system with Makefiles!

Further Steps  As a next step, we suggest the following:

• Continue with the Tutorials
• Look around in the files in sandbox/my_sandbox/apps/first_app or the demos in core/demos and extras/demos.
• For the tutorial, using the SeqAn build system is great! If you later want to use SeqAn as a library, have a look at Integration with your own Build System.

ToC

Contents

• Getting Started With SeqAn On Linux Using Eclipse CDT
  – Prerequisites
  – Install
  – A First Build
  – Hello World!
  – Further Steps

Getting Started With SeqAn On Linux Using Eclipse CDT

This article describes how to get started with SeqAn on Linux using Eclipse CDT.

We assume that you are using the Debian or a Debian-like Linux distributions such as Ubuntu. The only difference to other distributions is the name of the packages and the package management system in the Prerequisites section. It should be very simple for you to tailor these instructions to your requirements.

Prerequisites  Use the following command line to install the required dependencies: the Subversion client, the GNU C++ compiler, CMake for the build system, the Python script interpreter for running helper scripts, and Eclipse with the CDT plugin for C++ development.

~ # sudo apt-get install subversion g++ cmake python eclipse-cdt

The following command line installs optional dependencies: developer versions of zlib and libbzip2 (for compressed I/O support) and the Boost library (required by a few apps).

~ # sudo apt-get install zlib1g-dev libbz2-dev libboost-dev

Install  Now, go to the directory you want to keep your SeqAn install in (e.g. Development in your home folder).

~ # cd $HOME/Development

Then, use Subversion to retrieve the current SeqAn trunk:
You can now find the whole tree with the SeqAn library and applications in seqan-trunk.

A First Build  We will now use CMake to create Eclipse CDT project files. For this, we create a separate folder seqan-trunk-build on the same level as the folder seqan-trunk.

When using Eclipse CDT, we have to create separate project files for debug builds (including debug symbols with no optimization) and release builds (debug symbols are stripped, optimization is high). Thus, we create a subdirectory for each build type. We start with debug builds since this is best for learning: debug symbols are enabled and assertions are active.

Warning: Compiling debug mode yields very slow binaries since optimizations are disabled. Compile your programs in release mode if you want to run them on large data sets. The reason for disabling optimizations in debug mode is that the compiler performs less inlining and does not optimize variables away. This way, debugging your programs in a debugger becomes much easier.

The resulting directory structure will look as follows.

~/Development
  |-- seqan-trunk  source directory
  `-- seqan-trunk-build
      |-- debug-eclipse  build directory with debug symbols

Within the build directory debug-eclipse, we call CMake to generate The Eclipse CDT project in the debug mode.

default-eclipse # cmake ../../seqan-trunk -G "Eclipse CDT4 - Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug

Now, open Eclipse. If this is your first Eclipse start then you have to select a workspace. This is unrelated to the SeqAn project files generated earlier and you can pick any directory, e.g. use the default.

We will now import the generated CDT project: Select the Project Explorer in Eclipse (C++ perspective) and right click somewhere into the view. Select Import... and choose Existing Project into Workspace in the current dialog. Select Next and browse to your build directory in your workspace. Eclipse recognizes that there is a project file and selects it automatically. Confirm the selection with Finish.
Before you build any target you should change the launch preferences of Eclipse. In the menu select **Window > Preferences** Go to the menu **Run/Debug > Launching** and uncheck **Build (if required) before launching**. This prevents the project for building the target all (default preference), which can take quite a while depending on the system you are using. This can happen if, e.g. you accidentally run a binary before building the target after you changed some code.
Now you have successfully setup Eclipse and can build your targets using the Make Target view (usually on the right hand side of the workbench). To build RazerS 2, open the folder core > apps > razers2 in the Make Target view. Then, right-click on exe] razers2 and select Build Target. This will build the program razers2.
The last line you see on the console at the bottom of the screen should be

```
Built target razers2
make[2]: Leaving directory `/home/USER/Development/seqan-trunk-build/debug-eclipse`
```

**Hello World!** Now, let us create a sandbox for you. This sandbox will be your local workspace and you might want to have it versionized on your own Subversion repository at a later point. All of your development will happen in your sandbox.

We go back to the source directory and then use the SeqAn code generator to create a new sandbox.

```
debug-eclipse # cd ../../seqan-trunk
seqan-trunk # ./util/bin/skel.py repository sandbox/my_sandbox
```

Now that you have your own working space, we create a new application `first_app`.

```
seqan-trunk # ./util/bin/skel.py app first_app sandbox/my_sandbox
```

Details about the code generator are explained in *Using The Code Generator*.

Now, we go back into the build directory and call CMake again to make it detect the added app.

```
segan-trunk # cd ../seqan-trunk-build/debug-eclipse
debug # cmake .
```

**Tip:** When and where do you have to call CMake?
CMake is a cross-platform tool for creating and updating build files (IDE projects or Makefiles). When you first create the build files, you can configure things such as the build mode or the type of the project files.

Whenever you add a new application, a demo or a test or whenever you make changes to CMakeLists.txt you need to call CMake again. Since CMake remembers the settings you chose the first time you called CMake in a file named CMakeCache.txt, all you have to do is to switch to your debug or release build directory and call “cmake .” in there.

~ # cd $HOME/Development/seqan-trunk-build/debug
debug # cmake .

Do not try to call “cmake .” from within the seqan-trunk directory but only from your build directory.

The step above creates the starting point for a real-world application, including an argument parser and several other things that are a bit too complicated to fit into the Getting Started tutorial. Therefore, we will replace the program of the app first_app with a very simple example program.

Open the file sandbox/my_sandbox/apps/first_app/first_app.cpp (in seqan-trunk) with a text editor and replace its contents with the following:

```cpp
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h> // to stream a CharString into cout

int main(int, char const **) { 
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```

After CMake is done with building the project files you can simply refresh the imported build project in Eclipse in order to make the applied changes working: In the Project Explorer view, right-click on seqan-Debug@debug-eclipse and select Refresh (F5).

**Tip:** In some cases the refresh does not work. In this case simply close and reopen Eclipse.

Then, browse to your target in the Make Target view. Double-click on the target of your application and the binary is built immediately.

After the build is complete go into the menu and select Run > Run Configurations. Add a new C/C++ Application. Under Project select the seqan-build project. Then you can select your recently built binary under C/C++ Application:. Finally, you can give your application a name. Confirm your selection with Apply and hit the Run button in the bottom of the dialog. Done!
If everything went well, you will see something similar to this:

Congratulations, you have successfully created your first application within the SeqAn build system using Eclipse and CDT.

Further Steps As a next step, we suggest the following:

- Continue with the Tutorials
- Look around in the files in sandbox/my_sandbox/apps/first_app or the demos in core/demos and extras/demos.
- For the tutorial, using the SeqAn build system is great! If you later want to use SeqAn as a library, have a look at Integration with your own Build System.
Getting Started With SeqAn On Mac Os X Using Makefiles

This tutorial explains how to get started with SeqAn on Mac Os X using Makefiles.

We assume that you want to use MacPorts for installing some dependencies (MacPorts is a package management system that easily allows you to install Unix software on Os X). Of course, if you want to use a different way for installing the dependencies (e.g. using Homebrew) then you are free to do so.

**Prerequisites**

First, you have to install the Apple Xcode SDK (Apple ID needed).

**Warning:** Please choose Xcode SDK version 4.2 or lower, because the current version has some compatibility problems with the SeqAn build system.

After installing the Xcode SDK, please install MacPorts following these instructions. To check that the MacPorts install was successful, enter the following on your shell. If the `port` program is found then you can go on.

```
# port info
```

Next, install CMake and Subversion using the `port` command.

```
# port install cmake
# port install subversion
```

**Install**

Now, go to the directory you want to keep your SeqAn install in (e.g. Development in your home folder).

```
~ # cd $HOME/Development
```

Then, use Subversion to retrieve the current SeqAn trunk:

```
Development # svn co https://github.com/seqan/seqan/branches/master seqan-trunk
```

You can now find the whole tree with the SeqAn library and applications in `seqan-trunk`.

**A First Build**

Next, we will use CMake to create Makefiles for building the applications, demo programs (short: demos), and tests. For this, we create a separate folder `seqan-trunk-build` on the same level as the folder `seqan-trunk`.

```
Development # mkdir seqan-trunk-build
```
When using Makefiles, we have to create separate Makefiles for debug builds (including debug symbols with no optimization) and release builds (debug symbols are stripped, optimization is high). Thus, we create a subdirectory for each build type. We start with debug builds since this is best for learning: Debug symbols are enabled and assertions are active.

**Warning:** Compiling "debug mode yields very slow binaries" since optimizations are disabled. Compile your programs in release mode if you want to run them on large data sets. The reason for disabling optimizations in debug mode is that the compiler performs less inlining and does not optimize variables away. This way, debugging your programs in a debugger becomes much easier.

Development # mkdir seqan-trunk-build/debug
Development # cd seqan-trunk-build/debug

The resulting directory structure will look as follows.

```
~/Development
  |-- seqan-trunk source directory
  `-- seqan-trunk-build
      `-- debug build directory with debug symbols
```

Within the **build directory** `debug`, we use CMake to generate Makefiles in **Debug** mode.

```
debug # cmake ../../seqan-trunk -DCMAKE_BUILD_TYPE=Debug
```

We can then build one application, for example RazerS 2:

```
debug # make razers2
```

 Optionally, we could also use "make" instead of "make razers2". However, this can take a long time and is not really necessary.

**Hello World!** Now, let us create a **sandbox** for you. This sandbox will be your local workspace and you might want to have it versionized on your own Subversion repository at a later point. All of your development will happen in your sandbox.

We go back to the source directory and then use the SeqAn code generator to create a new sandbox.

```
debug # cd ../../seqan-trunk
seqan-trunk # ./util/bin/skel.py repository sandbox/my_sandbox
```

Now that you have your own working space, we create a new application **first_app**.

```
seqan-trunk # ./util/bin/skel.py app first_app sandbox/my_sandbox
```

Details about the code generator are explained in *Using The Code Generator*.

Now, we go back into the build directory and call CMake again to make it detect the added app.

```
seqan-trunk # cd ../seqan-trunk-build/debug
debug # cmake .
```

**Tip:** When and where do you have to call CMake?

CMake is a cross-platform tool for creating and updating build files (IDE projects or Makefiles). When you first create the build files, you can configure things such as the build mode or the type of the project files.

Whenever you add a new application, a demo or a test or whenever you make changes to `CMakeLists.txt` you need to call CMake again. Since CMake remembers the settings you chose the first time you called CMake in a file.
named `CMakeCache.txt`, all you have to do is to switch to your `debug` or `release` build directory and call “`cmake .`” in there.

Do not try to call “`cmake .`” from within the `seqan-trunk` directory but only from your build directory.

The step above creates the starting point for a real-world application, including an argument parser and several other things that are a bit too complicated to fit into the Getting Started tutorial. Therefore, we will replace the program of the app `first_app` with a very simple example program.

Open the file `sandbox/my_sandbox/apps/first_app/first_app.cpp` (in your `seqan-trunk` directory) with a text editor and replace its contents with the following:

```cpp
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h>    // to stream a CharString into cout

int main(int, char const **) {
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```

Afterwards, you can simply compile and run your application:

```
debug # make first_app
debug # ./bin/first_app
```

On completion, you should see the following output:

```
Hello World!
Hello SeqAn!
```

Congratulations, you have successfully created your first application within the SeqAn build system with Makefiles!

Further Steps  As a next step, we suggest the following:

- Continue with the Tutorials
- Look around in the files in `sandbox/my_sandbox/apps/first_app` or the demos in `core/demos` and `extras/demos`.
- For the tutorial, using the SeqAn build system is great! If you later want to use SeqAn as a library, have a look at Integration with your own Build System.
Getting Started With SeqAn On Mac OS X Using Xcode

This article describes how to get started with SeqAn on Mac OS X using XCode.

We assume that you want to use MacPorts for installing some dependencies (MacPorts is a package management system that easily allows you to install Unix software on OS X). Of course, if you want to use a different way for installing the dependencies (e.g. using Homebrew) then you are free to do so.

Prerequisites

First, you have to install the Apple Xcode via Apple Developer or preferably Xcode via the Mac App Store.

After having done so you also need to install the Command Line Tools. Open Xcode, go to Xcode > Preferences... or simply press to open the preferences. Switch to the Downloads tab. In the shown table you will find the Command Line Tools. Click on Install to install them.

Warning: If you already had Xcode installed and you updated to Xcode 4.3 it is likely that xcode-select is not pointing to the new Xcode location. To make sure you can properly work with Xcode 4.3 please open your Terminal and type:

```
~ # xcode-select -print-path
```

The command should output /Applications/Xcode.app/Contents/Developer. If it doesn’t please note down the actual output (for backup purpose) and type in:
This will configure \texttt{xcode-select} properly and make it work with SeqAn.

\begin{quote}
\textbf{Warning:} With the current MacPort version of CMake it is no more possible to open sub projects. Either open only the root project \texttt{seqan.xcodeproj} or install our \texttt{cmake-2.8.9.20120903-g6ef858-dirty-Darwin-i386.dmg} patched version of CMake instead.

\texttt{Note:} The package installs the binary at \texttt{/usr/bin/cmake}. Please add the following line to your \texttt{~/.profile} file:

\begin{verbatim}
export PATH=/usr/bin:$PATH
\end{verbatim}
\end{quote}

After installing the Xcode SDK, please install MacPorts following these instructions. To check that the MacPorts install was successful, enter the following on your shell. If the \texttt{port} program is found then you can go on.

\begin{verbatim}
~ # port info
\end{verbatim}

Next, install Subversion using the \texttt{port} command.

~ # sudo port install subversion

There is a problem with the current version of CMake. Please read Problem with CMake box above and either install our patched version of CMake (see attachments) or the current MacPort version which has a problem with sub projects:

~ # sudo port install cmake

Install Now, go to the directory you want to keep your SeqAn install in (e.g. Development in your home folder).

~ # cd $HOME/Development

Then, use Subversion to retrieve the current SeqAn trunk:

Development # svn co https://github.com/seqan/seqan/branches/master seqan-trunk

You can now find the whole tree with the SeqAn library and applications in \texttt{$HOME/Development/seqan-trunk}.

A First Build Next, we will use CMake to create an Xcode project for building the applications, demo programs (short: demos), and tests. For this, we create a separate folder \texttt{seqan-trunk-build} on the same level as the folder \texttt{seqan-trunk}.

\begin{verbatim}
# Development # mkdir -p seqan-trunk-build/xcode
\end{verbatim}

The resulting directory structure will look as follows.

~/Development
- seqan-trunk source directory
  seqan-trunk-build build directory
- xcode

Within the build directory \texttt{xcode}, we call CMake to generate Xcode project files.

Development # cd seqan-trunk-build/xcode
xcode # cmake ..../../seqan-trunk -G Xcode
This will generate several Xcode project files in `xcode`, namely for the SeqAn core applications, demos, and tests. The same will happen for the extras and sandbox directories.

Now, open the project for the core applications:

```bash
# xcode # open seqan.xcodeproj
```

Xcode starts and will look like this:

![Xcode screenshot](image)

Now we are ready to compile and run our first application. For this, please choose the target `razers2` in the top left corner of your Xcode application. When selected click on `Run` just left to where you chose the target.

![Target selection](image)

Optionally, we could also use “ALL_BUILD” instead of “razers2”. However, this can take a long time and is not
really necessary.

After having compiled and run `razers2` your Xcode should display `razers2`'s output in the bottom right area.

Hello World!  Now, let us create a sandbox for you. This sandbox will be your local workspace and you might want to have it versionized on your own Subversion repository at a later point. All of your development will happen in your sandbox.

We go back to the source directory and then use the SeqAn code generator to create a new sandbox.

```bash
xcode # cd ../../seqan-trunk
seqan-trunk # ./util/bin/skel.py repository sandbox/my_sandbox
```

Now that you have your own working space, we create a new application `first_app`.

```bash
seqan-trunk # ./util/bin/skel.py app first_app sandbox/my_sandbox
```

Details about the code generator are explained in *Using The Code Generator*.

Now, we go back into the build directory and call CMake again to make it detect the added app.

```bash
seqan-trunk # cd ../seqan-trunk-build/xcode
xcode # cmake .
```

**Tip:**

When and where do you have to call CMake?

CMake is a cross-platform tool for creating and updating build files (IDE projects or Makefiles). When you first create the build files, you can configure things such as the build mode or the type of the project files.
Whenever you add a new application, a demo or a test or whenever you make changes to CMakeLists.txt you need to call CMake again. Since CMake remembers the settings you chose the first time you called CMake in a file named CMakeCache.txt, all you have to do is to switch to your debug or release build directory and call “cmake .” in there.

~ # cd $HOME/Development/seqan-trunk-build/xcode
debug # cmake .

Do not try to call “cmake .” from within the seqan-trunk directory but only from your build directory.

The step above creates the starting point for a real-world application, including an argument parser and several other things that are a bit too complicated to fit into the Getting Started tutorial. Therefore, we will replace the program of the app first_app with a very simple example program.

Select the file /Sources/first_app/Source Files/first_app.cpp in Xcode and open it:

```
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h> // to stream a CharString into cout

int main(int, char const **)
{
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```

Replace its contents with the following:

```cpp
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h> // to stream a CharString into cout

int main(int, char const **)
{
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```
Afterwards, you can simply compile and run your application by clicking on the Run button on the very top left corner in Xcode.

On completion, you should see the following output:

Hello World!
Hello SeqAn!

Congratulations, you have successfully created your first application within the SeqAn build system with Xcode!

Further Steps As a next step, we suggest the following:

- **Continue with the Tutorials**
- Look around in the files in `sandbox/my_sandbox/apps/first_app` or the demos in `core/demos` and `extras/demos`. 
- For the tutorial, using the SeqAn build system is great! If you later want to use SeqAn as a library, have a look at **Integration with your own Build System**.
Getting Started With SeqAn On Windows Using Visual Studio

This tutorial explains how to get started with SeqAn on Windows using Visual Studio. We use Visual Studio 9 here, but you should be able to easily follow the tutorial if you are using any other supported version (8, 9, 10).

Prerequisites  We assume that you have the following software installed:

- TortoiseSVN for version control and retrieving
- Microsoft Visual Studio for compiling C++ code. If you do not have Visual Studio yet, you can download the free Visual Studio Express Edition from Microsoft. Note that at some point, you can pick between the Express Edition and a demo of the full version. Pick the Express Edition here for a smaller and continuously free version.
- CMake for generating project files
- Python 2.x for the code generator. Note that Python 3 will not work, thus use a Python 2 version.

Also, we assume that your user name is seqan_dev. Replace seqan_dev by your actual user name in the steps below.

Warning: When installing CMake and Python, make sure to let the installers add the paths to the cmake.exe and python.exe binaries to your PATH environment variable.

Install  First, create a new folder in your Windows Explorer for the SeqAn files (e.g. C:\Users\seqan_dev\Development\seqan-trunk). Right click on this folder and choose SVN Checkout... from the context menu.
Type the following in the field **URL of repository**

https://github.com/seqan/seqan/branches/master

The **Checkout repository** should be the folder you just created. Leave the other settings set to their default (**fully recursive, HEAD revision**) and click **OK**.

**A First Build**  
Next, we will use CMake to create a Visual Studio 9 project for building the applications, demo programs (short: demos), and tests.

We will do this using the command prompt. Start a Windows command prompt (`Start > All Programs > Accessories > Command Prompt`). Then, go to the parent directory of `seqan-trunk`. There, we create a separate folder `seqan-trunk-build`:

```
C:\Users\seqan_dev> C:
C:\Users\seqan_dev> cd C:\Users\seqan_dev\Development
C:\Users\seqan_dev\Development> mkdir seqan-trunk-build
```

In the future, we might create a different set of project files (e.g. when we upgrade our Visual Studio version or switch to 64 bit builds). Thus, we create a subdirectory called `vs9` for our Visual Studio 9 project.
C:\Users\seqan_dev\Development> mkdir seqan-trunk-build\vs9
C:\Users\seqan_dev\Development> cd seqan-trunk-build\vs9

The resulting directory structure will look as follows.

~/Development
  - seqan-trunk           source directory
  - seqan-trunk-build
    - vs9                  Visual Studio 9 project

Within the build directory vs9, we call CMake to generate the Visual Studio 9 project:

C:\Users\seqan_dev\Development\seqan-trunk-build\vs9> cmake ..\..\seqan-trunk -G "Visual Studio 9 2008"

---

**Tip:** Using a different version or multiple versions of Visual Studio.

Using a different Visual Studio version is easy: Simply use the appropriate generator. For example, here is how to use the Visual Studio 10 generator:

C:\...\seqan-trunk-build\vs10> cmake ..\..\seqan-trunk -G "Visual Studio 10"

Also, if you want to create 64 bit builds, you need to use another generator. Changing generators in an existing project directory is not possible.

- If you use another version of Visual Studio, you can find out the correct parameter by typing cmake --help.
- If you plan to use different versions of Visual Studio then follow Using Parallel Build Directories.
- Note that you have to choose the Win64 variants of the Visual Studio generators if you want to build 64 bit binaries.

---

Now, you can open the project file from the Windows explorer: go to the C:\Users\seqan_dev\Development\seqan-trunk-build\vs9\core\apps directory and double click on the file seqan_core_apps.sln. In the Solution Explorer to the left you will find all applications of the SeqAn core.
As an example you can open, compile, and execute the program RazerS 2. Click on the +/ in front of razers2 in the Solution Explorer and also on the +/ in front of Source Files. Double click on the file razers.cpp to open it in the main window. Right click on razers2 in the Solution Explorer and choose Set as StartUp Project from the context menu.
To compile and execute RazerS 2, choose Debug > Start Without Debugging from the main menu or press Ctrl + F5.

On successful compilation and execution, a terminal windows should pop up and display:

***********************************************************
*** RazerS - Fast Read Mapping with Sensitivity Control ***
*** (c) Copyright 2009 by David Weese ***
***********************************************************
Usage: razers2.exe [OPTION]... <GENOME FILE> <READS FILE>
razers2.exe [OPTION]... <GENOME FILE> <MP-READS FILE1> <MP-READS FILE2>
Try ‘razers2.exe --help’ for more information.
Press any key to continue . . .

Tip: ‘Starting with and without Debugging in Visual Studio
Starting a program without debugger in Visual Studio will make the program wait for the user pressing a key. This is very useful for inspecting the output. In Debug mode, the program does not wait when executed and the window
simply disappears. The reason for this is that when using the debugger, the user is expected to set a breakpoint.

---

**Hello World!**  Now, let us create a sandbox for you. This sandbox will be your local workspace and you might want to have it versionized on your own Subversion repository at a later point. All of your development will happen in your sandbox.

We go back to the source directory and then use the SeqAn code generator to create a new sandbox.

```
C:\Users\seqan_dev\Development\seqan-trunk-build\vs9> cd ..\..\seqan-trunk
C:\Users\seqan_dev\Development\seqan-trunk> python util\bin\skel.py repository sandbox\my_sandbox
```

Within this sandbox, we can now create a new application using the code generator.

```
C:\Users\seqan_dev\Development\seqan-trunk> python util\bin\skel.py app first_app sandbox\my_sandbox
```

Details about the code generator are explained in *Using The Code Generator*.

**Tip:** When and where do you have to call CMake?

CMake is a cross-platform tool for creating and updating build files (IDE projects or Makefiles). When you first create the build files, you can configure things such as the build mode or the type of the project files.

Whenever you add a new application, a demo or a test or whenever you make changes to CMakeLists.txt you need to call CMake again. Since CMake remembers the settings you chose the first time you called CMake in a file named CMakeCache.txt, all you have to do is to switch to your debug or release build directory and call “cmake .” in there.

```
C:\ # cd C:\Users\seqan_dev\Development\seqan-trunk-build\vs9
vs9 # cmake .
```

Do not try to call “cmake .” from within the seqan-trunk directory but only from your build directory.

The step above creates the starting point for a real-world application, including an argument parser and several other things that are a bit too complicated to fit into the Getting Started tutorial. Therefore, we will replace the program of the app first_app with a very simple example program.

Open the file `C:\Users\seqan_dev\Development\seqan-trunk\sandbox\my_sandbox\first_app\first_app.cpp` with the Visual Studio text editor and replace its contents with the following:

```cpp
#include <iostream>
#include <seqan/sequence.h> // CharString, ...
#include <seqan/file.h> // to stream a CharString into cout

int main(int, char const **) {
    std::cout << "Hello World!" << std::endl;
    seqan::CharString mySeqAnString = "Hello SeqAn!";
    std::cout << mySeqAnString << std::endl;
    return 1;
}
```

Now, we go back into the build directory and call CMake again to make it detect the added app.

```
C:\Users\seqan_dev\Development\seqan-trunk\sandbox\my_sandbox\first_app\first_app.sln
C:\Users\seqan_dev\Development\seqan-trunk-build\vs9> cmake .
```

Now, you can open the project file `C:\Users\seqan_dev\Development\seqan-trunk-build\vs9\sandbox\my_sandbox\first_app\first_app.sln` for your sandbox using Visual Studio. Right-click first_app in the Solution Explorer and click Set
as StartUp Project. Now, start the program without debugging using Debug > Start Without Debugging.

Visual Studio will now compile your program and execute it. A command line window will pop up and should display the following.

Hello World!
Hello SeqAn!
Press any key to continue . . .

Congratulations, you have successfully created your first application within the SeqAn build system using Visual studio.

Further Steps As a next step, we suggest the following:

- * Continue with the Tutorials
- If you have not done so, install optional dependencies of SeqAn. To read and write compressed files, follow Installing Contribs On Windows.
- Look around in the files in sandbox/my_sandbox/apps/first_app or the demos in core/demos and extras/demos.
- For the tutorial, using the SeqAn build system is great! If you later want to use SeqAn as a library, have a look at Integration with your own Build System.

2.1.2 A First Example

Learning Objective You will learn the most basic concepts of SeqAn. After this tutorial you will be ready to deal with the more specific tutorials, e.g. Sequences.

Difficulty Very basic

Duration 1.5h
Prerequisites  Basic C or C++ knowledge

Welcome to the SeqAn “Hello World”. This is the first practical tutorial you should look at when starting to use our software library.

We assume that you have some programming experience (preferably in C++ or C) and concentrate on SeqAn specific aspects. We will start out pretty slowly and hopefully the tutorial will make sense to you even if you are new to C++. However, to really leverage the power of SeqAn you will have to learn C++. There are many tutorials on C++, for example the tutorial at cplusplus.com.

This tutorial will walk you through a simple example program that highlights the things that are most prominently different from the libraries that many SeqAn newcomers are used to:

- extensive usage of C++ templates,
- generic programming using templates,
- using references instead of pointers in most places,
- and more.

Running Example

Our example program will do a pattern search of a short query sequence in a long database sequence. As the score, we will compute the number of equal characters at each position.

The following figure shows an example:

```
| score: 101 ... ... 801 ... |
| text: This is an awesome tutorial to get to know SeqAn! |
| pattern: tutorial tutorial |
| tutorial tutorial |
| ... ... |
```

The first position has a score of 1, because the i in the pattern matches the i in is. This is only a toy example for explanatory reasons and we ignore any more advanced implementations.

In SeqAn the program could look like this (we will explain every line of code shortly):

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

int main()
{

    // Initialization
    seqan::String<
    char>
    text = "This is an awesome tutorial to get to know SeqAn!";
    seqan::String<
    char>
    pattern = "tutorial";

    seqan::String<
    int>
    score;
    resize(score, seqan::length(text) - seqan::length(pattern) + 1);

    // Computation of the similarities
    // Iteration over the text (outer loop)
    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
    {
        int localScore = 0;
        // Iteration over the pattern for character comparison
        for (unsigned j = 0; j < seqan::length(pattern); ++j)
        {
            if (text[i + j] == pattern[j])
            {
                ++localScore;
            }
        }
        score[i] = localScore;
    }

    for (unsigned i = 0; i < seqan::length(text); ++i)
    {
        if (i < seqan::length(pattern))
        {
            std::cout << text[i] << "\n";
        }
        else
        {
            std::cout << "\n";
        }
        i += seqan::length(pattern);
    }

    std::cout << "\n";
    for (unsigned i = 0; i < seqan::length(text); ++i)
    {
        std::cout << score[i] << "\n";
    }

    return 0;
}
```
++localScore;

}  

score[i] = localScore;

}

// Printing the result

for (unsigned i = 0; i < seqan::length(score); ++i)

std::cout << score[i] << " ";
std::cout << std::endl;

return 0;

}

Whenever we use SeqAn classes or functions we have to explicitly write the namespace qualifier seqan:: in front of the class name or function. This can be circumvented if we include the line using namespace seqan; at the top of the working example. However, during this tutorial we will not do this, such that SeqAn classes and functions can be recognized more easily.

Attention: Argument-Dependent Name Lookup (Koenig Lookup)

Using the namespace prefix seqan:: is not really necessary in all places. In many cases, the Koenig lookup rule in C++ for functions makes this unnecessary. Consider the following, compiling, example.

seqan::String<char> s = "example";
unsigned i = length(s);

Here, the function length does not have a namespace prefix. The code compiles nevertheless. The compiler automatically looks for a function length in the namespace of its arguments.

Note that we follow the rules for variable, function, and class names as outlined in the SeqAn style guide. For example: (1) variables and functions use lower case, (2) struct, enum and classes use CamelCase, (3) metafunctions start with a capital letter, and (4) metafunction values are UPPERCASE.

Assignment 1

Type  Review

Objective  Create a demo program and replace its content with the code above.

Hint  Depending on your operating system you have different alternatives to create a demo application. An in depth description can be found in GettingStarted.

Solution  Click ‘more...’

// Copy the code into a demo program and have a look at the result.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

int main()
{
    // Initialization
    seqan::String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    seqan::String<char> pattern = "tutorial";

    seqan::String<int> score;
    resize(score, seqan::length(text) - seqan::length(pattern) + 1);
// Computation of the similarities
// Iteration over the text (outer loop)
for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
{
    int localScore = 0;
    // Iteration over the pattern for character comparison
    for (unsigned j = 0; j < seqan::length(pattern); ++j)
    {
        if (text[i + j] == pattern[j])
            ++localScore;
    }
    score[i] = localScore;
}

// Printing the result
for (unsigned i = 0; i < seqan::length(score); ++i)
    std::cout << score[i] << " ";
std::cout << std::endl;
return 0;

SeqAn and Templates

Let us now have a detailed look at the program.

We first include the IOStreams library that we need to print to the screen and the SeqAn’s <seqan/file.h> as well as <seqan/sequence.h> module from the SeqAn library that provides SeqAn String.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

The String class is one of the most fundamental classes in SeqAn, which comes as no surprise since SeqAn is used to analyse sequences (there is an extra tutorial for SeqAn sequences and alphabets).

In contrast to the popular string classes of Java or C++, SeqAn provides different string implementations and different alphabets for its strings. There is one string implementation that stores characters in memory, just like normal C++ strings. Another string implementation stores the characters on disk and only keeps a part of the sequence in memory. For alphabets, you can use strings of nucleotides, such as genomes, or you can use strings of amino acids, for example.

SeqAn uses template functions and template classes to implement the different types of strings using the generic programming paradigm. Template functions/classes are normal functions/classes with the additional feature that one passes the type of a variable as well as its value (see also: templates in cpp). This means that SeqAn algorithms and data structures are implemented in such a way that they work on all types implementing an informal interface (see information box below for more details). This is similar to the philosophy employed in the C++ STL (Standard Template Library). Even though we provide further tutorials on templates in SeqAn (Basic Techniques, Metafunctions, Generic Programming, Global Function Interface), they are more advanced and not required to follow this tutorial.

The following two lines make use of template programming to define two strings of type char, a text and a pattern.

    // Initialization
    seqan::String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    seqan::String<char> pattern = "tutorial";
In order to store the similarities between the pattern and different text positions we additionally create a string storing integer values.

```cpp
seqan::String<int> score;
```

Note that in contrast to the first two string definitions we do not know the values of the different positions in the string in advance. In order to dynamically adjust the length of the new string to the text we can use the function `resize`. The resize function is not a member function of the string class because SeqAn is not object oriented in the typical sense (we will see later how we adapt SeqAn to object oriented programming). Therefore, instead of writing `string.resize(newLength)` we use `resize(string, newLength).

```cpp
resize(score, seqan::length(text) - seqan::length(pattern) + 1);
```

**Note:** Global function interfaces.

SeqAn uses global interfaces for its data types/classes. Generally, you have to use `function(variable)` instead of `variable.function()`.

This has the advantage that we can extend the interface of a type outside of its definition. For example, we can provide a `length()` function for STL containers `std::string<T>` and `std::vector<T>` outside their class files. We can use such global functions to make one data type have the same interface as a second. This is called adaptation.

Additionally, we can use one function definition for several data types. For example, the alignment algorithms in SeqAn are written such that we can compute alignments using any String with any alphabet: There are more than 5 String variants in SeqAn and more than 8 built-in alphabets. Thus, one implementation can be used for more than 40 different data types!

After the string initializations it is now time for the similarity computation. In this toy example we simply take the pattern and shift it over the text from left to right. After each step, we check how many characters are equal between the corresponding substring of the text and the pattern. We implement this using two loops; the outer one iterates over the given text and the inner loop over the given pattern:

```cpp
// Computation of the similarities
// Iteration over the text (outer loop)
for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
{
    int localScore = 0;
    // Iteration over the pattern for character comparison
    for (unsigned j = 0; j < seqan::length(pattern); ++j)
    {
        if (text[i + j] == pattern[j])
            ++localScore;
    }
    score[i] = localScore;
}
```

There are two things worth mentioning here: (1) SeqAn containers or strings start at position 0 and (2) you will notice that we use `++variable` instead of `variable++` wherever possible. The reason is that `++variable` is slightly faster than its alternative, since the alternative needs to make a copy of itself before returning the result.

In the last step we simply print the result that we stored in the variable `score` on screen. This gives the similarity of the pattern to the string at each position.

```cpp
// Printing the result
for (unsigned i = 0; i < seqan::length(score); ++i)
    std::cout << score[i] << " ";
std::cout << std::endl;
```
Code Encapsulation

At this point, we have already created a working solution! However, in order to make it easier to maintain and reuse parts of the code we need to export them into functions. In this example the interesting piece of code is the similarity computation, which consists of an outer and inner loop. We encapsulate the outer loop in function `computeScore` and the inner loop in function `computeLocalScore` as can be seen in the following code.

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

int computeLocalScore(seqan::String<char> subText, seqan::String<char> pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;
    return localScore;
}

seqan::String<int> computeScore(seqan::String<char> text, seqan::String<char> pattern)
{
    seqan::String<int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);
    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);
    return score;
}

int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);
    for (unsigned i = 0; i < seqan::length(score); ++i)
        std::cout << score[i] << " ";
    std::cout << std::endl;
    return 0;
}
```

The function `computeScore()` now contains the fundamental part of the code and can be reused by other functions. The input arguments are two strings. One is the pattern itself and one is a substring of the text. In order to obtain the substring we can use the function `infix` implemented in SeqAn. The function call `infix(text, i, j)` generates a substring equal to `text[i ... j - 1]`, e.g. `infix(text, 1, 5)` equals “ello”, where `text` is “Hello World”. To be more precise, `infix()` generates a `Infix` which can be used as a string, but is implemented using pointers such that no copying is necessary and running time and memory is saved.

Assignment 2

Type Review

Objective Replace the code in your current file by the code above and encapsulate the print instructions.
Hint  The function head should look like this:

Solution

    // Copy the code into your current file and encapsulate the print instructions.
    #include <iostream>
    #include <seqan/file.h>
    #include <seqan/sequence.h>

    int computeLocalScore(seqan::String<char> subText, seqan::String<char> pattern)
    {
        int localScore = 0;
        for (unsigned i = 0; i < seqan::length(pattern); ++i)
            if (subText[i] == pattern[i])
                ++localScore;
        return localScore;
    }

    seqan::String<int> computeScore(seqan::String<char> text, seqan::String<char> pattern)
    {
        seqan::String<int> score;
        seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);
        for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
            score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);
        return score;
    }

    void print(seqan::String<int> text)
    {
        for (unsigned i = 0; i < seqan::length(text); ++i)
            std::cout << text[i] << " ";
        std::cout << std::endl;
    }

    int main()
    {
        seqan::String<char> text = "This is an awesome tutorial to get to now SeqAn!";
        seqan::String<char> pattern = "tutorial";
        seqan::String<int> score = computeScore(text, pattern);

        print(score);

        return 0;
    }

The Role of References in SeqAn

Let us now have a closer look at the signature of computeScore().

Both the text and the pattern are passed by value. This means that both the text and the pattern are copied when the function is called, which consumes twice the memory. This can become a real bottleneck since copying longer sequences is very memory and time consuming, think of the human genome, for example.

Instead of copying we could use references. A reference in C++ is created using an ampersand sign (&) and creates an alias to the referenced value. Basically, a reference is a pointer to an object which can be used just like the referenced
object itself. This means that when you change something in the reference you also change the original object it came from. But there is a solution to circumvent this modification problem as well, namely the word \texttt{const}. A \texttt{const} object cannot be modified.

**Important**: If an object does not need to be modified make it an nonmodifiably object using the keyword \texttt{const}. This makes it impossible to \textit{unwillingly} change objects, which can be really hard to debug. Therefore it is recommended to use it as often as possible.

Therefore we change the signature of \texttt{computeScore} to:

\begin{verbatim}
seqan::String<int> computeScore(seqan::String<char> const & text, seqan::String<char> const & pattern)
\end{verbatim}

Reading from right to left the function expects two references to \texttt{const} objects of type \texttt{String} of \texttt{char}.

**Assignment 3**

**Type Review**

**Objective** Adjust your current code to be more memory and time efficient by using references in the function header.

**Hint** The function head for \texttt{computeLocalScore} should look like this:

\begin{verbatim}
void computeLocalScore(seqan::String<char> const & subText, seqan::String<char> const & pattern)
\end{verbatim}

**Solution**

\begin{verbatim}
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

int computeLocalScore(seqan::String<char> const & subText, seqan::String<char> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;
    return localScore;
}

seqan::String<int> computeScore(seqan::String<char> const & text, seqan::String<char> const & pattern)
{
    seqan::String<int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);
    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);
    return score;
}

void print(seqan::String<int> const & text)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
\end{verbatim}
int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);
    print(score);
    return 0;
}

Generic and Reusable Code

As mentioned earlier, there is another issue: the function computeScore only works for Strings having the alphabet char. If we wanted to use it for Dna or AminoAcid strings then we would have to reimplement it even though the only difference is the signature of the function. All used functions inside computeScore can already handle the other datatypes.

The more appropriate solution is a generic design using templates, as often used in the SeqAn library. Instead of specifying the input arguments to be references of strings of char s we could use references of template arguments as shown in the following lines:

    template <typename TText, typename TPattern>
    seqan::String<int> computeScore(TText const & text, TPattern const & pattern)

The first line above specifies that we create a template function with two template arguments TText and TPattern. At compile time the template arguments are then replace with the correct types. If this line was missing the compiler would expect that there are types TText and TPattern with definitions.

Now the function signature is better in terms of memory consumption, time efficiency, and generality.

Important: The SeqAn Style Guide

The SeqAn style guide gives rules for formatting and structuring C++ code as well as naming conventions. Such rules make the code more consistent, easier to read, and also easier to use.

1. Naming Scheme. Variable and function names are written in lowerCamelCase, type names are written in UpperCamelCase. Constants and enum values are written in UPPER_CASE. Template variable names always start with ‘T’.

2. Function Parameter Order. The order is (1) output, (2) non-const input (e.g. file handles), (3) input, (4) tags. Output and non-const input can be modified, the rest is left untouched and either passed by copy or by const-reference (const &).

3. Global Functions. With the exception of constructors and a few operators that have to be defined in-class, the interfaces in SeqAn use global functions.


While we are trying to make the interfaces consistent with our style guide, some functions have incorrect parameter order. This will change in the near future to be more in line with the style guide.

Assignment 4

Type Review
Objective  Generalize the `computeLocalScore` function in your file.

Solution

```cpp
// Generalize the computeLocalScore function in you file.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;
    return localScore;
}

template <typename TText, typename TPattern>
seqan::String<int> computeScore(TText const & text, TPattern const & pattern)
{
    seqan::String<int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);
    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);
    return score;
}

void print(seqan::String<int> const & text)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);
    print(score);
    return 0;
}
```

From Object-Oriented Programming to SeqAn

There is another huge advantage of using templates: we can specialize a function without touching the existing function. In our working example it might be more appropriate to treat `AminoAcid` sequences differently. As you probably know, there is a similarity relation on amino acids: Certain amino acids are more similar to each other, than others. Therefore we want to score different kinds of mismatches differently. In order to take this into consideration we simple write a `computeLocalScore()` function for `AminoAcid` strings. In the future whenever ‘comput-Score’ is called always the version above is used unless the second argument is of type String. Note that the second
template argument was removed since we are using the specific type String-AminoAcid.

```cpp
template <typename TText>
int computeLocalScore(TText const & subText, seqan::String<String<AminoAcid>> const & pattern) {
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        localScore += seqan::score(seqan::Blosum62(), subText[i], pattern[i]);

    return localScore;
}
```

In order to score a mismatch we use the function `score()` from the SeqAn library. Note that we use the Blosum62 matrix as a similarity measure. When looking into the documentation of `score` you will notice that the score function requires an argument of type `Score`. This object tells the function how to compare two letters and there are several types of scoring schemes available in SeqAn (of course, you can extend this with your own). In addition, because they are so frequently used there are shortcuts as well. For example, `Blosum62` is really a shortcut for `Score<int, ScoreMatrix<AminoAcid, Blosum62_>` , which is obviously very helpful. Other shortcuts are `DnaString` for `String<Dna>` (sequence tutorial), `CharString` for `String<char>`,...

**Tip:** Template Subclassing

The main idea of template subclassing is to exploit the C++ template matching mechanism. For example, in the following code, the function calls (1) and (3) will call the function `myFunction()` in variant (A) while the function call (2) will call variant (B).

```cpp
struct SpecA;
struct SpecB;
struct SpecC;

template <typename TAlphabet, typename TSpec>
class String<TAlphabet, TSpec>;

template <typename TAlphabet, typename TSpec>
void myFunction(String<TAlphabet, TSpec> const & str); // Variant (A)

template <typename TAlphabet>
void myFunction(String<TAlphabet, SpecB> const & str); // Variant (B)

int main()
{
    String<char, SpecA> a;
    String<char, SpecB> b;
    String<char, SpecC> c;

    myFunction(a);   // calls (A)
    myFunction(b);   // calls (B)
    myFunction(c);   // calls (A)
}
```

For a detailed description and more examples see the tutorial *Template Subclassing.*

**Assignment 5**

**Type Application**
**Objective**  Provide a generic print function which is used when the input type is not `String<int>`.

**Hint**  Keep your current implementation and add a second function. Don’t forget to make both template functions.

Include `<seqan/score.h>` as well.

**Solution**

```cpp
// Provide a generic print function which is used when the input type is not String<int>.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

template<typename TText>
int computeLocalScore(TText const & subText, seqan::String<seqan::AminoAcid> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        localScore += seqan::score(seqan::Blosum62(), subText[i], pattern[i]);
    return localScore;
}

template<typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;
    return localScore;
}

template<typename TText, typename TPattern>
seqan::String<int> computeScore(TText const & text, TPattern const & pattern)
{
    seqan::String<int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);
    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);
    return score;
}

template<typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(seqan::String<int> const & text)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}
```
```cpp
int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1 1 0 0 0 0 0 0

    return 0;
}
```

## Tags in SeqAn

Sometimes you will see something like this:

```cpp
globalAlignment(align, seqan::MyersHirschberg())
```

Having a closer look you will notice that there is a default constructor call (`MyersHirschberg()`) within a function call. Using this mechanism one can specify which function to call at compile time. The `MyersHirschberg()` is only a tag to determine which specialisation of the `globalAlignment` function to call.

**If you want more information on tags then read on** otherwise you are now ready to explore SeqAn in more detail and continue with one of the other tutorials.

There is another use case of templates and function specialization.

This might be useful in a `print()` function, for example. In some scenarios, we only want to print the position where the maximal similarity between pattern and text is found. In other cases, we might want to print the similarities of all positions. In SeqAn, we use **tag-based dispatching** to realize this. Here, the type of the tag holds the specialization information.

**Tip:** Tag-Based Dispatching

You will often see tags in SeqAn code, e.g. `Standard()`. These are parameters to functions that are passed as const-references. They are not passed for their values but for their type only. This way, we can select different specializations at **compile time** in a way that plays nicely together with metafunctions, template specializations, and an advanced technique called [[Tutorial/BasicTechniques|metaprogramming]].

Consider the following example:

```cpp
struct TagA {}
struct TagB {}

void myFunction(TagA const &); // (1)
void myFunction(TagB const &); // (2)

int main()
{
    myFunction(TagA()); // (3)
    myFunction(TagB()); // (4)
    return 0;
}
```
The function call in line (3) will call myFunction() in the variant in line (1). The function call in line (4) will call myFunction() in the variant in line (2).

The code for the two different print() functions mentioned above could look like this:

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/score.h>

template <typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

struct MaxOnly {};

template <typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    seqan::String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < seqan::length(score); ++i)
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
        }
        else if (score[i] == maxScore)
        {
            appendValue(output, i);
        }
    }
    for (unsigned i = 0; i < seqan::length(output); ++i)
        std::cout << output[i] << " ";
    std::cout << std::endl;
}
```

If we call print() with something different than MaxOnly then we print all the positions with their similarity, because the generic template function accepts anything as the template argument. On the other hand, if we call print with MaxOnly only the positions with the maximum similarity as well as the maximal similarity will be shown.

**Assignment 6**

**Type** Review

**Objective** Provide a print function that prints pairs of positions and their score if the score is greater than 0.

**Hints** SeqAn provides a data type Pair.

**Solution**
// Provide a print function that prints pairs of positions and their score if the score is greater than 0.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

template <typename TText>
int computeLocalScore(TText const & subText, seqan::String<seqan::AminoAcid> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        localScore += seqan::score(seqan::Blosum62(), subText[i], pattern[i]);

    return localScore;
}

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

template <typename TText, typename TPattern>
seqan::String<seqan::int> computeScore(TText const & text, TPattern const & pattern)
{
    seqan::String<seqan::int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);

    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);

    return score;
}

template <typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(seqan::String<seqan::int> const & text)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

template <typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    print(text);
}

2.1. Tutorial
struct MaxOnly {};

template <typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    seqan::String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < seqan::length(score); ++i)
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
        }
        else if (score[i] == maxScore)
            appendValue(output, i);
    }
    print(output);
}

struct GreaterZero {};

template <typename TText>
void print(TText const & score, GreaterZero const & /*tag*/)
{
    seqan::String<seqan::Pair<int>> output;
    for (unsigned i = 1; i < seqan::length(score); ++i)
    {
        if (score[i] > 0)
            appendValue(output, seqan::Pair<int>{i, score[i]});
    }
    for (unsigned i = 0; i < seqan::length(output); ++i)
    {
        std::cout << "(" << output[i].i1 << "; " << output[i].i2 << ") ";
    }
    std::cout << std::endl;
}

int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);
    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1 1 0 0 0 0 0 0 0
    print(score, MaxOnly());
    // > 19
    print(score, GreaterZero());
    // > (2; 1) (5; 1) (12; 1) (17; 1) (19; 8) (21; 1) (26; 2) (28; 1) (31; 1) (33; 3) (35; 1)

    // And now for a protein pattern
    seqan::String<seqan::AminoAcid> protein = "tutorial";
    seqan::String<int> proteinScore = computeScore(text, protein);
// > This is an awesome tutorial to get to know SeqAn!
print(protein);
// > TXTXRIAL
print(proteinScore);
// > 6 -9 -3 -6 -6 0 -9 -8 -7 -3 -9 -5 -8 -4 -5 -6 -6 1 -6 25 -7 2 -6 -6 -9 -6 -5 -7 1 -7 -5 -4 -6 2 -6 -3 -8 -9 -10 -4 -6 0 0 0 0 0 0 0
print(proteinScore, MaxOnly());
// > 19
print(proteinScore, GreaterZero());
// > (17; 1) (19; 25) (21; 2) (28; 1) (33; 2)
return 0;
}

Obviously this is only a toy example in which we could have named the two `print()` functions differently. However, often this is not the case when the programs become more complex. Because SeqAn is very generic we do not know the datatypes of template functions in advance. This would pose a problem because the function call of function `b()` in function `a()` may depend on the data types of the template arguments of function `a()`.

**The Final Result**

Don’t worry if you have not fully understood the last section. If you have – perfect. In any case the take home message is that you use data types for class specializations and if you see a line of code in which the default constructor is written in a function call this typical means that the data type is important to distinct between different function implementations.

Now you are ready to explore more of the SeqAn library. There are several tutorials which will teach you how to use the different SeqAn data structures and algorithms. Below you find the complete code for our example with the corresponding output.

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

template <typename TText>
int computeLocalScore(TText const & subText, seqan::String<seqan::AminoAcid> const & pattern) {
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        localScore += seqan::score(seqan::Blosum62(), subText[i], pattern[i]);
    return localScore;
}

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern) {
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;
    return localScore;
}

template <typename TText, typename TPattern>
```
seqan::String<int> computeScore(TText const & text, TPattern const & pattern)
{
    seqan::String<int> score;
    seqan::resize(score, seqan::length(text) - seqan::length(pattern) + 1, 0);

    for (unsigned i = 0; i < seqan::length(text) - seqan::length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + seqan::length(pattern)), pattern);

    return score;
}

template<typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(seqan::String<int> const & text)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

template<typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    print(text);
}

struct MaxOnly {};

template<typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    seqan::String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < seqan::length(score); ++i)
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
        }
        else if (score[i] == maxScore)
            appendValue(output, i);
    }

    print(output);
}

struct GreaterZero {};

template<typename TText>
void print(TText const & score, GreaterZero const & /*tag*/)
{
```cpp
seqan::String<seqan::Pair<int>> output;
for (unsigned i = 1; i < seqan::length(score); ++i)
    if (score[i] > 0)
        appendValue(output, seqan::Pair<int>(i, score[i]));

for (unsigned i = 0; i < seqan::length(output); ++i)
    std::cout << "(" << output[i].i1 << "; " << output[i].i2 << ") "
    std::cout << std::endl;
}

int main()
{
    seqan::String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    seqan::String<char> pattern = "tutorial";
    seqan::String<int> score = computeScore(text, pattern);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 0 2 0 1 0 0 1 0 3 0 1 1 0 0 0 0 0 0 0 0
    print(score, MaxOnly());
    // > 19
    print(score, GreaterZero());
    // > (2; 1) (5; 1) (12; 1) (17; 1) (19; 8) (21; 1) (26; 2) (28; 1) (31; 1) (33; 3) '35; 1) (36; 1

    // And now for a protein pattern
    seqan::String<seqan::AminoAcid> protein = "tutorial";
    seqan::String<int> proteinScore = computeScore(text, protein);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(protein);
    // > TXTXRIAL
    print(proteinScore);
    // > 6 -9 -3 -6 -6 0 -9 -8 -7 -3 -9 -5 -8 -4 -5 -6 -6 1 -6 25 -7 2 -6 -6 -9 -6 -5 -7 -7 -5 -4 -
    print(proteinScore, MaxOnly());
    // > 19
    print(proteinScore, GreaterZero());
    // > (17; 1) (19; 25) (21; 2) (28; 1) (33; 2)

    return 0;
}
```

**2.1. Tutorial**
2.1.3 Background and Motivation

**Learning Objective** You will learn about the design goals and fundamental ideas used in the SeqAn library. Also, you will see how the SeqAn library obtains genericity while still retaining high performance.

**Difficulty** Very basic

**Duration** 1h

**Prerequisites** Basic C or C++ knowledge

This tutorial is meant to be the first chapter you read in the SeqAn Tutorial. It will give you an overview about the design goals, design decisions, and explain the motivation for these decisions. The next chapter *First Examples* will flesh out the most important points of this chapter with code examples of everyday SeqAn use.

**Library Design Aims**

The following lists some library design aims of the SeqAn library. Note that they are contradicting. The focus is on efficiency but small trade-offs are allowed to improve consistency and ease of use.

1. **Efficiency**. The focus of SeqAn is to provide a library of efficient and reusable algorithmic components for biological sequence analysis. Algorithms should have good practical implementations with low overhead, even at the cost of being harder to use.

2. **Consistency**. Be consistent wherever possible, even at slight costs of efficiency.

3. **Ease of use**. The library should be easily usable wherever possible, even at slight costs of efficiency.

4. **Reuseability and Generosity**. The algorithms in SeqAn should be reusable and generic, even at small costs of efficiency.

**Modern C++ (C++98)**

C++ is sometimes described as a language that most people know only 20% of but everyone knows a different 20%. This section gives an overview over some C++98 idioms we use. This might be no news if you are a seasoned C++ programmer who is apt at using the STL and Boost libraries. However, programmers coming from C and Java might find them interesting.

**References** References are alternatives to pointers in C++ to construct value aliases. Also see Wikipedia on C++ references.
Templates C++ allows you to perform generic programming using templates. While similar to generics in Java (C++ templates are more than a decade older), C++ templates are designed to write zero-overhead abstractions that can be written to be as efficient as hand-written code while retaining a high level of abstraction. See cplusplus.com on C++ templates. Note that there is no way to restrict the type that can be used in templates, there is no mechanism such as Java’s `extends` T in C++. Using an incompatible type leads to compiler errors because some operator or function could not be found.

Memory Management / No Pointers Instead of using raw pointers and memory, memory management should be done using containers. The STL provides containers such as `std::vector` and SeqAn offers `String`.

Memory Management in SeqAn

C++ allows to allocate complex objects on the stack (in contrast to Java where objects are always constructed on the heap). The objects are constructed when the code execution enters the scope/block they are defined in and freed when the block is left. Allocation of resources (e.g. memory) happens on construction and deallocation happens when the current block is left. This is best explained in an example.

`seqan::String<char>` is a class (actually an instantiation of the class template `String`) that allows to store strings of char values, similar to `std::vector<char>` or `std::string`.

When the variable `programName` is allocated, the constructor of the `String<char>` class is called. It allocates sufficient memory to store the value of `argv[0]` and then copies over the values from this string. The variable exists until the current block is left. Since it is defined in the `main()` function, this can only happen in the last line of `main()` at the return 0. When the variable goes out of scope, its value is deconstructed and all allocated memory is freed.

If an argument was given to the program, the block in the `if` clause is entered. When this happens, the variable `firstArg` is constructed, memory is allocated and the value of `argv[1]` is copied into the buffer. When the block is left, the variable is deconstructed and all memory is deallocated.

Note that all memory is released when the `main()` function is left, regardless whether it is left in the `return 0` or the `return 1`. Corresponding code in C would be (arguably) more messy, either requiring `goto` or multiple `free()` calls, one before either `return`.

Motivation for Template Programming

In this section, we will give a short rationale why C++ with heavy use of template programming was used for SeqAn.

Any sequence analysis will have sequence data structures and algorithms on sequences at its heart. Even when only considering DNA and amino acid alphabets, there are various variants for alphabets that one has to consider. Otherwise, important applications in bioinformatics cannot be covered:

- 4-character DNA,
- 5-character DNA with N,
- 15-character IUPAC, and
- 23-character amino acids.

A simple implementation could simply store such strings as ASCII characters. However, there are some implementation tricks that can lead to great reduction of memory usage (e.g. encoding eight 4-character DNA characters in one byte) or running time (fast lookup tables for characters or q-grams) for small alphabets. Thus, simply using a `std::string` would come at high costs to efficiency.

Given that in the last 10-15 years, Java and C# have gained popularity, one could think about an object oriented solution: strings could simply be arrays of `Character` objects. Using polymorphism (e.g. overwriting of functions in subclasses), one could then write generic and reusable algorithms. For example, the Java 2 platform defines the sort function for all objects implementing a `Comparable` interface. Note that such an implementation would have to rely...
on virtual functions of some sort. However, as we will see in the section OOP vs. Template Subclassing, this comes at a high performance cost, being in conflict with efficiency. For a sequence library, we could implement functions that map values from an alphabet to an ordinal value between 0 and $S - 1$ where $S$ is the number of elements in the alphabet.

Generic programming offers one way out: C++ templates allow to define template classes, e.g. the STL’s `std::vector<T>` or SeqAn’s `String`. Here, instead of creating a string class around an array of char values (or objects), we can leave the type of the array’s elements open. We can then introduce different types, e.g. `Dna` or `Dna5` for 4- and 5-character DNA alphabets.

Algorithms can be implemented using templated functions and the template types are fixed at compile time. Thus, the compiler does not have to use virtual function tables and other “crutches”, less indirection is involved, and more code can be inlined and aggressively optimized. When written appropriately, such algorithms can also work on different string implementations! Also, when defining our own alphabet types, we can directly influence how their abstractions (and APIs) work.

Thus, C++ allows us to implement (1) a generic and reusable library with (2) high level abstractions (and thus ease of use) that still allows the compiler to employ aggressive optimization and thus achieves (3) efficiency. With the words of the C++ inventor Bjarne Stroustrup:

> A high level of abstraction is good, not just in C++, but in general. We want to deal with problems at the level we are thinking about those problems. When we do that, we have no gap between the way we understand problems and the way we implement their solutions. We can understand the next guy’s code. We don’t have to be the compiler.

### OOP vs. Generic Programming

In SeqAn, we use a technique called template subclassing which is based on generic programming. This technique provides polymorphism into C++ programs at compile time using templates. Such static polymorphism is different from runtime polymorphism which is supported in C++ using subclassing and virtual functions. It comes at the cost of some additional typing but has the advantage that the compiler can inline all function calls and thus achieve better performance. An example will be given in the section “From OOP to SeqAn” in the First Steps Tutorial.

The important point is that in contrast to runtime polymorphism such static polymorphism allows the compiler to inline functions.

### Performance Example

The following small program shows impressive performance gains when using inlined functions instead of virtual functions. We will sort random quadruples of `int` values using the STL function `std::sort`.

In the program, we will sort `std::vector` objects of the two types `Item1` and `Item2`. The only difference is that the comparison operator `operator<()` for `Item1` can be inlined while `operator<()` for `Item2` is virtual and thus cannot be inlined.

The relevant lines in the code below are highlighted.

```c++
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>

#include <omp.h> // For omp_get_wtime() only.

const int ITERATIONS = 10;
const int N = 100*1000*1000;
const int SEED = 42;
```
struct Item1
{
    int i1, i2, i3, i4, i5;
    Item1() : i1(0), i2(0), i3(0), i4(0), i5(0)
    {}
    Item1(int i) : i1(i1), i2(i), i3(i), i4(i), i5(i)
    {}

    bool operator<(Item1 const & other) const
    {
        return i1 < other.i1;
    }
};

struct Item2
{
    int i1, i2, i3, i4, i5;
    Item2() : i1(0), i2(0), i3(0), i4(0), i5(0)
    {}
    Item2(int i) : i1(i1), i2(i), i3(i), i4(i), i5(i)
    {}

    virtual bool operator<(Item2 const & other) const
    {
        return i1 < other.i1;
    }
};

int main()
{
    double start = 0;
    double timeAvg = 0;
    double timeDev = 0;
    std::vector<double> times;

    std::cout << "Parameters\n";
    std::cout << " # iterations: " << ITERATIONS << "\n";
    std::cout << " # items : " << N << "\n";
    std::cout << " seed : " << SEED << "\n";

    // Generate random input.
    std::cout << "Generating input.\n";
    start = omp_get_wtime();
    std::tr1::mt19937 eng(SEED);
    std::tr1::uniform_int<int> unif;
    std::vector<int> nums;
    nums.reserve(N);
    for (int i = 0; i < N; ++i)
        nums.push_back(unif(eng));
    std::cout << " time " << omp_get_wtime() - start << " s\n";

    // Sort with inlining.
    times.clear();
timeAvg = 0;
timeDev = 0;
std::cout << "std::sort with inlining\n";
for (int i = 0; i < ITERATIONS + 1; ++i)
{
    std::vector<Item1> items(nums.begin(), nums.end());
    start = omp_get_wtime();
    std::sort(items.begin(), items.end());
    if (i > 0)
        times.push_back(omp_get_wtime() - start);
}
for (unsigned i = 0; i < times.size(); ++i)
    timeAvg += times[i];
timeAvg /= times.size();
for (unsigned i = 0; i < times.size(); ++i)
    timeDev += (timeAvg - times[i]) * (timeAvg - times[i]);
timeDev /= times.size();
timeDev = sqrt(timeDev);
std::cout << " time avg " << timeAvg << " s dev " << timeDev << "\n";

// Sorting with virtual operator<().
times.clear();
timeAvg = 0;
timeDev = 0;
std::cout << "std::sort without inlining\n";
for (int i = 0; i < ITERATIONS + 1; ++i)
{
    std::vector<Item2> items(nums.begin(), nums.end());
    start = omp_get_wtime();
    std::sort(items.begin(), items.end());
    if (i > 0)
        times.push_back(omp_get_wtime() - start);
}
for (unsigned i = 0; i < times.size(); ++i)
    timeAvg += times[i];
timeAvg /= times.size();
for (unsigned i = 0; i < times.size(); ++i)
    timeDev += (timeAvg - times[i]) * (timeAvg - times[i]);
timeDev /= times.size();
timeDev = sqrt(timeDev);
std::cout << " time avg " << timeAvg << " s dev " << timeDev << "\n";

return 0;

The resulting differences in running times on a Xeon X5550 @2.67 Ghz machine, compiled with g++ 4.4.5 are as follows. Thus, we have an improved performance with a **factor 2** of inlined functions instead of virtual function calls!

Parameters
---

- **# iterations**: 10
- **# items**: 100000000
- **seed**: 42

Generating input.

time 0.836878 s

std::sort with inlining

time avg 5.43477 s dev 0.0817846
2.1.4 Sequences

**Learning Objective** You will learn about the SeqAn sequence concept and its main class `String` as well as the class `Segment`. After completing this tutorial, you will be able to use important functionalities of sequences in SeqAn and you will be ready to continue with the more specific tutorials, e.g. Iterators, Alignment Representation, or Pairwise Sequence Alignment.

**Difficulty** Very basic

**Duration** 45 min

**Prerequisites** Basic C or C++ knowledge, the A First Example tutorial helps.

Sequences are the core concept of SeqAn. A sequence is a container that stores an ordered list of values. In SeqAn, there are three kinds of sequences: Strings, Sequence Adaptions and Segments.

The `String` class is one of the most fundamental classes in SeqAn. It is designed as a generic data structure that can be instantiated for all kinds of values, both simple (e.g. `char`, `Dna`, `AminoAcid`) and non-simple value types (e.g. `Tuple`, `String`). With sequence adaptions, SeqAn offers an interface for accessing data types that are not part of SeqAn, namely standard library strings and c-style char arrays. Thus those built-in types can be handled in a similar way as SeqAn strings, for example with the `length` function. Segments are contiguous subsequences that represent parts of other sequences.

This tutorial will deal with the SeqAn sequence classes `String` and `Segment`.

**Strings**

In this section, we will have a detailed look at the SeqAn class `String`. You will learn how to build and expand strings as well as how to compare and convert them.
Building Strings

Let’s first have a look at an example on how to define a String. The type of the contained value is specified by the first
template argument, e.g. char or int.

```
String<char> myText; // A string of characters.
String<int> myNumbers; // A string of integers.
```

Any type that provides a default constructor, a copy constructor and an assignment operator can be used as the alphabet / contained type of a String. This includes the C++ POD types, e.g. char, int, double etc., but you can use more complex types, e.g. Strings, too.

```
String<String<char>> myStringList; // A string of character strings.
```

**Hint:** Nested Sequences (aka “Strings of Strings”)

A set of sequences can either be stored in a sequence of sequences, for example in a `String<String<char>>`, or in `StringSet`. See the tutorial `String Sets` for more information about the class `StringSet`.

```
String<Dna> myGenome; // A string of nucleotides.
String<AminAcid> myProtein; // A string of amino acids.
```

For commonly used string parameterizations, SeqAn has a range of shortcuts implemented, e.g. `DnaString`, `RnaString` and `Peptide`.

```
// Instead of String<Dna> dnaSeq we can also write:
DnaString dnaSeq = "TATA";
```

The user can specify the kind of string that should be used in an optional second template argument of String. This is also known as selecting the specialization of a class in SeqAn. The default string implementation is Alloc String, which the best choice for most cases.

```
String<Dna> myGenome; // A default string of nucleotides.
String<Dna, Alloc<> > myGenome; // The same as above.
```

For some scenarios though, there might be other types more suitable. One such example is when processing extremely large strings that are much larger than the available main memory. In this case, using `External Strings` is a good choice.

```
// Most of the string is stored on the disk.
String<Dna, External<> > myLargeGenome;
```

More details about the different specializations you can find in the tutorial `Sequences In-Depth`.

**Tip:** String Simplify Memory Management

One advantage of using Strings is that the user does not need to reserve memory manually with `new` and does not need `delete` to free memory. Instead, those operations are automatically handled by the `String` class.

```
String<Dna> myGenome = "TATACGCG";
```

**Functionality**

SeqAn also provides the common C++ operators for strings. You can use them like STL strings, for example:
String<Dna> dnaSeq = "TATA";
dnaSeq += "CGCG";
std::cout << dnaSeq << std::endl;
TATACGCG

Each sequence object has a capacity, i.e. the maximum length of a sequence that can be stored in this object. While some sequence types have a fixed capacity, the capacity of other sequence classes like Alloc String or std::basic_string can be changed at runtime. The capacity can be set explicitly by functions such as reserve or resize. It can also be set implicitly by functions like append or replace, if the operation's result exceeds the length of the target string.

In the following example, a String of Dna5String, we first set the new length of the container with resize to two elements. After assigning two elements we append one more element with appendValue. In the last step the capacity is implicitly changed.

String<Dna5String> readList;
resize(readList, 2);
readList[0] = "GGTTTCGACG";
readList[1] = "AAGATGTCGC";
appendValue(readList, "TATGCATGAT");

Using the function length, we can now get the length of our strings, e.g.:

```cpp
std::cout << length(readList) << std::endl;
std::cout << length(readList[0]) << std::endl;
```

```
3
10
```

To empty a String, the function clear resets the object.

clear(readList);

SeqAn offers a range of other functions for the work with the String class, e.g. assign, assignValue, value, getValue, empty, etc. The full list of functions you can find in the documentation String.

### Assignment 1

**Type** Review

**Objective** In the following assignment, you will write a small function that builds the reverse complement of a given string. Copy the code below and add the following functionalities:

1. Use the resize function to resize the revComplGenome variable.

2. Using the getRevCompl function, get the reverse complement for every nucleotide genome and store it in reverse order revComplGenome.

3. Print out the original genome and the reverse complement.

```cpp
#include <seqan/sequence.h>
#include <iostream>

using namespace seqan;
```
Dna getRevCompl(Dna const & nucleotide) {
    if (nucleotide == (Dna)'A')
        return (Dna)'T';
    if (nucleotide == (Dna)'T')
        return (Dna)'A';
    if (nucleotide == (Dna)'C')
        return (Dna)'G';
    return (Dna)'C';
}

int main() {
    DnaString genome = "TATATACGCGCGAGTCGT";
    DnaString revComplGenome;
    // Your code snippet
    // And to check if your output is correct,
    // use the given SeqAn function reverseComplement(),
    // which modifies the sequence in-place
    reverseComplement(genome);
    std::cout << genome << std::endl;
    return 0;
}

Hints Remember that the last element in genome is stored at position length(genome) − 1.

Solution Click more... to see the solution.

#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace seqan;

Dna getRevCompl(Dna const & nucleotide) {
    if (nucleotide == (Dna)'A')
        return (Dna)'T';
    if (nucleotide == (Dna)'T')
        return (Dna)'A';
    if (nucleotide == (Dna)'C')
        return (Dna)'G';
    return (Dna)'C';
}

int main() {
    DnaString genome = "TATATACGCGCGAGTCGT";
    DnaString revComplGenome;
    resize(revComplGenome, length(genome));
    for (unsigned i = 0; i < length(genome); ++i) {
        revComplGenome[length(genome) - 1 - i] = getRevCompl(genome[i]);
    }
}
Workshop Assignment 1

Type Review

Objective In this assignment, you will do some simple string building tasks, and write a simple alignment of the given reads and chromosomes. Use the given code template to solve these subtasks:

1. Assume we have mapped the reads to the positions 7, 100, 172, and 272 in ‘chr1’. Store these positions in another string ‘alignPosList’.

2. Build another String bsChr1 as a copy of chr1, and exchange every ‘C’ with a ‘T’, as in a bisulfite treated genome.

3. Print alignments of the reads and chr1 (or bschr1) using the function printAlign and the string alignPosList.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 = "TATAATTTGCTATCGGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";
    DnaString chr2 = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACGTGTGTCACGTGCAGCGGATTATGCGCTCTGCGATATATCGCGCTAGATGTCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";

    // Build List containing all reads
typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";
    readList[1] = "TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";
    readList[2] = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACGTGTGTCACGTGCAGCGGATTATGCGCTCTGCGATATATCGCGCTAGATGTCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";
    readList[3] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTCGTCGTCGACTGCTGCTGACA";

    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACGTGTGTCACGTGCAGCGGATTATGCGCTCTGCGATATATCGCGCTAGATGTCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATAATTAGCGGTCTAGCATTTAG";"
"CACTGCTGACGTCGTTACCATGCTCGTGCAGCTACGCTGCTGCTGAC"
"ACATGCTGCTG";
append(chr1, chr2);

// Print readlist
std::cout << " \n Read list: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
    std::cout << readList[i] << std::endl;

// Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions
// Store the start position in a String alignPosList: 7, 100, 172, 272
// Optional
// Bisulfite conversion
// Assume chr1 is being bisulfate treated: Copy chr1 to a new genome bsChr1 and exchange every 'C'
DnaString bsChr1;

// Print alignments of the reads with chr1 (or bsChr1) sequence using the function printAlign
// and the positions in alignPosList.
// To do that, you have to create a copy of the fragment in chr1 (bsChr1) that is aligned to the read.
// Call of our function to print the simple alignment
for(unsigned i = 0; i < length(readList); ++i)
{
    // Temporary copy of begin position (beginPosition) from alignPosList of a given alignment between the read and the genome
    // Genome fragment
    DnaString genomeFragment;
    // We have to create a copy of the corresponding fragment of the genome, where the read aligns
    std::cout << " \n Print alignment: " << std::endl;
    printAlign(genomeFragment, readList[i]);
}
return 1;

Hints  You have to create a copy of the fragment in chr1 (bsChr1) that is aligned to the read.
Solution  Click more... to see the solution.

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template<typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 = "TATAATATATCTCGCGATATCGCTAGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Build List containing all reads
typedef String<DnaString> TDnaList;
TDnaList readList;
resize(readList, 4);
readList[0] = "TTGCTATCGGATATCGCTATACCTACCGATTATATCGCGCT";
readList[1] = "TCGATTACGTGAATCGATCTAATCGGCCTCCGTTATAGATCTACGCTACTGCTGACATT";
readList[2] = "AGCTGCGGTTCACTGCGGTTTGCGGTAGCTTGGCTGAGACTGGATGCGCGATGCGAGCAGACACATGCACA";
readList[3] = "CGTGCCACTGCTGACTGCTGTTGCTCACTGCTGCGGCTGCTGCTGACTGCTGCTGACACGTCTCTGTGTTCCGACGTGTCACGTGCACTGCTGACGTCGTGTTGTCACATCGTCGTGCTGCGTACTGCTGCTGACA";
// Append a second chromosome sequence fragment to chr1
DnaString chr2 = "AGCCTGCGTACGCTGTCGATGACTTGTCAAGCCGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACGTGTCACGTGCACTGCTGACGTCGTGTTGTCACATCGTCGTGCTGCGTACTGCTGCTGACA";
append(chr1, chr2);
// Print readlist
std::cout << "\nRead list: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
    std::cout << readList[i] << std::endl;
// Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions (no gaps).
// Store the start position in a String alignPosList: 7, 100, 172, 272
String<unsigned> alignPosList;
resize(alignPosList, 4);
alignPosList[0] = 7;
alignPosList[1] = 100;
alignPosList[2] = 172;
alignPosList[3] = 272;
// Optional
// Bisulfite conversion
// Assume chr1 is being bisulfate treated: Copy chr1 to a new genome bsChr1 and exchange every 'C' with a 'T'
DnaString bsChr1;
assign(bsChr1, chr1);
for(unsigned i = 0; i < length(bsChr1); ++i)
    if(bsChr1[i] == 'C')
        bsChr1[i] = 'T';
// Print alignments of the reads with chr1 (or bsChr1) sequence using the function printAlign and the positions in alignPosList.
// To do that, you have to create a copy of the fragment in chr1 (bsChr1) that is aligned to each read.
std::cout << "\nPrint alignment: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
{
    // Begin position beginPosition of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    // Genome fragment
    DnaString genomeFragment;
    // We have to create a copy of the corresponding fragment of the genome, where the read
    for (unsigned j = 0; j < length(readList[i]); ++j)
        appendValue(genomeFragment, chr1[beginPosition+j]);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, readList[i]);
}
return 1;
}
Each comparison involves a scan of the two sequences for searching the first mismatch between the strings. This could be costly if the two sequences share a long common prefix. Suppose we want to branch in a program depending on whether \( a < b, a == b, \) or \( a > b \).

```cpp
if (a < b) { /* code for case "a < b" */ }
else if (a > b) { /* code for case "a > b" */ }
else { /* code for case "a == b" */ }
```

In this case, although only one scan would be enough to decide what case is to be applied, each operator \( > \) and \( < \) performs a new comparison. SeqAn offers the class `Lexical` to avoid unnecessary sequence scans. Lexicals can store the result of a comparison, for example:

```cpp
// Compare a and b and store the result in comp
Lexical<> comp(a, b);

if (isLess(comp)) { /* code for case "a < b" */ }
else if (isGreater(comp)) { /* code for case "a > b" */ }
else { /* code for case "a == b" */ }
```

### Conversions

A sequence of type A values can be converted into a sequence of type B values, if A can be converted into B. SeqAn offers different conversion alternatives.

**Copy conversion.** The source sequence is copied into the target sequence. This can be done by assignment (`operator=`) or using the function `assign`.

```cpp
String<Dna> source = "acgtgcat";
String<char> target;
assign(target, source);
std::cout << target;
```

**Move conversion.** If the source sequence is not needed any more after the conversion, it is always advisable to use `move` instead of `assign`. The function `move` does not make a copy but can reuse the source sequence storage. In some cases, `move` can also perform an in-place conversion.

```cpp
String<char> source = "acgtgcat";
String<Dna> target;

// The in-place move conversion.
move(target, source);
std::cout << target;
```

### Assignment 2

**Type Review**
Objective In this assignment you will sort nucleotides. Copy the code below. Adjust the code such that all nucleotides, which are lexicographically smaller than a Dna5 ’G’ are stored in a list lesser, while all nucleotides which are greater, should be stored in a list greater. Print out the final lists.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

int main()
{
    String<Dna5> nucleotides = "AGTCGTGNNANCT";
    String<Dna5> selected;
    // Append all elements of nucleotides, apart of Gs, // to the list selected.
    for (unsigned i = 0; i < length(nucleotides); ++i){
        appendValue(selected, nucleotides[i]);
    }
    std::cout << "Selected nucleotides: " << selected << std::endl;
    return 0;
}
```

Solution Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    String<Dna5> nucleotides = "AGTCGTGNNANCT";
    String<Dna5> lesser;
    String<Dna5> greater;
    for (unsigned i = 0; i < length(nucleotides); ++i){
        if (nucleotides[i] < 'G')
            appendValue(lesser, nucleotides[i]);
        else if (nucleotides[i] > 'G')
            appendValue(greater, nucleotides[i]);
    }
    std::cout << "Lesser nucleotides: " << lesser << std::endl;
    std::cout << "Greater nucleotides: " << greater << std::endl;
}
```

Assignment 3

Type Transfer

Objective In this task you will compare whole sequences. Reuse the code from above. Instead of a String<Dna5> we will now deal with a String<Dna5String>. Build a string which contains the Dna5Strings “ATATANGCGT”, “AAGCATGANT” and “TGAAANTGAC”. Now check for all elements of the container, if they are lexicographically smaller or bigger than the given reference sequence “GATGCATGAT” and append them to a appropriate list. Print out the final lists.

Hints Try to avoid unnecessary sequence scans.
Solution  Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    String<Dna5String> nucleotidesList;
    Dna5String str1 = "ATATANGCGT";
    Dna5String str2 = "AAGGATGANT";
    Dna5String str3 = "TGAAANTGAC";
    resize(nucleotidesList, 3);
    nucleotidesList[0] = str1;
    nucleotidesList[1] = str2;
    nucleotidesList[2] = str3;

    String<Dna5String> lesser;
    String<Dna5String> greater;
    Dna5String ref = "GATGCATGAT";

    // For each Dna5String of the String:
    for (unsigned i = 0; i < length(nucleotidesList); ++i){
        // Compare the Dna5String with the given reference string
        // The result of the comparison is stored in comp
        Lexical<> comp(nucleotidesList[i], ref);
        // The function isLess checks only the stored result
        // without comparing the sequences again
        if (isLess(comp))
            appendValue(lesser, nucleotidesList[i]);
        else if (isGreater(comp))
            appendValue(greater, nucleotidesList[i]);
    }

    // Print the results
    std::cout << "Lesser sequences: " << std::endl;
    for (unsigned i = 0; i < length(lesser); ++i){
        std::cout << lesser[i] << ", ";
    }
    std::cout << std::endl;
    std::cout << "Greater sequences: " << std::endl;
    for (unsigned i = 0; i < length(greater); ++i){
        std::cout << greater[i] << ", ";
    }
}
```

Segments

The following section will introduce you into the **Segment** class of SeqAn.

**Segments** are contiguous subsequences that represent parts of other sequences. Therefore, their functionality is similar to the **String** functionality. In SeqAn, there are three kinds of segments: **InfixSegment**, **PrefixSegment**, and **SuffixSegment**. The metafunctions **Infix**, **Prefix**, and **Suffix**, respectively, return the appropriate segment data type for a given sequence type.

For prefixes, we use the function **prefix** to build the prefix. The first parameter is the sequence we build the prefix from, the second the **excluding** end position. For infixes, we have to provide both the including start and the excluding
end position. For suffixes, the second parameter of the function denotes the including starting position of the suffix:

```cpp
String<Dna> dnaSeq = "AGTTGGCATG";
Prefix<String<Dna> >::Type pre = prefix(dnaSeq, 4);
std::cout << "Prefix: " << pre << std::endl;

Infix<String<Dna> >::Type inf = infix(dnaSeq, 4, 7);
std::cout << "Infix: " << inf << std::endl;

Suffix<String<Dna> >::Type suf = suffix(dnaSeq, 4);
std::cout << "Suffix: " << suf << std::endl;
```

Prefix: AGTT
Infix: GGC
Suffix: GGCATG

Segments store a pointer on the underlying sequence object, the host, and an start and/or end position, depending on the type of segment. The segment is not a copy of the sequence segment.

**Warning:** Please note that it is not possible anymore to change the underlying sequence by changing the segment. If you want to change the host sequence, you have to explicitly modify this. If you want to modify only the segment, you have to explicitly make a copy of the string.

---

**Assignment 4**

**Type** Application

**Objective**

In this task you will use a segment to pass over an infix of a given sequence to a function without copying the corresponding fragment. Use the code given below. Let's assume that we have given a genome and a read sequence as well as the begin position of a given alignment. In the main function a fragment of the Dna5String genome is copied and passed together with the Dna5String read to a print function. Adjust the code to use an infix of the genome, instead of copying the corresponding fragment.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
// for two sequences of the same type
template <typename TText>
void printAlign(TText const & genomeFragment, TText const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : ";
    std::cout << genomeFragment << std::endl;
    std::cout << " read : ";
    std::cout << read << std::endl;
}

int main()
{
    // We have given a genome sequence
    Dna5String genome = "ATGGTTTCAACGTAATGCTGAACATGTCGCGT";
```
// A read sequence
Dna5String read = "TGGTNTCA";
// And the begin position of a given alignment between the read and the genome
unsigned beginPosition = 1;

Dna5String genomeFragment;
// We have to create a copy of the corresponding fragment of the genome, where the read aligns to
for (unsigned i = 0; i < length(read); ++i){
    appendValue(genomeFragment, genome[beginPosition+i]);
}
// Call of our function to print the simple alignment
printAlign(genomeFragment, read);

return 0;
}

Solution  Click more... to see the solution.

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
// .. for two sequences of different types
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : ";
    std::cout << genomeFragment << std::endl;
    std::cout << " read : ";
    std::cout << read << std::endl;
}

int main()
{
    // We have given a genome sequence
    Dna5String genome = "ATGGTTTCAACGTAATGCTGAACATGTCGCGT";
    // A read sequence
    Dna5String read = "TGGTNTCA";
    // And the begin position of a given alignment between the read and the genome
    unsigned beginPosition = 1;

    // Create Infix of type Dna5String and get the corresponding infix sequence of genome
    Infix<Dna5String>::Type inf = infix(genome, beginPosition, beginPosition + length(read));
    // Call of our function to print the simple alignment
    printAlign(inf, read);
    return 0;
}

Workshop Assignment 2

Type  Review

Objective  Take the solution from the workshop assignment above and change it to use Segments for building the
genome fragment.

Hints Note that because `printAlign` uses templates, you don’t have to change the function even though the type of `genomeFragment` is different.

Solution Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **) {
    // Build reads and genomes
    DnaString chr1 = "TATAATATTGCTATCCGATATCGCTAGCAGATTATCGCTAGCTAGCTACGGATTATGCCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Build List containing all reads
    typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGCGATATCGCTAGCTAGCAGATTATCGCTAGCTAGCTACGGATTATGCCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions (no gaps).
    // Store the start position in a String alignPosList: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;
    // Bisulfite conversion
    // Assume chr1 is being bisulfite treated: Copy chr1 to a new genome bsChr1 and exchange every 'C' with a 'T'
    DnaString bsChr1;
    assign(bsChr1, chr1);
    for(unsigned i = 0; i < length(bsChr1); ++i)
    if(bsChr1[i] == 'C')
        bsChr1[i] = 'T';
    // Print alignments using Segment: Do the same as above, but instead of using a for loop to build the fragment,
    // use the Segment class to build an infix of bsChr1.
    // Note: Because printAlign uses templates, we don’t have to change the function even though
    // genomeFragment is different.
```

2.1. Tutorial 67
std::cout << "
Print alignment using Segment: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
{
    // Begin and end position of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(readList[i]);
    // Build infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, readList[i]);
}
return 1;
}

2.1.5 Alphabets

Learning Objective You will learn the details about the alphabets in SeqAn.

Difficulty Basic

Duration 15 min

Prerequisites A First Example

This tutorial will describe the different alphabets used in SeqAn, or in other words, you will learn about the contained types of a SeqAn String. To continue with the other tutorials, it would be enough to know, that in SeqAn several standard alphabets are already predefined, e.g. Dna, Dna5, Rna, Rna5, Iupac, AminoAcid.

Types

Any type that provides a default constructor, a copy constructor and an assignment operator can be used as the alphabet / contained type of a String (see also the tutorial Sequences). This includes the C++ POD types, e.g. char, int, double etc. In addition you can use more complex types like String as the contained type of strings, e.g. String<String<char> >.

SeqAn also provides the following types that are useful in bioinformatics. Each of them is a specialization of the class SimpleType.
### Specialization

<table>
<thead>
<tr>
<th>Specialization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AminoAcid</td>
<td>Amino Acid Alphabet</td>
</tr>
<tr>
<td>Dna</td>
<td>DNA alphabet</td>
</tr>
<tr>
<td>Dna5</td>
<td>N alphabet including N character</td>
</tr>
<tr>
<td>DnaQ</td>
<td>N alphabet plus phred quality</td>
</tr>
<tr>
<td>DnaSQ</td>
<td>N alphabet plus phred quality including N character</td>
</tr>
<tr>
<td>Finite</td>
<td>Finite alphabet of fixed size.</td>
</tr>
<tr>
<td>Iupac</td>
<td>N Iupac code.</td>
</tr>
<tr>
<td>Rna</td>
<td>N alphabet</td>
</tr>
<tr>
<td>Rna5</td>
<td>N alphabet including N character</td>
</tr>
</tbody>
</table>

### Functionality

In SeqAn, alphabets are value types that can take a limited number of values and which hence can be mapped to a range of natural numbers. We can retrieve the number of different values of an alphabet, the alphabet size, by the metafunction `ValueSize`.

```cpp
typedef Dna TAlphabet;

unsigned alphSize = ValueSize<TAlphabet>::VALUE;
std::cout << "Alphabet size of Dna: " << alphSize << '\n';

Alphabet size of Dna: 4
```

Another useful metafunction called `BitsPerValue` can be used to determine the number of bits needed to store a value of a given alphabet.

```cpp
unsigned bits = BitsPerValue<TAlphabet>::VALUE;
std::cout << "Number of bits needed to store a value of type Dna: " << bits << '\n';

Number of bits needed to store a value of type Dna: 2
```

The order of a character in the alphabet (i.e. its corresponding natural number) can be retrieved by calling the function `ordValue`. See each specialization’s documentation for the ordering of the alphabet’s values.

```cpp
Dna a = 'A';
Dna c = 'C';
Dna g = 'G';
Dna t = 'T';

std::cout << "A: " << (unsigned)ordValue(a) << '\n';
std::cout << "C: " << (unsigned)ordValue(c) << '\n';
std::cout << "G: " << (unsigned)ordValue(g) << '\n';
std::cout << "T: " << (unsigned)ordValue(t) << '\n';
```

A: 0
C: 1
G: 2
T: 3

**Tip:** The return value of the `ordValue` function is determined by the metafunction `ValueSize`. `ValueSize` returns the type which uses the least amount of memory while being able to represent all possible values. E.g. `ValueSize` of `Dna` returns an `_uint8` which is able to represent 256 different characters. However, note that `std::cout` has no visible symbol for printing all values on the screen, hence a cast to `unsigned` might be necessary.
Assignment 1

Type  Application

Objective  In this task you will learn how to access all the letters of an alphabet. Use the piece of code from below and adjust the function `showAllLettersOfMyAlphabet()` to go through all the characters of the current alphabet and print them.

```cpp
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>

using namespace seqan;

// We want to define a function, which takes
// the alphabet type as an argument

void showAllLettersOfMyAlphabet(TAlphabet const &)
{
    // ...
}

int main()
{
    showAllLettersOfMyAlphabet(AminoAcid());
    showAllLettersOfMyAlphabet(Dna());
    showAllLettersOfMyAlphabet(Dna5());
    return 0;
}
```

Hints  You will need the Metafunction `ValueSize`.

Solution  Click more... to see the solution.

```cpp
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>

using namespace seqan;

// We define a function which takes
// the alphabet type as an argument

void showAllLettersOfMyAlphabet(TAlphabet const &)
{
    typedef typename Size<TAlphabet>::Type TSize;
    // We need to determine the alphabet size
    // using the metafunction ValueSize
    TSize alphSize = ValueSize<TAlphabet>::VALUE;
    // We iterate over all characters of the alphabet
    // and output them
    for (TSize i = 0; i < alphSize; ++i)
        std::cout << i << ',' << TAlphabet(i) << " ";
    std::cout << std::endl;
}

int main()
{
```
2.1.6 String Sets

Learning Objective You will learn the advantages of StringSets and how to work with them.

Difficulty Basic

Duration 15 min

Prerequisites Sequences

A set of sequences can either be stored in a sequence of sequences, for example in a \texttt{String\langle String\langle char\rangle \rangle}, or in a \texttt{StringSet}. This tutorial will introduce you to the SeqAn class \texttt{StringSet}, its background and how to use it.

Background

One advantage of using \texttt{StringSet} is that it supports the function \texttt{concat} that returns a \texttt{concatenator} of all sequences in the string set. A \texttt{concatenator} is an object that represents the concatenation of a set of strings. This way, it is possible to build up index data structures for multiple sequences by using the same construction methods as for single sequences.

There are two kinds of \texttt{StringSet} specializations in SeqAn: \texttt{Owner StringSet}, the default specialisation, and \texttt{Dependent StringSet}; see the list below for details. \texttt{Owner StringSets} actually store the sequences, whereas \texttt{Dependent StringSets} just refer to sequences that are stored outside of the string set.

\begin{verbatim}
StringSet\langle DnaString \rangle ownerSet;
StringSet\langle DnaString, Owner\rangle ownerSet2; // same as above
StringSet\langle DnaString, Dependent\rangle dependentSet;
\end{verbatim}

The specialization \texttt{ConcatDirect StringSet} already stores the sequences in a concatenation. The concatenators for all other specializations of \texttt{StringSet} are \texttt{virtual} sequences, that means their interface \texttt{simulates} a concatenation of the sequences, but they do not literally concatenate the sequences into a single sequence. Hence, the sequences do not need to be copied when a concatenator is created.
One string can be an element of several Dependent StringSets. Typical tasks are, e.g., to find a specific string in a string set, or to test whether the strings in two string sets are the same. Therefore a mechanism to identify the strings in the string set is needed, and, for performance reasons, this identification should not involve string comparisons. SeqAn solves this problem by introducing ids, which are by default unsigned int values.

The following list lists the different StringSet specializations:

**Specialization Owner<ConcatDirect>**  The sequences are stored as parts of a long string. Since the sequences are already concatenated, concat just needs to return this string. The string set also stores lengths and starting positions of the strings. Inserting new strings into the set or removing strings from the set is more expensive than for the default OwnerStringSet specialization, since this involves moving all subsequent sequences in memory.

**Specialization Dependent<Tight>**  This specialization stores sequence pointers consecutively in an array. Another array stores an id value for each sequence. That means that accessing given an id needs a search through the id array.

**Specialization Dependent<Generous>**  The sequence pointers are stored in an array at the position of their ids. If a specific id is not present, the array stores a zero at this position. The advantage of this specialization is that accessing the sequence given its id is very fast. On the other hand, accessing a sequence given its position \( i \) can be expensive, since this means we have to find the \( i \)-th non-zero value in the array of sequence pointers. The space requirements of a string set object depends on the largest id rather than the number of sequences stored in the set. This could be inefficient for string sets that store a small subset out of a large number of sequences.

### Building String Sets

Use the function `appendValue` to append strings to string sets.

```cpp
StringSet<DnaString> stringSet;
DnaString str0 = "TATA";
DnaString str1 = "CGCG";
appeadValue(stringSet, str0);
appeadValue(stringSet, str1);
```

### Functionality

This section will give you a short overview of the functionality of the class `StringSet`.

There are two ways for accessing the sequences in a string set: (1) the function `value` returns a reference to the sequence at a specific position within the sequence of sequences, and (2) `valueById` accesses a sequence given its id. We can retrieve the id of a sequence in a `StringSet` with the function `positionToId`.

```cpp
// (1) Access by position
std::cout << "Owner: " << value(stringSet, 0) << \n;
std::cout << "Position 0: " << value(stringSet, 0) << \n;

// Get the corresponding ids
unsigned id0 = positionToId(stringSet, 0);
unsigned id1 = positionToId(stringSet, 1);

// (2) Access by id
std::cout << "Id 0: " << valueById(stringSet, id0) << \n;
```

**Owner:**

```
Position 0: TATA
Id 0: TATA
```
In the case of Owner StringSets, id and position of a string are always the same, but for Dependent StringSets, the ids can differ from the positions. For example, if a Dependent StringSet is used to represent subsets of strings that are stored in Owner StringSets, one can use the position of the string within the Owner StringSet as id of the strings. With the function assignValueById, we can add the string with a given id from the source string set to the target string set.

```cpp
// Let's create a string set of type dependent to represent strings, // which are stored in the StringSet of type Owner
StringSet<DnaString, Dependent<Tight> > depSet;
// We assign the first two strings of the owner string set to the dependent StringSet, // but in a reverse order
assignValueById(depSet, stringSet, id1);
assignValueById(depSet, stringSet, id0);
```

```
std::cout << "Dependent: " << '"' << '
';
// (1) Access by position
std::cout << "Pos 0: " << value(depSet, 0) << '
';
// (2) Access by id
std::cout << "Id 0: " << valueById(depSet, id0) << '
';
```

```
Dependent:
Position 0: CGCG
Id 0: TATA
```

With the function positionToId we can show that, in this case, the position and the id of a string are different.

```
std::cout << "Position 0: Id " << positionToId(depSet, 0) << '
';
std::cout << "Position 1: Id " << positionToId(depSet, 1) << '
';
```

```
Position 0: Id 1
Position 1: Id 0
```

Also see Efficiently Importing Millions Of Sequences for remarks on ConcatDirect StringSets.

### Iterating over String Sets

As well as for other containers, SeqAn has implemented iterators for StringSets. The general usage of iterators is described in the tutorial Iterators. The following example illustrates, how to iterate over the StringSet.

```cpp
typedef Iterator<StringSet<DnaString> >::Type TStringSetIterator;
for (TStringSetIterator it = begin(stringSet); it != end(stringSet); ++it)
{
    std::cout << *it << '
';
}
```

```
TATA
CGCG
```

If we want to iterate over the contained Strings as well, as if the StringSet would be one sequence, we can use the function concat to get the concatenation of all sequences. Therefore we first use the metafunction Concatenator to receive the type of the concatenation. Then, we can simply build an iterator for this type and iterate over the concatenation of all strings.

```cpp
typedef Concatenator<StringSet<DnaString> >::Type TConcat;
TConcat concatSet = concat(stringSet);
```

```
Iterator<TConcat>::Type it = begin(concatSet);
Iterator<TConcat>::Type itEnd = end(concatSet);
for (; it != itEnd; goNext(it))
```
Assignment 1

Type  Review

Objective  Build a string set with default specialization and which contains the strings "AAA", "CCC", "GGG" and "TTT". After that print the length of the string set and use a simple for-loop to print all elements of the strings set.

Solution  Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    // Build strings
    DnaString str0 = "AAA";
    DnaString str1 = "CCC";
    DnaString str2 = "GGG";
    DnaString str3 = "TTT";
    // Build string set and append strings
    StringSet<DnaString> stringSet;
    appendValue(stringSet, str0);
    appendValue(stringSet, str1);
    appendValue(stringSet, str2);
    appendValue(stringSet, str3);
    // Print the length of the string set
    std::cout << length(stringSet) << std::endl;
    // Print all elements
    for (unsigned i = 0; i < length(stringSet); ++i){
        std::cout << stringSet[i] << std::endl;
    }
    return 0;
}
```

Assignment 2

Type  Application

Objective  In this task you will test, whether a Dependent StringSet contains a string without comparing the actual sequences. Use the given code frame below and adjust it in the following way:

1. Build a Owner StringSet to store the given strings.
2. Get the corresponding ids for each position and store them.
3. Build a DependentStringSet and assign the strings of the owner string set from position 0, 1, and 3 by their id to it.

4. Write a function `isElement` which takes a `StringSet<Dependent<> >` and a `Id` as arguments and checks whether a string set contains a string with a given id.

5. Check if the string set contains the string of position 3 and 2 and print the result.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    int return 0;
}
```

**Hints** You can use the SeqAn functions `positionToId` and `assignValueById`.

**Solution** Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

// Check whether the string set contains the string with the given id,
// without comparing the actual sequences
template<typename TStringSet, typename TId>
bool isElement(TStringSet & stringSet1, TId & id){
    for (unsigned i = 0; i < length(stringSet1); ++i){
        // Get the id of the element at position i
        if (positionToId(stringSet1, i) == id)
            return true;
    }
    return false;
}

int main()
{
    // Build strings
    DnaString str0 = "TATA";
    DnaString str1 = "CGCG";
    DnaString str2 = "TTAAGGCC";
    DnaString str3 = "ATGC";
    DnaString str4 = "AGTGTCA";
    // Your code
}
```
Workshop Assignment 4

Type  Review

Objective  In this assignment, we pick up the example from the workshop assignments from the sequences and iterators tutorials. Take the last solution and change the code to build and use StringSets.

1. Build a StringSet of readList. Reuse the Rooted iterator above.
2. Iterate over the StringSet and print out the values.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **) {
    // Build reads and genomes
    DnaString chr1 = "TATAATATTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Build List containing all reads
typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
```
readList[0] = "TTGCTATCGCGATATGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT";
readList[1] = "TGGATTACGGTCGATATGCTATATATTGCGGCGTACGAACTATGCGGGTACCCGCTTACGAT";
readList[2] = "AGGCTGCGTACGTTGAGTCGTCGTTTAGTCAAGCCGGGCTGTTCATGCTGCAAGCAGACATGCACA";
readList[3] = "CGTGCACTGTCGTCGTCGTCGCCGCTGGCTGGCTGTCGCTGACACATGCACACGTCTCTGTGTTCCGACGTGTGTCACGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCTACTGCTGCTGACA";

// Append a second chromosome sequence fragment to chr1
DnaString chr2 = "AGGCTGCGTACGTTGAGTCGTCGTTTAGTCAAGCCGGGCTGTTCATGCTGCAAGCAGACATGCACA";
append(chr1, chr2);

// Print readlist
std::cout << "\nRead list: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
    std::cout << readList[i] << std::endl;

// Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions.
// Store the start position in a String alignPosList: 7, 100, 172, 272
String<unsigned> alignPosList;
resize(alignPosList, 4);
alignPosList[0] = 7;
alignPosList[1] = 100;
alignPosList[2] = 172;
alignPosList[3] = 272;

// Print alignments using Segment
std::cout << "\nPrint alignment using Segment: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
{
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(readList[i]);
    // Build infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, readList[i]);
}

// Iterators :
// Print alignments using Iterators: Do the same as above, but use Iterators to iterate over the reads.
// First, use Standard Iterators.
Iterator<TDnaList>::Type it = begin(readList);
Iterator<TDnaList, Standard>::Type itEnd = end(readList); //same Iterator as above

std::cout << "\nPrint alignment using Standard Iterators: " << std::endl;
for(; it != itEnd; goNext(it))
{
    // Get the right index for alignPosList
    int i = position(it, readList);
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(value(it));
    // Build Infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, value(it));
}

// Now, use Rooted Iterators.
Iterator<TDnaList, Rooted>::Type it2 = begin(readList);

std::cout << "\nPrint alignment using Rooted Iterators: " << std::endl;
for(; !atEnd(it2); goNext(it2))
{
    int i = position(it2);
// Temporary copy of begin and end position (beginPosition) from alignPosList
// of a given alignment between the read and the genome
unsigned beginPosition = alignPosList[i];
unsigned endPosition = beginPosition + length(value(it2));
// Build Infix
Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
// Call of our function to print the simple alignment
printAlign(genomeFragment, value(it2));
}
return 1;
}

Solution  Click more... to see the solution.

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **){
    // Build reads and genomes
    DnaString chr1 = "TATAATATCTACGATAATGCTAGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Build List containing all reads
    typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATACGATATCGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    readList[1] = "TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    readList[2] = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA";
    readList[3] = "CGTGCACGTGACGTCGTACGTGTTGTCACATCGTCGTGCGTGCTACTGCTGCTGACA";
    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACGTGTGTCACGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCTACTGCTGCTGACA";
    append(chr1, chr2);
    // Print readlist
    std::cout << "
 Read list: " << std::endl;
    for(unsigned i = 0; i < length(readList); ++i)
        std::cout << readList[i] << std::endl;
    // Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions.
    // Store the start position in a String alignPosList: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;
    // Print alignments using Segment
    std::cout << "
 Print alignment using Segment: " << std::endl;
    for(unsigned i = 0; i < length(readList); ++i)
    {
        // Temporary copy of begin and end position (beginPosition) from alignPosList
    }
// of a given alignment between the read and the genome
unsigned beginPosition = alignPosList[i];
unsigned endPosition = beginPosition + length(readList[i]);

// Build Infix
Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
// Call of our function to print the simple alignment
printAlign(genomeFragment, readList[i]);
}

// Iterators :) 
// Print alignments using Iterators: Do the same as above, but use Iterators to iterate over the read list.
// First, use Standard Iterators.
Iterator<TDnaList>::Type it = begin(readList);
Iterator<TDnaList, Standard>::Type itEnd = end(readList); // same Iterator as above

std::cout << "\nPrint alignment using Standard Iterators: " << std::endl;
for(; it != itEnd; goNext(it))
{
    // Get the right index for alignPosList
    int i = position(it, readList);
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(value(it));
    // Build Infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, value(it));
}

// Now, use Rooted Iterators.
Iterator<TDnaList, Rooted>::Type it2 = begin(readList);
std::cout << "\nPrint alignment using Rooted Iterators: " << std::endl;
for(; !atEnd(it2); goNext(it2))
{
    int i = position(it2);
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(value(it2));
    // Build Infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, value(it2));
}

// StringSets
// Build StringSet of readList: Build a StringSet of DnaQString and append the reads from readList.
// Reuse the Rooted Iterator from above.
typedef StringSet<DnaQString> TDnaListSet;
TDnaListSet readStringSet;
goBegin(it2);
for(; !atEnd(it2); goNext(it2))
    appendValue(readStringSet, value(it2));

// Iterate over StringSet
Iterator<TDnaListSet, Rooted>::Type it3 = begin(readStringSet);
std::cout << "Print reads stored in a StringSet using Rooted Iterators: " << std::endl;
for(; !atEnd(it3); goNext(it3))
  std::cout << value(it3) << std::endl;
return 1;
}

2.1.7 Sequences In-Depth

Learning Objective  You will learn in detail how to optimize the usage of sequences dependent on your needs.

Difficulty  Advanced

Duration  20 min

Prerequisites  Sequences

Sequences, particularly Strings, are fundamental in SeqAn. You learned already how to use the default implementation of strings and how to easily work with them. In the most cases the default string specialization is well suited as well as the default behavior for capacity changes. Nevertheless, sometimes you might want to change the default behavior for efficiency reasons and adjust it to your specific needs.

String Specializations

In this section you will learn about the different string specializations and when to use them.

The user can specify the kind of string that should be used in an optional second template argument of String.

String<Dna> dnaSeq1; // The default string implementation: Alloc
String<Dna, Alloc<> > dnaSeq2; // The same as above

In most cases, the implementation Alloc String (the default when using a String<T>) is the best choice. Exceptions are when you want to process extremely large strings that are a bit larger than the available memory (consider Alloc String) or much larger so most of them are stored on the hard disk and only parts of them are loaded in main memory (consider External String).

The following list describes in detail the different specializations:

Specialization Alloc String
  • Description Expandable string that is stored on the heap.
  • Applications The default string implementation that can be used for general purposes.
  • Limitations Changing the capacity can be very costly since all values must be copied.

Specialization Array String
• **Description** Fast but non-expandable string. Fast storing of fixed-size sequences.

• **Limitations** Capacity must already be known at compile time. Not suitable for storing large sequences.

**Specialization Block String**

• **Description** String that stores its sequence characters in blocks.

• **Applications** The capacity of the string can quickly be increased. Good choice for growing strings or stacks.

• **Limitations** Iteration and random access to values is slightly slower than for Alloc String.

**Specialization Packed String**

• **Description** A string that stores as many values in one machine word as possible.

• **Applications** Suitable for storing large strings in memory.

• **Limitations** Slower than other in-memory strings.

**Specialization External String**

• **Description** String that is stored in secondary memory.

• **Applications** Suitable for storing very large strings (>2GB). Parts of the string are automatically loaded from secondary memory on demand.

• **Limitations** Slower than other string classes.

**Specialization CStyle String**

• **Description** Allows adaption of strings to C-style strings.

• **Applications** Used for transforming other String classes into C-style strings (i.e. null terminated char arrays). Useful for calling functions of C-libraries.

• **Limitations** Only sensible if value type is `char` or `wchar_t`.

```cpp
// String with maximum length 100.
String<char, Array<100> > myArrayString;
// String that takes only 2 bits per nucleotide.
String<Dna, Packed<> > myPackedString;
```

**Overflow Strategies**

The following section will describe how you can improve capacity changes for your sequences.

Each sequence object has a capacity, i.e. the reserved space for this object. The capacity can be set explicitly by functions such as `reserve` or `resize`. It can also be set implicitly by functions like `append`, `assign`, `insert` or `replace`, if the operation’s result exceeds the length of the target sequence.

If the current capacity of a sequence is exceeded by changing the length, we say that the sequence overflows. There are several overflow strategies that determine what actually happens when a string should be expanded beyond its capacity. The user can specify this for a function call by additionally handing over a tag. If no overflow strategy is specified, a default overflow strategy is selected depending on the type of the sequence.

The following overflow strategies exist:

**Exact** Expand the sequence exactly as far as needed. The capacity is only changed if the current capacity is not large enough.

**Generous** Whenever the capacity is exceeded, the new capacity is chosen somewhat larger than currently needed. This way, the number of capacity changes is limited in a way that resizing the sequence only takes amortized constant time.
**Limit** Instead of changing the capacity, the contents are limited to current capacity. All values that exceed the capacity are lost.

**Insist** No capacity check is performed, so the user has to ensure that the container’s capacity is large enough.

The next example illustrates how the different strategies could be used:

```cpp
typedef String<Dna> dnaSeq;
// Sets the capacity of dnaSeq to 5.
resize(dnaSeq, 4, Exact());
// Only "TATA" is assigned to dnaSeq, since dnaSeq is limited to 4.
assign(dnaSeq, "TATAGGGG", Limit());
std::cout << dnaSeq << std::endl;
// Use the default expansion strategy.
append(dnaSeq, "GCGCGC");
std::cout << dnaSeq << std::endl;
```

**Workshop Assignment 1**

**Type** Review

**Objective** Build a string of Dna (default specialization) and use the function `appendValue` to append a million times the nucleotide ‘A’. Do it both using the overflow strategy `Exact` and `Generous`. Measure the time for the two different strategies.

**Solution** Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

#include <time.h>

using namespace seqan;

int main()
{
    unsigned num = 1000000;
    time_t start;

    String<Dna> str;
    clear(str);
    start = time(NULL);
    for (unsigned i = 0; i < num; ++i){
        appendValue(str, 'A', Exact());
    }
    std::cout << "Strategy Exact() took: " << time(NULL) - start << " s\n";
    clear(str);
    start = time(NULL);
    for (unsigned i = 0; i < num; ++i){
        appendValue(str, 'A', Generous());
    }
    std::cout << "Strategy Generous() took: " << time(NULL) - start << " s\n";
}
2.1.8 Iterators

Learning Objective  You will learn how to use iterators to traverse containers in SeqAn. After this tutorial, you will be ready to continue with the tutorials about iterating on more complex structures, e.g. Index Iterators.

Difficulty  Basic

Duration  30 min

Prerequisites  Sequences

Iterators are objects that can be used to browse through the values of containers such as Strings or Segments. SeqAn also offers a range of iterators to traverse efficiently more complex data structures, e.g. Graphs, whose specific usage will be explained in the corresponding tutorials. This tutorial will introduce you into the basic concept of iterators using String iterators as illustration.

Defining Iterators

This section will show you how to define different kinds of iterators.

The metafunction Iterator can be used to determine the appropriate iterator type for a given a container. Some containers offer several kinds of iterators, which can be selected by an optional argument of Iterator. For example, the tag Standard can be used to get an iterator type that resembles the C++ standard random access iterator. The more elaborated Rooted iterator, i.e., an iterator that knows its container, can be selected by specifying the Rooted tag. The construction of an iterator in SeqAn, e.g. for a Dna String, could look like the following:

```cpp
Iterator<DnaString>::Type it1; // A standard iterator
Iterator<DnaString, Standard>::Type it2; // Same as above
Iterator<DnaString, Rooted>::Type it3; // A rooted iterator
```

Tip:  The default iterator implementation is Standard. Rooted iterators offer some convenience for the user. They offer additional functions like container for determining the container on which the iterator works, and they simplify the interface for other functions like atEnd. Moreover, rooted iterators may change the container’s length or capacity, which makes it possible to implement a more intuitive variant of a remove algorithm.
While rooted iterators can usually be converted into standard iterators, it is not always possible to convert standard iterators back into rooted iterators, since standard iterators may lack the information about the container they work on. Therefore, many functions that return iterators like `begin` or `end` return rooted iterators instead of standard iterators; this way, they can be used to set both rooted and standard iterator variables. Alternatively it is possible to specify the returned iterator type explicitly by passing the iterator kind as a tag argument, e.g. `begin(str, Standard())`.

Traversing Containers

In this section you will learn how to iterate over a container using the basic functionality of iterators.

An iterator always points to one value of the container. The function `value`, which is equivalent to the `operator*`, can be used to access this value by reference. In contrast `getValue` return a copy of the value. Functions like `goNext` or `goPrevious`, which are equivalent to `operator++` and `operator--` respectively, can be used to move the iterator to other values within the container.

The functions `begin` and `end`, applied to a container, return iterators to the begin and to the end of the container. Note that similar to C++ standard library iterators, the iterator returned by `end` does not point to the last value of the container but to the position behind the last one. If the container is empty then `end() == begin()`.

The following code prints out a sequence and demonstrates how to iterate over a string.

```cpp
dna5_string genome = "ACGTACGTACGT";
typedef iterator<dnas::dna5_string>::type iterator;
for (TIterator it = begin(genome); it != end(genome); goNext(it))
{
    std::cout << value(it);
}

ACGTACGTACGT
```

A Working Example

Let us now clarify the usage of iterators with a working example. The following program demonstrates the usage of iterators.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main() {
    Dna5String genome = "TATANNNGCGCG";
    typedef Iterator<Dna5String>::Type TIterator;
    for (TIterator it = begin(genome); it != end(genome); goNext(it))
    {
        std::cout << *it;
        ++it;
    }
    std::cout << std::endl;
```

The metafunction `Iterator` returns the iterator type for a given container type. In this case the default implementation `Standard` is used.

```cpp
    Dna5String genome = "TATANNNGCGCG";
    Iterator<Dna5String>::Type it = begin(genome);
    Iterator<Dna5String>::Type itEnd = end(genome);
```

We can use iterators to iterate over the elements of a container, e.g. to print the elements.

```cpp
    while (it != itEnd) {
        std::cout << *it;
        ++it;
    }
    std::cout << std::endl;
```
Instead of comparing the two iterators `it` and `itEnd`, we could also use the function `atEnd` to check whether we reached the end of the container.

```cpp
for (goBegin(it, genome); !atEnd(it, genome); goNext(it)){
    std::cout << *it;
}
std::cout << std::endl;
```

Next we will use Rooted Iterators. Since Rooted Iterators know their container, the functions `goBegin` and `atEnd` do not need to get the container as an argument. The following example prints for each element of the Dna5 String `genome` its complement:

```cpp
Iterator<Dna5String, Rooted >::Type it2 = begin(genome);
for (goBegin(it2); !atEnd(it2); goNext(it2)) {
    if (getValue(it2) == 'A')
        std::cout << 'T';
    else if (getValue(it2) == 'T')
        std::cout << 'A';
    else if (getValue(it2) == 'G')
        std::cout << 'C';
    else if (getValue(it2) == 'C')
        std::cout << 'G';
    else std::cout << 'N';
}
std::cout << std::endl;
```

Some iterators support iteration in reverse order with `goPrevious` as you can see in the next example. Note that `goPrevious` is called before the value of `it2` is accessed. Remember that the end position of a container is always the position behind the last item in the container.

```cpp
goEnd(it2);
while (!atBegin(it2)) {
    goPrevious(it2);
    std::cout << getValue(it2);
}
std::cout << std::endl;
```

`assignValue` can be used to change the value of an iterator.

```cpp
assignValue(begin(genome), 'N');
std::cout << genome << std::endl;
```

The output of the program is as follows.

```
TATANNNGCGCG
TATANNNGCGCG
ATATNNNCGCGC
GCGCGNNNATAT
NATANNNGCGCG
```

Assignment 1

Type  Review

Objective
Copy the code below, which replaces all N’s of a given String with A’s. Adjust the code to use iterators to traverse the container. Use the Standard iterator.

Solution

Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    Dna5String genome = "ANTGGTNNAACNGTAANTGCTGANNACATGTNCGCGTGTA";
    Iterator<Dna5String>::Type it = begin(genome);
    Iterator<Dna5String>::Type itEnd = end(genome);

    for (; it != itEnd; goNext(it))
    {
        if (getValue(it) == 'N')
            value(it) = 'A';
    }
    std::cout << "Modified genome: " << genome << std::endl;
    return 0;
}
```

Assignment 2

Type Application

Objective Use the code from above and change the Standard to a Rooted iterator. Try to shorten the code wherever possible.

Solution Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    Dna5String genome = "ANTGGTNNAACNGTAANTGCTGANNACATGTNCGCGTGTA";
    Iterator<Dna5String, Rooted >::Type it = begin(genome);

    for (; !atEnd(it); goNext(it))
    {
        if (getValue(it) == 'N')
            value(it) = 'A';
    }
    std::cout << "Modified genome: " << genome << std::endl;
    return 0;
}
```
Workshop Assignment 3

Type  Review

Objective  In this assignment, we pick up the example from the workshop assignments from the sequences tutorial. Take the last solution and change the code to use Iterators. First, use Standard Iterators to do this.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 = "TATAATATTGCTATCCGGATATCGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGT"
                    "GTGCGATCGATTAGCTAGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    DnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCCGGATATCGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGT"
            "GTGCGATCGATTAGCTAGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    readList[1] = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA";
    readList[2] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA";
    readList[3] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA";

    // Append a second chromosomes sequence fragment to chr1
    DnaString chr2 = "AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA"
                    "TCATGTGCGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA";
    append(chr1, chr2);

    // Print readlist
    std::cout << "
Read list: " << std::endl;
for (unsigned i = 0; i < length(readList); ++i)
    std::cout << readList[i] << std::endl;

    // Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions.
    // Store the start position in a String: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;
```
// Print alignments using Segment
std::cout << " \n Print alignment using Segment: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
{
    // Begin and end position of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(readList[i]);
    // Build infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, readList[i]);
}

// Iterators :) 
// Print alignments using Iterators: Do the same as above, but use Iterators to iterate over the read list.
std::cout << " \n Print alignment using Standard Iterators: " << std::endl;

return 1;
}

Solution  Click more... to see the solution

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 = "TATAATATTGCTATCGGATATCGCTAGCTACGGATTATCGCTGCTGCTATCGGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    append(chr1, chr2);
    std::cout << " \n Read list: " << std::endl;
    for(unsigned i = 0; i < length(readList); ++i)
std::cout << readList[i] << std::endl;

// Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions.
// Store the start position in a String alignPosList: 7, 100, 172, 272
String<unsigned> alignPosList;
resize(alignPosList, 4);
alignPosList[0] = 7;
alignPosList[1] = 100;
alignPosList[2] = 172;
alignPosList[3] = 272;

// Print alignments using Segment
std::cout << "\nPrint alignment using Segment: " << std::endl;
for(unsigned i = 0; i < length(readList); ++i)
{
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(readList[i]);
    // Build infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, readList[i]);
}

// Iterators :
// Print alignments using Iterators: Do the same as above, but use Iterators to iterate over
// First, use Standard Iterators.
Iterator<TDnaList>::Type it = begin(readList);
Iterator<TDnaList, Standard>::Type itEnd = end(readList); //same Iterator as above

std::cout << "\nPrint alignment using Standard Iterators: " << std::endl;
for(; it != itEnd; goNext(it))
{
    // Get the right index for alignPosList
    int i = position(it, readList);
    // Temporary copy of begin and end position (beginPosition) from alignPosList
    // of a given alignment between the read and the genome
    unsigned beginPosition = alignPosList[i];
    unsigned endPosition = beginPosition + length(value(it));
    // Build Infix
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, value(it));
}

return 1;
}

Workshop Assignment 4

Type Review

Objective  Now, use rooted iterators in the example from Workshop Assignment 3.

Solution  Click more... to see the solution.
```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
using namespace seqan;

// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout << "Alignment " << std::endl;
    std::cout << " genome : " << genomeFragment << std::endl;
    std::cout << " read : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 = "TATAATATTGCTAGCTACGATATCAGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTGTGTGCGATCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Build List containing all reads
typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT";
    readList[1] = "TCGATTAGCGCTAGCTACGATATCAGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT";
    readList[2] = "AGCCTGCGTACGTTGCGTGGTTGGTCACATCGTCGTGCGTGCGTGATCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    readList[3] = "CGTGCACTGCGTACGTTGCGTGGTTGGTCACATCGTCGTGCGTGCGTGATCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 = "AGCCTGCGTACGTTGCGTGGTTGGTCACATCGTCGTGCGTGCGTGATCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATTTAG";
    append(chr1, chr2);
    // Print read list
    std::cout << "Read list: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
        std::cout << readList[i] << std::endl;
    // Assume we have mapped the 4 reads to chr1 (and chr2) and now have the mapping start positions
    // Store the start position in a String alignPosList: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;
    // Print alignments using Segment
    std::cout << "Print alignment using Segment: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
    {
        // Temporary copy of begin and end position (beginPosition) from alignPosList
        unsigned beginPosition = alignPosList[i];
        unsigned endPosition = beginPosition + length(readList[i]);
        // Build infix
        Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition);
        // Call of our function to print the simple alignment
        printAlign(genomeFragment, readList[i]);
    }
    // Iterators :
    // Print alignments using Iterators: Do the same as above, but use Iterators to iterate over read data.
    // First, use Standard Iterators.
    Iterator<TDnaList>::Type it = begin(readList);
    Iterator<TDnaList, Standard>::Type itEnd = end(readList); //same Iterator as above
```
Print alignment using Standard Iterators: \\
for(; it != itEnd; goNext(it)) \\
{ 
    // Get the right index for alignPosList 
    int i = position(it, readList); 
    // Temporary copy of begin and end position (beginPosition) from alignPosList 
    // of a given alignment between the read and the genome 
    unsigned beginPosition = alignPosList[i]; 
    unsigned endPosition = beginPosition + length(value(it)); 
    // Build Infix 
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition); 
    // Call of our function to print the simple alignment 
    printAlign(genomeFragment, value(it)); 
} 
// Now, use Rooted Iterators. 
Iterator<TDnaList, Rooted>::Type it2 = begin(readList); 
std::cout << " \n Print alignment using Rooted Iterators: " << std::endl; 
for(; !atEnd(it2); goNext(it2)) 
{ 
    int i = position(it2); 
    // Temporary copy of begin and end position (beginPosition) from alignPosList 
    // of a given alignment between the read and the genome 
    unsigned beginPosition = alignPosList[i]; 
    unsigned endPosition = beginPosition + length(value(it2)); 
    // Build Infix 
    Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition, endPosition); 
    // Call of our function to print the simple alignment 
    printAlign(genomeFragment, value(it2)); 
} 
return 1; 

ToC

Contents

- Alignment Representation
  - Align Data Structure
    - Gap Space vs. Source Space
    - Iterating over Gapped Sequences
    - Assignment 1
  - AlignmentGraph Data Structure
    - Assignment 2

2.1.9 Alignment Representation

Learning Objective This tutorial introduces you to the two data structures that can be used to represent an alignment in SeqAn. You will learn basic techniques to create and modify such data structures and how to access certain information from these data structures.

Difficulty Basic

Duration 0:45h
Before we want to explain SeqAn’s alignment algorithms in detail, we will give you an insight in the underlying data structures that are used to actually represent an alignment in SeqAn. First, we put our focus on the possible representations of alignments and the ways to access and edit different information of an alignment. The two main objects for this purpose are the **Align** and the **Alignment Graph** data structure.

**Align Data Structure**

The **Align** data structure is simply a set of multiple **Gaps** data structures. A Gaps data structure is a container storing gap information for a given source sequence. The gap information is put on top of the source sequence (coordinates of the gapped sequence refer to the gap space) without directly applying them to the source (coordinates of the ungapped sequence refer to the source space). This way operating with gaps sustains very flexible.

There are two specializations available for the Gaps data structures: **Array Gaps** and **Anchor Gaps**. They differ in the way they implement the gap space.

**Note:** In general, using **Array Gaps** is sufficient for most applications. This specialization is also the default one if nothing else is specified. It simply uses an array which stores the counts of gaps and characters in an alternating order. Thus, it is quite efficient to extend existing gaps while it is more expensive to search within the gapped sequence or insert new gaps. Alternatively, one should prefer **Anchor Gaps** if many conversions between coordinates of the gap and the source space are needed as binary search can be conducted to search for specific positions.

Now, let’s start by constructing our first alignment. Before we can make use of any of the mentioned data structures, we need to tell the program where to find the definitions. This can be achieved by including the header file `<seqan/align.h>` which contains the necessary data structures and functions associated with the alignments. The next steps would be to implement the main function of our program and to define the types that we want to use.

```c
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{

    We first define the type of the input sequences (**TSequence**). Then we can define the type of our actual Align object we want to use. In an Align object, the gapped sequences are arranged in rows. You can use the Metafunction **Row** to get the correct type of the used Gaps objects. In the following we use the term **row** to explicitly refer to a gapped sequence as a member of the Align object. We will use the term **gapped sequence** to describe functionalities that is related to the Gaps data structure independent of the Align object.

    **typedef** String<char> TSequence;           // sequence type
    **typedef** Align<TSequence, ArrayGaps> TAlign; // align type
    **typedef** Row<TAlign>::Type TRow;          // gapped sequence type

    After defining the types, we can continue to actually construct our own Align object. Therefore, we need to resize the alignment object in order to reserve space for the sequences we want to add. In our case, we assume a pairwise alignment, hence we reserve space for 2 sequences. With the function **row**, we get access to the gapped sequence at a specific row in the alignment object. This is similar to the **value** function used in **String Sets**. Now, we can assign the source to the corresponding gapped sequence.

    TSequence seq1 = "CDFGDC";
    TSequence seq2 = "CDEFGAHGC";
    TAlign align;
    resize(rows(align), 2);
```
assignSource(row(align,0),seq1);
assignSource(row(align,1),seq2);

After assigning the sources to the gapped sequences, we need to add some gaps to make it to look like a real alignment. You can use the functions `insertGap()` and `removeGap()` to insert and delete one gap or `insertGaps()` and `removeGaps()` to insert and delete multiple gaps in a gapped sequence.

```cpp
std::cout << align;
TRow &row1 = row(align,0);
TRow &row2 = row(align,1);
insertGap(row1,2);
insertGaps(row1,5,2);
std::cout << align;
```

Congratulations! You have created your first alignment. Note that we used a reference declaration `TRow &` for the variables `row1` and `row2`. Without the reference, we would only modify copies of rows and the changes would not effect our `align` object.

**Gap Space vs. Source Space**

In the next steps, we want to dig a little deeper to get a feeling for the gap space and the source space. As mentioned above, the gaps are not inserted into the source but put on top of them in a separate space, the gap space. When inserting gaps, the gap space is modified and all coordinates right of the inserted gap are shifted to the right by the size of the gap. At the same time, the coordinates of the source remain unchanged. Using the function `toSourcePosition()`, we can determine to which position in the source space our current position in the gapped sequence (gap space) maps.

```cpp
std::cout << std::endl << "ViewToSource1: ";
for(unsigned i = 0; i < length(row1); ++i)
    std::cout << toSourcePosition(row1, i) << ", ";
std::cout << std::endl << "ViewToSource2: ";
for(unsigned i = 0; i < length(row2); ++i)
    std::cout << toSourcePosition(row2, i) << ", ";
std::cout << std::endl;
```

If the position in the gap space is actually a gap, then `toSourcePosition()` returns the source position of the next character to the right that is not a gap. Vice versa, we can determine where our current source position maps into the gap space using the function `toViewPosition()`.

```cpp
std::cout << std::endl << "SourceToView1: ";
for(unsigned i = 0; i < length(source(row1)); ++i)
    std::cout << toViewPosition(row1, i) << ", ";
std::cout << std::endl << "SourceToView2: ";
for(unsigned i = 0; i < length(source(row2)); ++i)
    std::cout << toViewPosition(row2, i) << ", ";
std::cout << std::endl;
```

And here is the output of this short example program so far:

```
0 .
CDFGDC
||
CDEFGA
0 .
```
In the first alignment, it seems that the end of the second row is cropped off to match the size of the first one. This effect takes place only in the visualization but is not explicitly applied to the gapped sequence. The second alignment is the one we manually constructed. Here, you can see that the second row is expanded to its full size while it matches the size of the first row. However, it is possible to explicitly crop off the ends of a gapped sequence by using the functions `setClippedBeginPosition` and `setClippedEndPosition`. These functions shrink the gap space and can be understood as defining an infix of the gapped sequence. After the clipping, the relative view position changes according to the clipping and so does the mapping of the source positions to the gap space. The mapping of the view positions to the source space does not change.

```cpp
std::cout << std::endl << "Before clipping:\n" << align;
setClippedBeginPosition(row1, 1);
setClippedEndPosition(row1, 7);
setClippedBeginPosition(row2, 1);
setClippedEndPosition(row2, 7);
std::cout << std::endl << "After clipping:\n" << align;

std::cout << std::endl << "ViewToSource1: ";
for(unsigned i = 0; i < length(row1); ++i)
    std::cout << toSourcePosition(row1, i) << ",";

std::cout << std::endl << "ViewToSource2: ";
for(unsigned i = 0; i < length(row2); ++i)
    std::cout << toSourcePosition(row2, i) << ",";
std::cout << std::endl;

std::cout << std::endl << "SourceToView1: ";
for(unsigned i = 0; i < length(source(row1)); ++i)
    std::cout << toViewPosition(row1, i) << ",";

std::cout << std::endl << "SourceToView2: ";
for(unsigned i = 0; i < length(source(row2)); ++i)
    std::cout << toViewPosition(row2, i) << ",";
std::cout << std::endl;
```

Here the output of the clipping procedure.

**Before clipping:**

```
  0 .
CD-FG--DC
|| || |
CDEFGAHGC
```

**After clipping:**

```
  0 .
D-FG--
||
```
Iterating over Gapped Sequences

In the last part of this section, we are going to iterate over a Gaps object. This can be quite useful if one needs to parse the alignment rows to access position specific information. First, we have to define the type of the Iterator, which can be easily done by using the metafunction Iterator. Remember that we iterate over an TRow object. Then we have to construct the iterators it which points to the begin of row1 using the begin function and itEnd which points behind the last value of row1 using the end function. If you need to refresh the Iterator Concept you can read the Tutorial Iterators. While we iterate over the gapped sequence, we can ask if the current value, at which the iterator it points to, is a gap or not by using the function isGap(). Use gapValue to print the correct gap symbol.

```cpp
typedef Iterator<TRow>::Type TRowIterator;
TRowIterator it = begin(row1);
TRowIterator itEnd = end(row1);
for (; it != itEnd; ++it)
{
    if (isGap(it))
        std::cout << gapValue<char>();
    else
        std::cout << *it;
}
std::cout << std::endl;
```

We will now reset the clipping of row1 using clearClipping and iterate again over it to see its effect.

```cpp
clearClipping(row1);
it = begin(row1);
itEnd = end(row1);
for (; it != itEnd; ++it)
{
    if (isGap(it))
        std::cout << gapValue<char>();
    else
        std::cout << *it;
}
std::cout << std::endl;
```

```cpp
return 0;
```
Here you can see how resetting the clipping positions brings back our complete row.

Assignment 1

Type  Review

Objective  Construct an alignment using the Align data structure for the sequences "ACGTCACCTC" and "ACGGGCCCTATC". Insert two gaps at the second position and insert one gap at the fifth position of the first sequence. Insert one gap at the ninth position of the second sequence. Iterate over the rows of your Align object and print the total count of gaps that exist within the alignment.

Hints  You can use the function countGaps to count the number of consecutive gaps starting from the current position of the iterator.

Solution

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    // Defining all types that are needed.
    typedef String<char> TSequence;
    typedef Align<TSequence, ArrayGaps> TAlign;
    typedef Row<TAlign>::Type TRow;
    typedef Iterator<TRow>::Type TRowIterator;

    TSequence seq1 = "ACGTCACCTC";
    TSequence seq2 = "ACGGGCCCTATC";

    // Initializing the align object.
    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align,0),seq1);
    assignSource(row(align,1),seq2);

    // Use references to the rows of align.
    TRow & row1 = row(align,0);
    TRow & row2 = row(align,1);

    // Insert gaps.
    insertGaps(row1,2,2);
    insertGap(row1,7); // We need to pass the view position which is changed due to the previous insertion.
    insertGaps(row2,9,2);

    // Initialize the row iterators.
    TRowIterator itRow1 = begin(row1);
    TRowIterator itRow2 = begin(row2);

    // Iterate over both rows simultaneously.
    int gapCount = 0;
    for(;itRow1 != itEndRow1; ++itRow1, ++itRow2)
    {
        if(isGap(itRow1))
        {
            gapCount += countGaps(itRow1);
        }
    }
}
```
AlignmentGraph Data Structure

Another very useful representation of alignments is given by the Alignment Graph. It is a graph in which each vertex corresponds to a sequence segment, and each edge indicates an ungapped alignment between the connected vertices, or more precisely between the sequences stored in those vertices. Here is an example of such a graph:

In the following we will actually construct this example step by step. First we include the iostream header from the STL and the <seqan/align.h> header to include all necessary functions and data structures we want to use. We use the namespace seqan and write the main function with an empty body.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{

At the begin of the function we define our types we want to use later on. We define TSequence as the type of our input strings. Since we work with a Dna alphabet we define TSequence as a String over a Dna alphabet. For the AlignmentGraph we need two StringSets. The TStringSet is used to actually store the input sequences and the TDepStringSet is internally used by the AlignmentGraph. That is the AlignmentGraph does not copy the sources into its data structure but rather stores a reference to each of the given input strings as it does not modify the input sequences. The Dependent StringSet facilitates this behavior. In the end we define the actual AlignmentGraph type.

```cpp
typedef String<Dna> TSequence;
typedef StringSet<TSequence> TStringSet;
typedef StringSet<TSequence, Dependent<> > TDepStringSet;
typedef Graph<Alignment<TDepStringSet> > TAlignGraph;
```  

We first create our two input sequences TTGT and TTAGT append them to the StringSet strings using the appendValue function and pass the initialized strings object as a parameter to the constructor of the AlignmentGraph alignG.
TSequence seq1 = "TTGT";
TSequence seq2 = "TTAGT";

TStringSet strings;
appendValue(strings, seq1);
appendValue(strings, seq2);

TAlignGraph alignG(strings);

Before we construct the alignment we print the unmodified AlignmentGraph. Then we add some alignment information to the graph. In order to add an ungapped alignment segment we have to add an edge connecting two nodes of different input sequences. To do so we can use the function addEdge and specify the two vertices that should be connected. Since we do not have any vertices yet, we create them on the fly using the function addVertex. The function addVertex gets as second parameter the id which points to the the correct input sequence within the strings object. We can use the function positionToId() to receive the id that corresponds to a certain position within the underlying Dependent StringSet of the AlignmentGraph. We can access the Dependent StringSet using the function stringSet().

The third parameter of addVertex specifies the begin position of the segment within the respective input sequence and the fourth parameter specifies its length. Now, we add an edge between the two vertices of each input sequence which covers the first two positions. In the next step we have to add a gap. We can do this simply by just adding a vertex that covers the inserted string. Finally we have to add the second edge to represent the last ungapped sequence and then we print the constructed alignment.

::std::cout << alignG << ::std::endl;

addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),0), 0, 2),
    addVertex(alignG, positionToId(stringSet(alignG),1), 0, 2));

addVertex(alignG, positionToId(stringSet(alignG), 1),2,1);

addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),0), 2, 2),
    addVertex(alignG, positionToId(stringSet(alignG),1), 3, 2));

::std::cout << alignG << ::std::endl;

return 0;
}

Here the output of the program. The first output prints the empty adjacency and edge list. The second output prints our desired alignment.

Adjacency list:
Edge list:

Alignment matrix:
0 .
  TT-GT
    ||
TTAGT

The general usage of graphs is explained in the Graphs tutorial.

Assignment 2

Type Review

Objective Construct a multiple sequence alignment using the Alignment Graph data structure. Use the three sequences GARFIELDTHECAT, GARFIELDTHEBIGCAT and THEBIGCAT and align them such that you obtain
the maximal number of matches.

**Hints** The function `findVertex` returns the vertex of an AlignmentGraph that covers the given position in the given sequence.

**Solution**

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    // Define the types we need.
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef StringSet<TSequence, Dependent<> > TDepStringSet;
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph;

    // Initializing the sequences and the string set.
    TSequence seq1 = "GARFIELDTHECAT";
    TSequence seq2 = "GARFIELDTHEBIGCAT";
    TSequence seq3 = "THEBIGCAT";

    TStringSet strings;
    appendString(strings, seq1);
    appendString(strings, seq2);
    appendString(strings, seq3);

    // Load the string set into the Alignment Graph.
    TAlignGraph alignG(strings);

    // Add two vertices covering "GARFIELD" in the first and the second sequence and connect them with an edge.
    addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),0), 0, 8),
            addVertex(alignG, positionToId(stringSet(alignG),1), 0, 8));

    // Add two vertices covering "THE" in the first and the second sequence and connect them with an edge.
    addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),0), 8, 3),
            addVertex(alignG, positionToId(stringSet(alignG),1), 8, 3));

    // Find the vertex covering "THE" in the first sequence and add the vertex covering "THE" in the third sequence and connect them with an edge.
    addEdge(alignG, findVertex(alignG, positionToId(stringSet(alignG),0), 8),
            addVertex(alignG, positionToId(stringSet(alignG),2), 0, 3));

    // Find the vertices covering "THE" in the second and the third sequence and connect them with an edge.
    addEdge(alignG, findVertex(alignG, positionToId(stringSet(alignG),1), 8),
            findVertex(alignG, positionToId(stringSet(alignG),2), 0));

    // Add two vertices covering "FAT" in the second and the third sequence and connect them with an edge.
    addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),1), 11, 3),
            addVertex(alignG, positionToId(stringSet(alignG),2), 3, 3));

    // Add two vertices covering "CAT" in the first and the second sequence and connect them with an edge.
    addEdge(alignG, addVertex(alignG, positionToId(stringSet(alignG),0), 11, 3),
            addVertex(alignG, positionToId(stringSet(alignG),1), 14, 3));

    // Add two vertices covering "CAT" in the first and the second sequence and connect them with an edge.
    addEdge(alignG, findVertex(alignG, positionToId(stringSet(alignG),0), 11),
            findVertex(alignG, positionToId(stringSet(alignG),1), 14));
}
```

2.1. Tutorial 99
```cpp
addVertex(alignG, positionToId(stringSet(alignG), 2), 6, 3));

// Find the vertices covering "CAT" in the second and the third sequence and connect them with an edge.
addEdge(alignG, findVertex(alignG, positionToId(stringSet(alignG), 1), 14), findVertex(alignG, positionToId(stringSet(alignG), 2), 6));

::std::cout << alignG << ::std::endl;

return 0;
```

---

2.1.10 Pairwise Sequence Alignment

**Learning Objective** You will learn how to compute global and local alignments, how you can use different scoring schemes, and how you can customize the alignments to fulfill your needs.

**Difficulty** Average

**Duration** 1h

**Prerequisites** *A First Example, Iterators, Alphabets, Sequences, Alignment Representation*

Alignments are one of the most basic and important ways to measure similarity between two or more sequences. In general, a pairwise sequence alignment is an optimization problem which determines the best transcript of how one sequence was derived from the other. In order to give an optimal solution to this problem, all possible alignments between two sequences are computed using a **Dynamic Programming** approach. Scoring schemes allow the comparison of the alignments such that the one with the best score can be picked. Despite of the common strategy to compute an alignment, there are different variations of the standard DP algorithm laid out for special purposes.
We will first introduce you to the scoring schemes followed by the global alignments. Subsequent, you will learn how to compute local alignments. Finally, we will demonstrate how you can reduce the search space using a band.

**Scoring Schemes**

Scoring schemes define the score for aligning two characters of a given alphabet and the score for gaps within alignments. Given an alignment between two sequences and a scoring scheme, the score of the alignment can be computed as the sum of the scores for aligned character pairs plus the sum of the scores for all gaps.

An example for a scoring scheme is the Levenshtein distance, for which each mismatch between two aligned characters costs 1 and each character that is aligned with a gap costs 1. Translated into scores instead of costs, misalignments get a score of -1 and gaps a score of -1 per character, while matches costs nothing. This scoring scheme is the default for SimpleScore.

SeqAn offers two kinds of scoring scheme:

**Simple Score**  This scoring scheme differentiates between “match” (the two aligned characters are the same), “mismatch” (the two aligned characters are different), and gaps.

The score for a gap of length $k$ is $gapOpen + (k - 1) \cdot gapExtend$. If $gapOpen$ equals $gapExtend$ the score scheme uses linear gap costs, otherwise it uses affine gap costs.

The functions `scoreMatch` and `scoreMismatch` access values for match and mismatch. The function `scoreGap`, or `scoreGapExtend` and `scoreGapOpen` access values for gaps.

Scoring Matrices

These scoring schemes store a score value for each pair of characters. This value can be accessed using `score`. Examples for this kind of scoring scheme are Pam120 and Blosum62. The class `MatrixScore` can be used to store arbitrary scoring matrices. Also see the [Working With Custom Score Matrices](#) on custom scoring matrices.

**Tip:** The order of the different costs in the scoring scheme is `match`, `mismatch`, `gapExtend` and `gapOpen`. If you want to use linear gap costs you could also omit the last parameter `gapOpen` and the scoring scheme would automatically choose the linear gap cost function.

**Global Alignments**

In this section, we want to compute a global alignment using the Needleman-Wunsch algorithm. We will use the Levenshtein distance as our scoring scheme.

A program always starts with including the headers that contain the components (data structures and algorithms) we want to use. To gain access to the alignment algorithms we need to include the `<seqan/align.h>` header file. We tell the program that it has to use the `seqan` namespace and write the `main` function with an empty body.

A good programming practice is to define all types that shall be used by the function at the beginning of the function body. In our case, we define a `TSequence` type for our input sequences and an `Align` object (TAlign) type to store the alignment. For more information on the `Align` datastructure, please read the tutorial [Alignment Representation](#).

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
```
After we defined the types, we can define the variables and objects. First, we create two input sequences `seq1 = "CDFGHC"` and `seq2 = "CDEFGAHC"`. We then define an `align` object where we want to put the sequences into, we resize it to manage two `Gaps` objects, and then assign the sequences to it.

```cpp
typedef String<char> TSequence; // sequence type
typedef Align<TSequence, ArrayGaps> TAlign; // align type
```

```cpp
TSequence seq1 = "CDFGHC";
TSequence seq2 = "CDEFGAHC";
TAlign align;
resize(rows(align), 2);
assignSource(row(align,0),seq1);
assignSource(row(align,1),seq2);
```

Now, we can compute our first alignment. To do so, we simply call the function `globalAlignment` and give as input parameters the `align` object and the scoring scheme representing the Levenshtein distance. The `globalAlignment` function returns the score of the best alignment, which we store in the `score` variable. Afterwards, we print the computed score and the corresponding alignment.

```cpp
int score = globalAlignment(align, Score<int,Simple>(0,-1,-1));
::std::cout << "Score: " << score << ::std::endl;
::std::cout << align << ::std::endl;
return 0;
}
```

The output is as follows:

`Score: -2`

```
0
CD-FG-HC
|| || ||
CDEFGAHC
```

Assignment 1

Type Review

Objective

Compute a global alignment between the DNA sequences "AAATGACGATTG" and "AGTCGGATCTACTG" using the Gotoh algorithm with the following scoring parameters: `match = 4`, `mismatch = -2`, `gapOpen = -4` and `gapExtend = -2`. Store the alignment in an `Align` object and and print it together with the score.

Hints The Gotoh algorithm uses an affine gap function. In SeqAn you can switch between linear and affine gap functions using the scoring scheme by setting different parameters for `gapOpen` and `gapExtend`. Note, the order of the scoring parameters is important. Have a look on the scoring scheme section above if you are not sure about the correct ordering.

Solution First we have to define the body of our program. This includes the definition of the library headers that we want to use. In this case it is the `<iostream>` from the STL and the `<seqan/align.h>` header file defining all algorithms and data structures we want to use. After we added the namespace and opened the `main` body we define our types we want to use in this function. We use an `String` with the `Dna` alphabet, since we know that we work with DNA sequences. The second type is our `Align` object storing the alignment later on.
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<Dna> TSequence;                      // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;         // align type

    In the next step we initialize our objects. This includes the both input sequences seq1 and seq2 and align.
    We resize the underlying set of align that manages the separate Gaps data structures. Finally, we assign the
    input sequences as sources to the corresponding row of align.

    TSequence seq1 = "AAATGACGGATTG";
    TSequence seq2 = "AGTCGGATCTACTG";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align,0),seq1);
    assignSource(row(align,1),seq2);

    Now we compute the alignment using a scoring scheme with affine gap costs. The first parameter corresponds
    to the match value, the second to the mismatch value, the third to the gap extend value and the last one to
    the gap open value. We store the computed score of the best alignment in the equally named variable score.
    In the end we print the score and the alignment using print methods provided by the iostream module of the
    STL.

    int score = globalAlignment(align, Score<int,Simple>{4,-2,-2,-4});
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << align << ::std::endl;

    return 0;
}

Congratulation! You have computed an alignment using affine gap costs. Here the result of the program:

Score: 16

```
  0 . . .
AAATGACGGAT----TG
| | | | | | | |
A---GTCCGGATCTACTG
```
Overlap Alignments

In contrast to the global alignment, an overlap alignment does not penalize gaps at the beginning and at the end of the sequences. This is referred to as **free end-gaps**. It basically means that overlap alignments can be shifted such that the end of the one sequence matches the beginning of the other sequence, while the respective other ends are gapped.

We use the `AlignConfig` object to tell the algorithm which gaps are free. The `AlignConfig` object takes four explicitly defined bool arguments. The first argument stands for **initial gaps** in the vertical sequence of the alignment matrix (first row) and the second argument stands for **initial gaps** in the horizontal sequence (first column). The third parameter stands for **end gaps** in the horizontal sequence (last column) and the fourth parameter stands for **end gaps** in the vertical sequence (last row). Per default the arguments of AlignConfig are set to `false` indicating a standard global alignment as you have seen above. In an overlap alignment all arguments are set to `true`. This means the first row and first column are initialized with zeros and the maximal score is searched in the last column and in the last row.

Just let us compute an overlap alignment to see how it works. We will also make use of the `Alignment Graph` to store the alignment this time. We start again with including the necessary headers and defining all types that we need. We define the `TStringSet` type to store our input sequences in a `StringSet` and we define the `TDepStringSet` which is an `DependentStringSet` used internally by the `AlignmentGraph`. 

![AlignmentConfig Diagram](image.png)
```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence; // sequence type
    typedef StringSet<TSequence> TStringSet; // container for strings
    typedef StringSet<TSequence, Dependent<> > TDepStringSet; // dependent string set
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph; // alignment graph

    // Before we can initialize the AlignmentGraph we append the input sequences to the StringSet strings. Then we simply pass strings as an argument to the constructor of the AlignmentGraph alignG.
    TSequence seq1 = "blablubalu";
    TSequence seq2 = "abba";
    TStringSet sequences;
    appendValue(sequences, seq1);
    appendValue(sequences, seq2);
    TAlignGraph alignG(sequences);

    // Now we are ready to compute the alignment. This time we change two things when calling the globalAlignment function. First, we use an AlignmentGraph to store the computed alignment and second we use the AlignConfig object to compute the overlap alignment.
    int score = globalAlignment(alignG, Score<int, Simple>(1, -1, -1), AlignConfig<true, true, true, true>()) ;
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << alignG << ::std::endl;
    return 0;
}
```

The output is as follows.

Score: 2
Alignment matrix:
0 . :
  blablubalu
|| ||
--ab--ba--

### Assignment 2

**Type** Review

**Objective** Compute a semi-global alignment between the sequences AAATGACGGATTG and TGGGA using the costs 1, -1, -1 for match, mismatch and gap, respectively. Use an AlignmentGraph to store the alignment. Print the score and the resulting alignment to the standard output.

**Hint** A semi-global alignment is a special form of an overlap alignment often used when aligning short sequences against a long sequence. Here we only allow free free end-gaps at the beginning and the end of the shorter sequence.

**Solution** First we have to define the body of our program. This includes the definition of the library headers that we want to use. In this case we include the `iostream` header from the STL and the `<seqan/align.h>`
header, which defines all algorithms and data structures we want to use. After we added the namespace and opened the main function body we define our types we want to use in this function. We use an String with the Dna alphabet, since we know that we work with DNA sequences. We use an additional StringSet to store the input sequences. In this scenario we use an AlignmentGraph to store the alignment. Remember, that the AlignmentGraph uses an DependentStringSet to map the vertices to the correct input sequences.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<Dna> TSequence; // sequence type
    typedef StringSet<TSequence> TStringSet; // container for strings
    typedef StringSet<TSequence, Dependent<> > TDepStringSet; // dependent string set
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph; // alignment graph

    In the next step we initialize our input StringSet strings and pass it as argument to the constructor of the AlignmentGraph alignG.

    TSequence seq1 = "AAATGACGGATTG";
    TSequence seq2 = "TGGGA";
    TStringSet sequences;
    appendValue(sequences,seq1);
    appendValue(sequences,seq2);
    TAlignGraph alignG(sequences);

    Now we compute the alignment using the Levenshtein distance and a AlignConfig object to set the correct free end-gaps. In this example we put the shorter sequence on the vertical axis of our alignment matrix. Hence, we have to use free end-gaps in the first and last row, which corresponds to the first and the last parameter in the AlignConfig object. If you add the shorter sequence at first to strings, then you simply have to flip the bool values of the AlignConfig object.

    int score = globalAlignment(alignG, Score<int,Simple>{1,-1,-1}, AlignConfig<true, false, false, true>();
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << alignG << ::std::endl;

    return 0;
}
```

Here the result of the program.

Score: 3
Alignment matrix:
0 . .
AAATGACGGATTG
|| ||||
---TG--GGA---

Specialized Alignments

SeqAn offers specialized algorithms that can be selected using a tag. Note that often these specializations are restricted in some manner. The following list shows different alignment tags for specialized alignment algorithms and the restrictions of the algorithms.
**Hirschberg** The Hirschberg algorithm computes an alignment between two sequences in linear space. The algorithm can only be used with an Align object (or Gaps). It uses only linear gap costs and does no overlap alignments.

**MyersBitVector** The MyersBitVector is a fast alignment specialization using bit parallelism. It only works with the Levenshtein distance and outputs no alignments.

**MyersHirschberg** The MyersHirschberg is a combination of the rapid MyersBitVector and the space efficient Hirschberg algorithm, which additionally enables the computation of an alignment. It only works with the Levenshtein distance and for Align objects.

**Tip:** In SeqAn you can omit the computation of the traceback to get only the score by using the function `globalAlignmentScore`. This way you can use the alignment algorithms for verification purposes, etc.

In the following example, we want to compute a global alignment of two sequences using the Hirschberg algorithm. We are setting the match score to 1, and mismatch as well as gap penalty to −1. We print the alignment and the score.

First the necessary includes and typedefs:

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence; // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign; // align type

    TSequence seq1 = "GARFIELDTHECAT";
    TSequence seq2 = "GARFIELDTHEBIGCAT";

    TAlign align;
    resize(rows(align),2);
    assignSource(row(align,0),seq1);
    assignSource(row(align,1),seq2);

    int score = globalAlignment(align,Score<int,Simple>(1,-1,-1),Hirschberg());
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << align << ::std::endl;

    return 0;
}
```

The output is as follows.

```
Score: 11

0 . : .
GARFIELDTHE---CAT
|||********| |||
GARFIELDTHEBIGCAT
```

In the following example, we want to compute a global alignment of two sequences using the Hirschberg algorithm. We are setting the match score to 1, and mismatch as well as gap penalty to −1. We print the alignment and the score.

First the necessary includes and typedefs:

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence; // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign; // align type

    TSequence seq1 = "GARFIELDTHECAT";
    TSequence seq2 = "GARFIELDTHEBIGCAT";

    TAlign align;
    resize(rows(align),2);
    assignSource(row(align,0),seq1);
    assignSource(row(align,1),seq2);

    int score = globalAlignment(align,Score<int,Simple>(1,-1,-1),Hirschberg());
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << align << ::std::endl;

    return 0;
}
```

The output is as follows.

```
Score: 11

0 . : .
GARFIELDTHE---CAT
|||********| |||
GARFIELDTHEBIGCAT
```
Assignment 3

Type  Application

Objective  Write a program that computes a fast global alignment between the Rna sequences AAGUGACUUUAUUG and AGUCGGAUCUACUG using the Align data structure and the Levenshtein distance. Print the score and the alignment. Additionally, output for each row of the Align object the view positions of the gaps.

Hint  You can use an iterator to iterate over a row. Use the metafunction Row to get the type of the row used by the Align object. Use the function isGap to check whether the current value of the iterator is a gap or not. The gaps are already in the view space.

Solution  As usual, first the necessary includes and typedefs. Our sequence type is String<Dna>. TAlign and TRow are defined as in the previous example program. The type Iterator<TRow>::Type will be used to iterate over the rows of the alignment.

```cpp
#include <iostream>
#include <seqan/align.h>
using namespace seqan;

int main()
{
    typedef String<Rna> TSequence;
    typedef Align<TSequence, ArrayGaps> TAlign;
    typedef Row<TAlign>::Type TRow;
    typedef Iterator<TRow>::Type TRowIterator;

    TSequence seq1 = "AAGUGACUUUAUUG";
    TSequence seq2 = "AGUCGGAUCUACUG";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);

    unsigned aliLength = _max(length(row(align, 0)), length(row(align, 1)));  
    for(unsigned i = 0; i < length(rows(align)); ++i)
    {
        TRowIterator it = iter(row(align, i), 0);
        TRowIterator itEnd = iter(row(align, i), aliLength);
        unsigned pos = 0;
        std::cout << "Row " << i << " contains gaps at positions: ";
        while (it != itEnd)
        {
            if(isGap(it))
                std::cout << pos << std::endl;
        }
    }
}
```

In the next step we initialize our input StringSet strings and pass it as argument to the constructor of the AlignmentGraph alignG.

```cpp
TSequence seq1 = "AAGUGACUUUAUUG";
TSequence seq2 = "AGUCGGAUCUACUG";

TAlign align;
resize(rows(align), 2);
assignSource(row(align, 0), seq1);
assignSource(row(align, 1), seq2);
```

Now we compute the alignment using the levenshtein distance and a AlignConfig object to set the correct free end-gaps. In this example we put the shorter sequence on the vertical axis of our alignment matrix. Hence, we have to use free end-gaps in the first and last row, which corresponds to the first and the last parameter in the AlignConfig object. If you add the shorter sequence at first to strings, then you simply have to flip the bool values of the AlignConfig object.

```cpp
unsigned aliLength = _max(length(row(align, 0)), length(row(align, 1)));  
for(unsigned i = 0; i < length(rows(align)); ++i)
{
    TRowIterator it = iter(row(align, i), 0);
    TRowIterator itEnd = iter(row(align, i), aliLength);
    unsigned pos = 0;
    std::cout << "Row " << i << " contains gaps at positions: ";
    while (it != itEnd)
    {
        if(isGap(it))
            std::cout << pos << std::endl;
    }
```
++it;
++pos;
}
}

return 0;
}

The output of the program is as follows.

Score: -6

|AAGU--GA-CUUAUUG
| || || || | ||
|A-GUCGGAUCU-ACUG

Row 0 contains gaps at positions:
4
5
8

Row 1 contains gaps at positions:
1
11

Local Alignments

Now let’s look at local pairwise alignments.

SeqAn offers the classical Smith-Waterman algorithm that computes the best local alignment with respect to a given scoring scheme, and the Waterman-Eggert algorithm, which computes not only the best but also suboptimal local alignments.

We are going to demonstrate the usage of both in the following example where first the best local alignment of two character strings and then all local alignments of two DNA sequences with a score greater than or equal to 4 are computed.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    Align< String<char> > ali;
    resize(rows(ali), 2);
    assignSource(row(ali, 0), "aphilologicaltheorem");
    assignSource(row(ali, 1), "bizarreamphibology");

    Now the best alignment given the scoring parameters is computed by the function localAlignment. The returned score value is printed directly, and the alignment itself in the next line. The functions clippedBeginPosition and clippedEndPosition can be used to retrieve the begin and end position of the matching subsequences within the original sequences.
Next, several local alignments of the two given DNA sequences are going to be computed. First, the `Align` object is created.

```
Align< String<Dna> > ali2;
resize(rows(ali2), 2);
assignSource(row(ali2, 0), "ataagcgtctcg");
assignSource(row(ali2, 1), "tcatagagttgc");
```

A `LocalAlignmentEnumerator` object needs to be initialized on the `Align` object. In addition to the Align object and the scoring scheme, we now also pass the `finder` and a minimal score value, 4 in this case, to the `localAlignment` function. The `WatermanEggert` tag specifies the desired Waterman-Eggert algorithm. While the score of the local alignment satisfies the minimal score cutoff, the alignments are printed with their scores and the subsequence begin and end positions.

```
Score<int> scoring(2, -1, -2, 0);
LocalAlignmentEnumerator<Score<int>, Unbanded> enumerator(scoring, 5);
while (nextLocalAlignment(ali2, enumerator))
{
    std::cout << "Score = " << getScore(enumerator) << std::endl;
    std::cout << ali2;
    std::cout << "Aligns Seq1[" << clippedBeginPosition(row(ali2, 0)) << "":" << (clippedEndPosition(row(ali2, 0))-1) << "] and Seq2[" << clippedBeginPosition(row(ali2, 1)) << "":" << (clippedEndPosition(row(ali2, 1))-1) << "]" << std::endl << std::endl;
}
return 0;
```

Here is the output of the first part of our example program:

```
Score = 19
0 . :
  a-philolog
| ||||| |
  amphibolog

Aligns Seq1[0:9] and Seq2[7:16]
```

The second part outputs two alignments:

```
Score = 9
0 .
ATAAGCGT
|| |||
ATA-GAGT

Aligns Seq1[0:7] and Seq2[2:9]
Score = 5
0 .
TC-TCG
|| |
TCATAG
```
Assigns Seq1[7:12] and Seq2[0:5]

Assignment 4

Type Review

Objective Write a program which computes the 3 best local alignments of the two AminoAcid sequences “PNCFDAKQRTASRPL” and “CFDKQKNNRTATRDTA” using the following scoring parameters: match = 3, mismatch = -2, gap open = -5, gap extension = -1.

Hint Use an extra variable to enumerate the <tt>k</tt> best alignments.

Solution The usual includes.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{

The initialization of the Align object.

    Align< String<AminoAcid> > ali;
    resize(rows(ali), 2);
    assignSource(row(ali, 0), "PNCFDAKQRTASRPL");
    assignSource(row(ali, 1), "CFDKQKNNRTATRDTA");

Computing the three best alignments with the desired scoring parameters:

    Score<int> sc(3,-2,-1,-5);
    unsigned count = 0;
    LocalAlignmentEnumerator<Score<int>, Unbanded> enumerator(sc);
    while (nextLocalAlignment(ali, enumerator) && count < 3)
    {
        std::cout << "Score = " << getScore(enumerator) << std::endl;
        std::cout << ali;
        ++count;
    }

    return 0;
}
```

The resulting output is as follows.

Score = 21

0 . :
CFDAKQ---RTASR
|| | | || || |
CFDKQKNNRTATR

Score = 8

0 .
AKQR-TA
| | ||
AT-RDTA
Often it is quite useful to reduce the search space in which the optimal alignment can be found, e.g., if the sequences are very similar, or if only a certain number of errors is allowed. To do so you can define a band, whose intersection with the alignment matrix defines the search space. To define a band we have to pass two additional parameters to the alignment function. The first one specifies the position where the lower diagonal of the band crosses the vertical axis. The second one specifies the position where the upper diagonal of the band crosses the horizontal axis. You can imagine the matrix as the fourth quadrant of the Cartesian coordinate system. Then the main diagonal of an alignment matrix is described by the function $f(x) = -x$ and all diagonals that crosses the vertical axis below this point are specified with negative values and all diagonals that crosses the horizontal axis right of it are specified with positive values (see image). A given band is valid as long as the relation $\text{lower diagonal} \leq \text{upper diagonal}$ holds. In case of equality, the alignment is equivalent to the hamming distance problem, where only substitutions are considered.

Important: The alignment algorithms return `MinValue<ScoreValue>::VALUE` if a correct alignment cannot be computed due to invalid compositions of the band and the specified alignment preferences. Assume, you compute a global alignment and the given band does not cover the last cell of the alignment matrix. In this case it is not possible...
to compute a correct alignment, hence `MinValue<ScoreValue>::VALUE` is returned, while no further alignment information are computed.

Let’s compute a banded alignment. The first step is to write the `main` function body including the type definitions and the initializations.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;  // sequence type
    typedef Align<TSequence,ArrayGaps> TAlign;  // align type

    TSequence seq1 = "CDFGHC";
    TSequence seq2 = "CDEFGAHC";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);

    After we initialized everything, we will compute the banded alignment. We pass the values -2 for the lower diagonal and 2 for the upper diagonal.

    int score = globalAlignment(align, Score<int,Simple>(0,-1,-1),-2,2);
    ::std::cout << "Score: " << score << ::std::endl;
    ::std::cout << align << ::std::endl;
    return 0;
}
```

And here is the output:

```
Score: -2
0
  CD-FG-HC
|| || ||
CDEFGAHC
```

**Assignment 5**

**Type** Transfer

**Objective** Write an approximate pattern matching algorithm using alignment algorithms. Report the positions of all hits where the pattern matches the text with at most 2 errors. Output the number of total edits used to match the pattern and print the corresponding cigar string of the alignment without leading and trailing gaps in the pattern.

**Text:** "MISSISSIPPIANDMISSOURI" **Pattern:** "SISSI"

**Hint**

- The first step would be to verify at which positions in the text the pattern matches with at most 2 errors.
- Use the `infix` function to return a subsequence of a string.
- A CIGAR string is a different representation of an alignment. It consists of a number followed by an operation. The number indicates how many consecutive operations of the same type are executed. Operations
can be <tt>M</tt> for match, <tt>S</tt> for mismatch, <tt>I</tt> for insertion and <tt>D</tt> for deletion. Here is an example:

ref: AC--GTCATTT
r01: ACGTCTCA---
Cigar of r01: 2M2I1S3M3D

Solution (Step 1)

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text = "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text,i,i+length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp,pattern,MyersBitVector()) >= -2)
        {
            appendValue(locations,i);
        }
    }
    return 0;
}
```

Solution (Step 2)

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text = "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
```
for (unsigned i = 0; i < length(text) - length(pattern); ++i)
{
    // Compute the MyersBitVector in current window of text.
    TSequence tmp = infix(text,i,i+length(pattern));

    // Report hits with at most 2 errors.
    if (globalAlignmentScore(tmp,pattern,MyersBitVector()) >= -2)
    {
        appendValue(locations,i);
    }
}

TGaps gapsText;
TGaps gapsPattern;
assignSource(gapsPattern, pattern);
::std::cout << "Text: " << text << "\nPattern: " << pattern << ::std::endl;
for (TIterator it = begin(locations); it != end(locations);++it)
{
    // Clear previously computed gaps.
    clearGaps(gapsText);
    clearGaps(gapsPattern);

    // Only recompute the area within the current window over the text.
    TSequence textInfix = infix(text,*it,*it + length(pattern));
    assignSource(gapsText, textInfix);

    // Use semi-global alignment since we do not want to track leading/trailing gaps
    // Restrict search space using a band allowing at most 2 errors in vertical/horizontal direction.
    int score = globalAlignment(gapsText,gapsPattern,Score<int>{0,-1,-1},AlignConfig<true,false,false,true>(),-2,2);
    ::std::cout << "Hit at position " << *it << "\ntotal edits: " << abs(score) << ::std::endl;
}
return 0;

---

Solution (Step 3)

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int>> ::Type TIterator;

    TSequence text = "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text,i,i+length(pattern));
```
// Report hits with at most 2 errors.
if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
{
    appendValue(locations, i);
}

TGaps gapsText;
TGaps gapsPattern;
assignSource(gapsPattern, pattern);
::std::cout << "Text: " << text << "\nPattern: " << pattern << ::std::endl;
for (TIterator it = begin(locations); it != end(locations); ++it)
{
    // Clear previously computed gaps.
    clearGaps(gapsText);
clearGaps(gapsPattern);

    // Only recompute the area within the current window over the text.
    TSequence textInfix = infix(text, *it, *it + length(pattern));
    assignSource(gapsText, textInfix);

    // Use semi-global alignment since we do not want to track leading/trailing gaps in pattern.
    // Restrict search space using a band allowing at most 2 errors in vertical/horizontal direction.
    int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -1), AlignConfig<true, false, false, true>(), -2, 2);
    TGapsIterator itGapsPattern = begin(gapsPattern);
    TGapsIterator itGapsEnd = end(gapsPattern);

    // Remove trailing gaps in pattern.
    int count = 0;
    while (isGap(--itGapsEnd))
    {++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);
    }

    // Remove leading gaps in pattern.
    if (isGap(itGapsPattern))
    {
        setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
        setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
    }
::std::cout << "Hit at position " << *it << " total edits: " << abs(score) << ::std::endl;
}
return 0;

Solution (Step 4)
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
typedef StringSet<TSequence> TStringSet;
typedef Gaps<TSequence, ArrayGaps> TGaps;
typedef Iterator<TGaps>::Type TGapsIterator;
```cpp
typedef Iterator<String<int>> :: Type TIterator;

TSequence text = "MISSISSIPPIANDMISSOURI";
TSequence pattern = "SISSI";

String<int> locations;
for (unsigned i = 0; i < length(text) - length(pattern); ++i) {
    // Compute the MyersBitVector in current window of text.
    TSequence tmp = infix(text, i, i + length(pattern));

    // Report hits with at most 2 errors.
    if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2) {
        appendValue(locations, i);
    }
}

TGaps gapsText;
TGaps gapsPattern;
assignSource(gapsPattern, pattern);
::std::cout << "Text: " << text << "\nPattern: " << pattern << ::std::endl;
for (TIterator it = begin(locations); it != end(locations); ++it) {
    // Clear previously computed gaps.
    clearGaps(gapsText);
    clearGaps(gapsPattern);

    // Only recompute the area within the current window over the text.
    TSequence textInfix = infix(text, *it, *it + length(pattern));
    assignSource(gapsText, textInfix);

    // Use semi-global alignment since we do not want to track leading/trailing gaps in pattern.
    // Restrict search space using a band allowing at most 2 errors in vertical/horizontal direction.
    int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -1), AlignConfig<true, false, false, true>(), -2, 2);

    TGapsIterator itGapsPattern = begin(gapsPattern);
    TGapsIterator itGapsEnd = end(gapsPattern);

    // Remove trailing gaps in pattern.
    int count = 0;
    while (isGap(--itGapsEnd))
        ++count;
    setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

    // Remove leading gaps in pattern.
    if (isGap(itGapsPattern)) {
        setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
        setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
    }

    // Reinitialize the iterators.
    TGapsIterator itGapsText = begin(gapsText);
    itGapsPattern = begin(gapsPattern);
    itGapsEnd = end(gapsPattern);

    // Use a stringstream to construct the cigar string.
```
::std::stringstream cigar;

while (itGapsPattern != itGapsEnd)
{
    // Count insertions.
    if (isGap(itGapsText))
    {
        int numGaps = countGaps(itGapsText);
        cigar << numGaps << "I";
        itGapsText += numGaps;
        itGapsPattern += numGaps;
        continue;
    }
    ++itGapsText;
    ++itGapsPattern;
}
::std::cout << "Hit at position " << *it << "total edits: " << abs(score) << ::std::endl;

return 0;

Solution (Step 5)

#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
typedef StringSet<TSequence> TStringSet;
typedef Gaps<TSequence, ArrayGaps> TGaps;
typedef Iterator<TGaps>::Type TGapsIterator;
typedef Iterator<String<int> >::Type TIterator;

TSequence text = "MISSISSIPPIANDMISSOURI";
TSequence pattern = "SISSI";

String<int> locations;
for (unsigned i = 0; i < length(text) - length(pattern); ++i)
{
    // Compute the MyersBitVector in current window of text.
    TSequence tmp = infix(text,i,i+length(pattern));

    // Report hits with at most 2 errors.
    if (globalAlignmentScore(tmp,pattern,MyersBitVector()) >= -2)
    {
        appendValue(locations,i);
    }
}

TGaps gapsText;
TGaps gapsPattern;
assignSource(gapsPattern, pattern);
::std::cout << "Text: " << text << "Pattern: " << pattern << ::std::endl;
for (TIterator it = begin(locations); it != end(locations); ++it)
{
    // Clear previously computed gaps.
    clearGaps(gapsText);
}
clearGaps(gapsPattern);

// Only recompute the area within the current window over the text.
TSequence textInfix = infix(text,*it,*it + length(pattern));
assignSource(gapsText, textInfix);

// Use semi-global alignment since we do not want to track leading/trailing gaps in pattern.
// Restrict search space using a band allowing at most 2 errors in vertical/horizontal direction.
int score = globalAlignment(gapsText,gapsPattern,Score<int>(0,-1,-1),AlignConfig(true,false,false,true>(),-2,2);

TGapsIterator itGapsPattern = begin(gapsPattern);
TGapsIterator itGapsEnd = end(gapsPattern);

// Remove trailing gaps in pattern.
int count = 0;
while(isGap(--itGapsEnd))
  ++count;
setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

// Remove leading gaps in pattern.
if(isGap(itGapsPattern))
{
  setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
  setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
}

// Reinitialize the iterators.
TGapsIterator itGapsText = begin(gapsText);
itGapsPattern = begin(gapsPattern);
itGapsEnd = end(gapsPattern);

// Use a stringstream to construct the cigar string.
::std::stringstream cigar;
while (itGapsPattern != itGapsEnd)
{
  // Count insertions.
  if (isGap(itGapsText))
  {
    int numGaps = countGaps(itGapsText);
    cigar << numGaps << "I";
    itGapsText += numGaps;
    itGapsPattern += numGaps;
    continue;
  }
  // Count deletions.
  if (isGap(itGapsPattern))
  {
    int numGaps = countGaps(itGapsPattern);
    cigar << numGaps << "D";
    itGapsText += numGaps;
    itGapsPattern += numGaps;
    continue;
  }
  ++itGapsText;
  ++itGapsPattern;
}

// Output the hit position in the text, the total number of edits and the corresponding cigar string.
::std::cout << "Hit at position " << *it << "\total edits: " << abs(score) << ::std::endl;
```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text = "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text,i,i+length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp,pattern,MyersBitVector()) >= -2)
        {
            appendValue(locations,i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    ::std::cout << "Text: " << text << "\nPattern: " << pattern << ::std::endl;
    for (TIterator it = begin(locations); it != end(locations);++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text,*it,*it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/trailing gaps in the pattern.
        // Restrict search space using a band allowing at most 2 errors in vertical/horizontal direction.
        int score = globalAlignment(gapsText,gapsPattern,Score<int>(0,-1,-1),AlignConfig<true,false,false,true>(),-2,2);

        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
    }
}
```
while(isGap(--itGapsEnd))
  ++count;
setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

// Remove leading gaps in pattern.
if(isGap(itGapsPattern))
{
  setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
  setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
}

// Reinitialize the iterators.
TGapsIterator itGapsText = begin(gapsText);
itGapsPattern = begin(gapsPattern);
itGapsEnd = end(gapsPattern);

// Use a stringstream to construct the cigar string.
::std::stringstream cigar;
int numChar = 0;
while (itGapsPattern != itGapsEnd)
{
  // Count insertions.
  if (isGap(itGapsText))
  {
    int numGaps = countGaps(itGapsText);
    cigar << numGaps << "I";
    itGapsText += numGaps;
    itGapsPattern += numGaps;
    continue;
  }
  // Count deletions.
  if (isGap(itGapsPattern))
  {
    int numGaps = countGaps(itGapsPattern);
    cigar << numGaps << "D";
    itGapsText += numGaps;
    itGapsPattern += numGaps;
    continue;
  }
  // Count matches.
  while (*itGapsText == *itGapsPattern && itGapsPattern != itGapsEnd)
  {
    ++numChar;
    ++itGapsText;
    ++itGapsPattern;
  }
  if (numChar != 0)
  {
    cigar << numChar << "M";
    numChar = 0;
    continue;
  }

  // Count mismatches.
  while (*itGapsText != *itGapsPattern && itGapsPattern != itGapsEnd)
  {
    ++numChar;
    ++itGapsText;
    ++itGapsPattern;
  }
}
+++itGapsPattern;
}
if (numChar != 0)
cigar << numChar << "S";
numChar = 0;

// Output the hit position in the text, the total number of edits and the corresponding cigar string.
::std::cout << "Hit at position " << *it << "\total edits: " << abs(score) << "\to
}

return 0;
}

Complete Solution (and more explanations) Write the main body of the program with type definition and initialization of the used data structures.

```cpp
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{

typedef String<char> TSequence;
typedef StringSet<TSequence> TStringSet;
typedef Gaps<TSequence, ArrayGaps> TGaps;
typedef Iterator<TGaps>::Type TGapsIterator;
typedef Iterator<String<int> >::Type TIterator;

TSequence text = "MISSISSIPPIANDMISSOURI";
TSequence pattern = "SISSI";

String<int> locations;
for (unsigned i = 0; i < length(text) - length(pattern); ++i)
{
    // Compute the MyersBitVector in current window of text.
    TSequence tmp = infix(text, i, i + length(pattern));

    // Report hits with at most 2 errors.
    if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
    {
        appendValue(locations, i);
    }
}

In the first part of the algorithm we implement an alignment based verification process to identify positions in the database at which we can find our pattern with at most 2 errors. We slide the 5*5 alignment matrix position by position over the database and use the MeyersBitVector to verify the hits. If the score is greater or equal than -2, then we have found a hit. We store the begin position of the hit in locations.

String<int> locations;
for (unsigned i = 0; i < length(text) - length(pattern); ++i)
{
    // Compute the MyersBitVector in current window of text.
    TSequence tmp = infix(text, i, i + length(pattern));

    // Report hits with at most 2 errors.
    if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
    {
        appendValue(locations, i);
    }
}

In the second part of the algorithm we iterate over all reported locations. This time we compute a semi-global alignment since we won’t penalize gaps at the beginning and at the end of our pattern. We also compute a band allowing at most 2 errors in either direction. Don’t forget to clear the gaps in each iteration, otherwise we might encounter wrong alignments.

TGaps gapsText;
TGaps gapsPattern;
assignSource(gapsPattern, pattern);
::std::cout << "Text: " << text << "\Pattern: " << pattern << "\to
```
for (TIterator it = begin(locations); it != end(locations); ++it)
{
    // Clear previously computed gaps.
    clearGaps(gapsText);
    clearGaps(gapsPattern);
    // Only recompute the area within the current window over the text.
    TSequence textInfix = infix(text, *it, *it + length(pattern));
    assignSource(gapsText, textInfix);

    // Use semi-global alignment since we do not want to track leading/trailing gaps.
    // Restrict search space using a band allowing at most 2 errors in vertical/horizontal.
    int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -1), AlignConfig(true, false, false, true), -2, 2);

In the next part we determine the cigar string for the matched pattern. We have to remove leading and trailing gaps in the gapsPattern object using the functions setClippedBeginPosition and setClippedEndPosition. We also need to set the clipped begin position for the gapsText object such that both Gaps begin at the same position.

    TGapsIterator itGapsPattern = begin(gapsPattern);
    TGapsIterator itGapsEnd = end(gapsPattern);

    // Remove trailing gaps in pattern.
    int count = 0;
    while (isGap(--itGapsEnd))
        ++count;
    setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

    // Remove leading gaps in pattern.
    if (isGap(itGapsPattern))
    {
        setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
        setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
    }

    // Reinitialize the iterators.
    TGapsIterator itGapsText = begin(gapsText);
    itGapsPattern = begin(gapsPattern);
    itGapsEnd = end(gapsPattern);

    // Use a stringstream to construct the cigar string.
    ::std::stringstream cigar;
    int numChar = 0;
    while (itGapsPattern != itGapsEnd)
    {

    First, we identify insertions using the functions isGap and countGaps.

        // Count insertions.
        if (isGap(itGapsText))
        {
            int numGaps = countGaps(itGapsText);
            cigar << numGaps << "I";
            itGapsText += numGaps;
            itGapsPattern += numGaps;
        }

We do the same to identify deletions.
If there is neither an insertion nor a deletion, then there must be a match or a mismatch. As long as we encounter matches we move forward in the Gaps structures and count the number of consecutive matches. When we are done we report the match count.

In a similar procedure we determine the consecutive substitutions. Finally we print out the position of the hit, its total number of edits and the corresponding cigar string.

Here is the output of this program.

Text: MISSISSIPPIANDMISSOURI Pattern: SISSI
Hit at position 0 total edits: 1 cigar: 1S1M
Hit at position 1 total edits: 1 cigar: 1I1M
Hit at position 2 total edits: 1 cigar: 1M1I
Hit at position 3 total edits: 0 cigar: 5M
Hit at position 4 total edits: 1 cigar: 1I1M
Hit at position 6 total edits: 2 cigar: 2M2S1M
Hit at position 14 total edits: 2 cigar: 1S3M1S
2.1.11 Multiple Sequence Alignment

Learning Objective  You will learn how to compute a multiple sequence alignment using SeqAn’s alignment data structures and algorithms.

Difficulty  Basic
Duration  15 min
Prerequisites  A First Example, Sequences, Alphabets, Alignment Representation

Apart from pairwise alignments, also multiple sequence alignments can be computed in SeqAn. The easiest way to do this is by using the function globalMsaAlignment. This function computes a heuristic alignment based on a consistency-based progressive alignment strategy as described in SeqAn::Tcoffee paper.

In the following example, we align four amino acid sequences using the AlignmentGraph data structure and the Blosum62 scoring matrix with gap extension penalty -11 and gap open penalty -1. The required header for multiple sequence alignments is <seqan/graph_msa.h>.

```cpp
#include <iostream>
#include <seqan/align.h>
#include <seqan/graph_msa.h>
using namespace seqan;

int main()
{
    typedef String<AminoAcid> TSequence;
    StringSet<TSequence> seq;
    appendValue(seq,"DPKKPRGKMSSYAFFVQTSREEHKHHPDASVNFSEFSKCSERWKTMASAEKGKFEDMAKADKARYEREMKTYIPPKGE");
    appendValue(seq,"RVKRPMNAFIVWSRDQRRKMALENFRMNEISQKLYQWKLMEAEKWPFFQEAQKLQAMHEKYPYKYPFRHH");
    appendValue(seq,"FPKKPLTPYFRFFMEKRAKYAKLHEPSNLKDILKHKSKYHELPEKKEKMYIQDFQEREKQFEFERNLARFREDHPD");
    appendValue(seq,"HIKKPLNAPFLYMKEAEMRANVVAESTLKESSAANIQILGRRWHALSREEQAKYYELARKERQLMQLYPGSARDNY");

    Graph<Alignment<StringSet<TSequence, Dependent<> > > aliG(seq);
    globalMsaAlignment(aliG, Blosum62(-1, -11));
    std::cout << aliG << std::endl;
    return 0;
}
```

And here is the output of this example program:
Alignment matrix:
0 . : . : . : . : . : .:
DPKKPRGKMSSYAFFVQTSREEHKKHHPDASVNFSEFKKCSERWTMSA
   |    |    |    |    |    |    |
RVKRPMNAFIWVSRDQRKMALENPRMNSEISQQLGVWMKTLAEEKWPFQEAQKLQAIIHREKYPNYKYRPRRRAKMLPK
   |    |    |    |    |    |    |
FPKKP---LTPYFRFFMEKRKAKLHP---EMSMLDLTKLKKSKKRLPE
   |    |    |    |    |    |    |
HIKKP---LNAFMLMKMRVQAEST---LKESAAINQILGRWHRHALSR

Assignment 1

Type Review

Objective Compute a multiple sequence alignments between the four protein sequences

- DPPKRPGKMSSYAFFVQTSREEHKKHHPDASVNFSEFKKCSERWTMSA
- RVKRPMNAFIWVSRDQRKMALENPRMNSEISQQLGVWMKTLAEEKWPFQEAQKLQAIIHREKYPNYKYRPRRRAKMLPK
- FPPKPLTPYFRFFMEKRKAKLHP---EMSMLDLTKLKKSKKRLPE
- HIKKP---LNAFMLMKMRVQAEST---LKESAAINQILGRWHRHALSR

using a Align object and the Blosum80 score matrix.

Repeat the above example using the Align data structure and the Blosum80 scoring matrix.

Solution After the usual includes, the Align object align is initialized and the four sequences are appended as rows.

```cpp
#include <iostream>
#include <seqan/align.h>
#include <seqan/graph_msa.h>

using namespace seqan;

int main()
{
    typedef String<AminoAcid> TSequence;
    Align<TSequence> align;
    resize(rows(align), 4);
    assignSource(row(align, 0), "DPKKPRGKMSSYAFFVQTSREEHKKKHHPDASVNFSEFKKCSERWTMSA");
    assignSource(row(align, 1), "RVKRPMNAFIWVSRDQRKMALENPRMNSEISQQLGVWMKTLAEEKWPFQEAQKLQAII=");
    assignSource(row(align, 2), "FPKKP---LTPYFRFFMEKRKAKLHP---EMSMLDLTKLKKSKKRLPE");
    assignSource(row(align, 3), "HIKKP---LNAFMLMKMRVQAEST---LKESAAINQILGRWHRHALSR");

    globalMsaAlignment(align, Blosum80(-1, -11));
    std::cout << align << std::endl;
}
```

Now the MSA is computed, using the Blosum80 matrix for scoring.
return 0;
}

And here is the output:

```
DPKPFRGKMSSYAFFVOTSREEHKKHPDASVNFEFSKKSERJWTMSA
RVKRP---MNAPIVQSDQRKMALENFRMR-NS-EISQQLGYYQKMLTE
FPKKP---LTPYFRFFMEKRAKYAKLHPEMS-NL-DLTKILSKYKELPE
HIKKP---LNAFMLYMNNVVAESTLKE-SA-AINQILGRKHALSR
```

```
KEKGFEDMAKAKARYEREMKTY---------------------IP--PKG---E
AEKWPFFQEAOKLOAMH-RE-K-----YP-------NYKYRPRLAKMLPK
KKKMYIQDFQREKQERFERNLAFREDHP-------DL--IQ--NAK--K
EEQAKYYYELARKERQLH-MQ-L-----YPGWSARDNYGKKRRKRE---K
```

2.1.12 Indices

**Learning Objective**  You will get an overview of the different kinds of indices in SeqAn and how they are used.

**Difficulty**  Average

**Duration**  1 h

**Prerequisites**  *Sequences*

**Indices in SeqAn**

Indices in SeqAn are substring indices, meaning that they allow efficient pattern queries in strings or sets of strings. In contrast to, e.g., online-search algorithms that search through the text in $O(n)$, substring indices find a pattern in sublinear time $o(n)$. 

2.1. Tutorial
You can find the following indices in SeqAn.

**IndexEsa** Extended Suffix Array [AKO04]

**IndexWotd** Lazy suffix tree [GKS03]

**IndexDFI** Deferred Frequency Index [WS08]

**IndexQGram** Q-gram index

**PizzaChiliIndex** An adapter for the Pizza & Chili index API

**FMIndex** [FM01]

### Index Construction

We will now show how we can create the different indices in SeqAn before we show how they are used for pattern search.

All the mentioned indices belong to the generic `Index` class. A SeqAn index needs two pieces of information: the type of the `String` or `StringSet` to be indexed and the index specialization, such as `IndexEsa` or `FMIndex`.

The following code snippet creates an enhanced suffix array index of a string of type `Dna5`.

```cpp
String<Dna5> genome = "ACGTACGTACGTN";
Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
```

In contrast, the next code snippet creates a FM index over a set of amino acid sequences:

```cpp
StringSet<String<AminoAcid> > protein;
appendValue(protein, "VXLAGZ");
appendValue(protein, "GKTVXL");
appendValue(protein, "XLZ");

Index<StringSet<String<AminoAcid> >, FMIndex> fmIndex(protein);
```

### Assignment 1

**Type** Review

**Objective** Copy the code below and

1. change it to build an `IndexEsa` over a string of type `Dna`,
2. add an `IndexEsa` over a `StringSet` of `Strings` of type `Dna`.

```cpp
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    String<char> text = "This is the first example";
    Index<String<char>, FMIndex<> > index(text);

    return 0;
}
```

**Solution**


```cpp
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    // One possible solution to the first sub assignment
    String<Dna> text = "ACGTTTGACAGCT";
    Index<String<Dna>, IndexEsa<> > index(text);

    // One possible solution to the second sub assignment
    StringSet<String<Dna> > stringSet;
    appendString(stringSet, "ACGTCATCAT");
    appendString(stringSet, "ACTTTG");
    appendString(stringSet, "CACCCCCCTATTT");

    Index<StringSet<String<Dna> >, IndexEsa<> > indexSet(stringSet);

    return 0;
}
```

## Index Based Pattern Search (Strings)

SeqAn provides two methods for searching for a pattern in index structures. One method uses iterators and is similar to traversing search trees or tries. The tutorial *Index Iterators* explains this method in more detail. In this section you will learn how to find a pattern with the *Finder* interface.

The *Finder* is an object that stores all necessary information for searching for a pattern using an index. The following line of code shows how the *Finder* is initialized.

```cpp
String<Dna5> genome = "ACGTACGTACGTN";
Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
Finder<Index<String<Dna5>, IndexEsa<> >, IndexEsa<> > esaFinder(esaIndex);
```

After initialization it is possible to use the *find* function in order to trigger a search for all occurrences of a given pattern in the underlying *String* or *StringSet*. In this example, we search for the pattern ACGT:

```cpp
String<Dna5> genome = "ACGTACGTACGTN";
Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
Finder<Index<String<Dna5>, IndexEsa<> >, IndexEsa<> > esaFinder(esaIndex);

find(esaFinder, "ACGT");
```

Calling the function *find* invokes the localization of all occurrences of a given pattern. It works by modifying pointers of the *Finder* to tables of the index. For example, the *Finder* of `esaIndex` stores two pointers, pointing to the first and last suffix array entry that stores an occurrence of the pattern.

The return value of the *find* function tells us whether or not a given pattern occurs in the text. Furthermore, if there are several instances of a pattern, consecutive calls of *find* will modify the *Finder* such that it points to the next occurrence after each call:

```cpp
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
```
The above code is not very useful, since we do not know the locations of the first, second or third pattern occurrence. The function position will help here. position called on a finder returns the location of the \( x \)th pattern, where \( x \) can be the first, second, or any other occurrence of the pattern.

```cpp
#include <seqan/sequence.h>
#include <seqan/index.h>
using namespace seqan;

int main()
{
    String<Dna5> genome = "ACGTACGTACGTN";
    Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
    Finder<Index<String<Dna5>, IndexEsa<> > > esaFinder(esaIndex);

    find(esaFinder, "ACGT"); // first occurrence of "ACGT"
    position(esaFinder); // -> 0
    find(esaFinder, "ACGT"); // second occurrence of "ACGT"
    position(esaFinder); // -> 4
    find(esaFinder, "ACGT"); // third occurrence of "ACGT"
    position(esaFinder); // -> 8
}
```

**Tip:** Indices in SeqAn are build on demand. That means that the index tables are not build when the constructor is called, but when we search for a pattern for the first time.

## Assignment 2

**Type** Application

**Objective** Write a small program that prints the locations of all occurrences of "TATAA" in "TTATTAAGCGTATAGCCCTATAAATATAA".

**Hints** Use the find function as the conditional instruction of a while loop.

**Solution**

```cpp
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    String<Dna5> genome = "TTATTAAGCGTATAGCCCTATAAATATAA";
    Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
    Finder<Index<String<Dna5>, IndexEsa<> > > esaFinder(esaIndex);

    find(esaFinder, "TATAA"); // first occurrence of "TATAA"
    position(esaFinder); // -> 0
    find(esaFinder, "TATAA"); // second occurrence of "TATAA"
    position(esaFinder); // -> 6
    find(esaFinder, "TATAA"); // third occurrence of "TATAA"
    position(esaFinder); // -> 10
    find(esaFinder, "TATAA"); // forth occurrence of "TATAA"
    position(esaFinder); // -> 14
}
```
```cpp
while(find(esaFinder, "TATAA"))
{
    std::cout << position(esaFinder) << std::endl;
}

return 0;
```

You might have noticed that we only applied the FMIndex and IndexEsa in the examples. The reason for this is that even though everything stated so far is true for the other indices as well, IndexWotd and IndexDfi are more useful when used with iterators as explained in the tutorial Index Iterators and the IndexQGram uses Shapes which is also explained in another tutorial.

One last remark is necessary.

**Important:** If you search for two different patterns with the same Finder object, you have to call the clear function of the finder between the search for the two patterns. Otherwise the behavior is undefined.

### Handling Multiple Sequences (StringSets)

The previous sections already described how an index of a set of strings can be instantiated. A character position of a StringSet can be one of the following:

1. A local position (default), i.e. a Pair (seqNo, seqOfs) where seqNo identifies the string within the StringSet and the seqOfs identifies the position within this string.
2. A global position, i.e. a single integer value between 0 and the sum of string lengths minus 1. This integer is the position in the gapless concatenation of all strings in the StringSet to a single string.

For indices, the meta-function SAValue determines, which position type (local or global) will be used for internal index tables (suffix array, q-gram array) and what type of position is returned by functions like position of a Finder. SAValue returns a Pair (local position) by default, but could be specialized to return an integer type (global position) for some applications. If you want to write algorithms for both variants you should use the functions posLocalize, posGlobalize, getSeqNo, and getSeqOffset.

### Storing and Loading

Storing and loading an index can be done with:

```cpp
const char *fileName = "/home/user/myindex";
save(index, fileName);
```

or

```cpp
const char *fileName = "/home/user/myindex";
open(index, fileName);
```

If you have built your q-gram index with variable shapes (i.e. SimpleShape GenericShape), you have to keep in mind that q or the shape is not stored or loaded. This must be done manually directly before or after loading with resize oder stringToShape.

A newly instantiated index is initially empty. If you assign a text to be indexed, solely the text fibre is set. All other fibres are empty and created on demand. Normally, a full created index should be saved to disk. Therefore, you have to create the required fibres explicitly by hand.
const char *fileName = "/home/user/myindex";
indexRequire(index, QGramSADir());
save(index, fileName);

For the IndexEsa index you could do:

const char *fileName = "/home/user/myindex";
indexRequire(index, EsaSA());
indexRequire(index, EsaLcp());
indexRequire(index, EsaChildtab()); // for TopDown iterators
indexRequire(index, EsaBwt()); // for (Super-)MaxRepeats iterators
save(index, fileName);

Indexes based on external strings, e.g. Index<String<Dna, External<> >, IndexEsa<> > or Index<String<Dna, MMap<> >, IndexEsa<> > cannot be saved, as they are persistent implicitly. The first thing after instantiating such an index should be associating it to a file with:

Index<String<Dna, External<> >, IndexEsa<> > index;
const char *fileName = "/home/user/myindex";
open(index, fileName);

The file association implies that any change on the index, e.g. fibre construction, is synchronized to disk. When instantiating and associating the index the next time, the index contains its previous state and all yet constructed fibres.

Reducing the memory consumption

One option is to change the data types used. This option to reduce the memory consumption has no drawback concerning running time but one has to make sure that the text to index does not exceed 4.29 billion characters. The critical observation is that each suffix array entry consumes 64 bit of memory per default where 32 bit would be sufficient if the text size is appropriate. In order to change the size type of the suffix array entry we simply have to overload the metafunction SAValue.

template<>
struct SAValue<String<Dna> >
{
    typedef unsigned Type;
}

If your text is a StringSet than SAValue will return a Pair that can be overloaded in the same way.

template<>
struct SAValue<StringSet<String<Dna> > >
{
    typedef Pair<unsigned, unsigned> Type;
}

The first type of the pair is used as the type for the index of a string in the string set. So if you only have a few strings you could save even more memory like this.

template<>
struct SAValue<StringSet<String<Dna> > >
{
    typedef Pair<unsigned char, unsigned> Type;
}
How To: Accessing Index Fibres Directly

See Accessing Index Fibres for more information.

2.1.13 Index Iterators

Learning Objective  You will know the different kinds of index indices and how to use them for searching.

Difficulty  Average

Duration  1.5 h

Prerequisites  Sequences, Iterators

Virtual String Tree Iterator

The search procedure of IndexEsa, IndexWotd, IndexDfi and FMIndex are suffix array based. This can be utilized in form of a common iterator interface. This common interface is the Virtual String Tree Iterator (VSTree Iterator) in SeqAn, which lets you access the IndexEsa, IndexWotd and IndexDfi as if using a suffix tree (Suffix Tree definition) and the FMIndex as if using a prefix trie.

In the first part of this tutorial we will concentrate on the TopDown Iterator which is one of the two index iterator specializations (besides the BottomUp Iterator). The second part will then deal with the DFS.

Top-Down Iteration

For index based pattern search or algorithms traversing only the upper parts of the suffix tree the TopDown Iterator or TopDown History Iterator is the best solution. Both provide the functions goDown and goRight to go down to the first child node or go to the next sibling. The TopDown History Iterator additionally provides goUp to go back to the parent node. The child nodes in IndexEsa indices are lexicographically sorted from first to last. For IndexWotd and IndexDfi indices this holds for all children except the first.

In the next example we want to use the TopDown Iterator to efficiently search a text for exact matches of a pattern. We therefore want to use goDown which has an overload to go down an edge beginning with a specific character.
Important: Note that the iterator traverses the complete edge. It does not stop after the first characters if the edge represents more than one character. This is true for all tree iterators. The only exception is the iterator of the FMIndex, which is a trie iterator.

First we create an index of the text "How much wood would a woodchuck chuck?"

```cpp
int main ()
{
    typedef Index<CharString> TIndex;
    TIndex index("How many wood would a woodchuck chuck?");
}
```

Afterwards we create the TopDown Iterator using the metafunction Iterator, which expects two arguments, the type of the container to be iterated and a specialization tag (see the VSTree Iterator hierarchy and the Iterators Tutorial for more details).

```cpp
Iterator< TIndex, TopDown<> >::Type it(index);
```

The main search can then be implemented using the functions repLength and representative. Since goDown might cover more than one character it is necessary to compare parts of the pattern against the representative of the iterator. The search can now be implemented as follows. The algorithm descends the suffix tree along edges beginning with the corresponding pattern character. In each step the unseen edge characters have to be verified.

```cpp
CharString pattern = "wood";
while (repLength(it) < length(pattern))
{
    // go down edge starting with the next pattern character
    if (!goDown(it, pattern[repLength(it)])) return 0;
    unsigned endPos = std::min((unsigned)repLength(it), (unsigned)length(pattern));
    // compare remaining edge characters with pattern
    std::cout << representative(it) << std::endl;
    if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
        infix(pattern, parentRepLength(it) + 1, endPos)) return 0;
}
```

If all pattern characters could successfully be compared we end in the topmost node who’s leaves point to text positions starting with the pattern. Thus, the suffixes represented by this node are the occurrences of our pattern and can be retrieved with getOccurrences.

```cpp
// if we get here the pattern was found
// output match positions
for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
    std::cout << getOccurrences(it)[i] << std::endl;
```

Program output:

```
F 0
wo 9
wood 22
```

Alternatively, we could have used goDown to go down the path of a pattern instead single characters:

```cpp
if (goDown(it, "wood"))
    for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
        std::cout << getOccurrences(it)[i] << std::endl;
```
Assignment 1

Type Review

Objective Copy the code into a demo program and replace the text with a string set containing the strings "How much", "wood would" and "a woodchuck chuck?".

Solution

```cpp
// FRAGMENT(includes)
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main ()
{
    StringSet<String<char>> text;
    appendValue(text, "How many");
    appendValue(text, " wood would");
    appendValue(text, " a woodchuck chuck?");

    typedef Index<StringSet<String<char>>> TIndex;
    TIndex index(text);
    Iterator< TIndex, TopDown<> >::Type it(index);

    CharString pattern = "wood";
    while (repLength(it) < length(pattern))
    {
        // go down edge starting with the next pattern character
        if (!goDown(it, pattern[repLength(it)])) return 0;

        unsigned endPos = _min(repLength(it), length(pattern));
        // compare remaining edge characters with pattern
        std::cout << representative(it) << std::endl;
        if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
            infix(pattern, parentRepLength(it) + 1, endPos)) return 0;
    }

    // if we get here the pattern was found
    // output match positions
    for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
        std::cout << getOccurrences(it)[i] << std::endl;

    return 0;
}
```

The difference is the format of the positions of the found occurrences. Here, we need a Pair to indicate the string within the StringSet and a position within the string.
Assignment 2

Type  Review

Objective  Write a program that traverses the nodes of the suffix tree of "tobeornottobe" in the order shown here:

At each node print the text of the edges from the root to the node. You may only use the functions goDown, goRight, goUp and isRoot to navigate and representative which returns the string that represents the node the iterator points to.

Hint

• Use a TopDown History Iterator.
• The code skeleton could look like this:

```cpp
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main ()
{
    typedef Index<CharString> TIndex;
    TIndex index("tobeornottobe");
    Iterator< TIndex, TopDown<ParentLinks<> > > ::Type it(index);
    do
    {
        ...
    } while (isRoot(it));

    return 0;
}
```

Solution  One iteration step of a preorder DFS can be described as follows:

• if possible, go down one node
• if not:
  – if possible, go to the next sibling
  – if not:
    * go up until it is possible to go to a next sibling
    * stop the whole iteration after reaching the root node

Thus, the DFS walk can be implemented in the following way:

```cpp
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main () {
    typedef Index<CharString> TIndex;
    TIndex index("tobeornottobe");
    Iterator< TIndex, TopDown<ParentLinks<> > >::Type it(index);

    do {
        std::cout << representative(it) << std::endl;
        if (!goDown(it) && !goRight(it))
            while (goUp(it) && !goRight(it)) ;
    } while (!isRoot(it));

    return 0;
}
```

Assignment 3

Type Review

Objective Modify the program to efficiently skip nodes with representatives longer than 3. Move the whole program into a template function whose argument specifies the index type and call this function twice, once for the `IndexEsa` and once for the `IndexWotd` index.

Solution We modify the DFS traversal to skip the descent if we walk into a node whose representative is longer than 3. We then proceed to the right and up as long as the representative is longer than 3.

```cpp
template < typename TIndexSpec >
void constrainedDFS () {
    typedef Index<CharString, TIndexSpec> TIndex;
    TIndex index("tobeornottobe");
    typename Iterator< TIndex, TopDown<ParentLinks<> > >::Type it(index);

    do {
        std::cout << representative(it) << std::endl;
        if (!goDown(it) || repLength(it) > 3)
            do {
                if (!goRight(it))
                    while (goUp(it) && !goRight(it)) ;
            } while (repLength(it) > 3);
    } while (!isRoot(it));

    std::cout << std::endl;
}
```
int main ()
{
    constrainedDFS< IndexEsa<> > ();
    constrainedDFS< IndexWotd<> > ();
    return 0;
}

Depth-First Search

The tree traversal in assignment 2 is equal to a the tree traversal in a full depth-first search (dfs) over all suffix tree nodes beginning either in the root (preorder dfs) or in a leaf node (postorder dfs). A preorder traversal (Preorder DFS) halts in a node when visiting it for the first time whereas a postorder traversal (Postorder DFS) halts when visiting a node for the last time. The following two figures give an example in which order the tree nodes are visited.

Figure 2.1: Preorder DFS

Since these traversals are frequently needed SeqAn provides special iterators which will we describe next.

We want to construct the suffix tree of the string “abracadabra” and output the substrings represented by tree nodes in preorder dfs. In order to do so, we create the string “abracadabra” and an index specialized with the type of this string.
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

The Iterator metafunction expects two arguments, the type of the container to be iterated and a specialization tag, as described earlier. In this example we chose a TopDown History Iterator whose signature in the second template argument is `TopDown< ParentLinks<Preorder> >`.

```cpp
int main ()
{
    String<char> myString = "abracadabra";

    typedef Index<String<char>> TMyIndex;
    TMyIndex myIndex(myString);

As all DFS suffix tree iterators implement the VSTree Iterator, they can be used via `goNext`, `atEnd`, etc.

    Iterator<TMyIndex, TopDown< ParentLinks<Preorder> > ::Type myIterator(myIndex);

    while (!atEnd(myIterator))
    {
        std::cout << representative(myIterator) << std::endl;
        ++myIterator;
    }

    return 0;
}
```

Program output:

```
a
abra
abracadabra
```
Tip: There are currently 2 iterators in SeqAn supporting a DFS search:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Preorder</th>
<th>Postorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottomUpIterator</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>TopDownHistoryIterator</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

If solely a postorder traversal is needed the BottomUp Iterator should be preferred as it is more memory efficient. Please note that the BottomUp Iterator is only applicable to IndexEsa indices.

Tip: A relaxed suffix tree (see Suffix Tree) is a suffix tree after removing the $ characters and empty edges. For some bottom-up algorithms it would be better not to remove empty edges and to have a one-to-one relationship between leaves and suffixes. In that cases you can use the tags PreorderEmptyEdges or PostorderEmptyEdges instead of Preorder or Postorder or EmptyEdges for the TopDown Iterator.

Note that the goNext is very handy as it simplifies the tree traversal in assignment 2 greatly.

Assignment 4

Type Review

Objective Write a program that constructs an index of the StringSet “toboeornottobe”, “thebeecolomb”, “beingjohnmalkovich” and outputs the strings corresponding to suffix tree nodes in postorder DFS.

Solution First we have to create a StringSet of CharString (shortcut for String<char>) and append the 3 strings to it. This could also be done by using resize and then assigning the members with operator[]. The first template argument of the index class has to be adapted and is now a StringSet.

```c++
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main ()
{
    StringSet<CharString> myStringSet;
    appendValue(myStringSet, "toboeornottobe");
    appendValue(myStringSet, "thebeecolomb");
    appendValue(myStringSet, "beingjohnmalkovich");

    typedef Index<StringSet<CharString>> TMyIndex;
    TMyIndex myIndex(myStringSet);

    To switch to postorder DFS we have two change the specialization tag of ParentLinks from Preorder to Postorder. Please note that the TopDownHistoryIterator always starts in the root node, which is the last postorder DFS node. Therefore, the iterator has to be set explicitly to the first DFS node via goBegin.
Iterator< TMyIndex, TopDown< ParentLinks<Postorder> > >::Type myIterator(myIndex);

// Top-down iterators start in the root node which is not the first node of a
// postorder DFS. Thus we have to manually go to the DFS start with goBegin
goBegin(myIterator);
while (!atEnd(myIterator))
{
    std::cout << representative(myIterator) << std::endl;
    ++myIterator;
}

Alternatively to a TopDownHistoryIterator you also could have used a BottomUpIterator with the same result. The BottomUpIterator automatically starts in the first DFS node as it supports no random access.

Iterator< TMyIndex, BottomUp<> >::Type myIterator2(myIndex);

while (!atEnd(myIterator2))
{
    std::cout << representative(myIterator2) << std::endl;
    ++myIterator2;
}

return 0;
}

Program output:

alkovich
beeonthecomb
beingjohnmalkovich
beornottobe
be
b
ch
comb
c
ebeeonthecomb
ecomb
eeonthecomb
eingjohnmalkovich
eonthecomb
eornottobe
eo
e
gjohnmalkovich
hebeonthecomb
hecomb
he
hnmaalkovich
h
ich
ingjohnmalkovich
i
johnmalkovich
kovich
lkovich
malkovich
mb
m

2.1. Tutorial
As a last assignment let's try out one of the specialised iterators, which you can find at the bottom of this page. Look there for the specialisation which iterates over all maximal unique matches (MUMS).

**Assignment 5**

**Type** Review

**Objective** Write a program that outputs all maximal unique matches (MUMs) between "CDFGHC" and "CDEFGAHC".

**Solution** Again, we start to create a `StringSet` of `CharString` and append the 2 strings.

```cpp
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main ()
{
    StringSet<CharString> myStringSet;
    appendValue(myStringSet, "CDFGHC");
    appendValue(myStringSet, "CDEFGAHC");

    typedef Index< StringSet<CharString> > TMyIndex;
    TMyIndex myIndex(myStringSet);

    Iterator< TMyIndex, Mums >::Type myIterator(myIndex);
    while (!atEnd(myIterator))
```

After that we simply use the predefined iterator for searching MUMs, the *MumsIterator*. Its constructor expects the index and optionally a minimum MUM length as a second parameter. The set of all MUMs can be represented by a subset of suffix tree nodes. The iterator will halt in every node that is a MUM of the minimum length. The corresponding match is the node’s representative.

```cpp
    // The MUMs
    while (!atEnd(myIterator))
    {
        // Output the MUM
    }
```
Accessing Suffix Tree Nodes

In the previous subsection we have seen how to walk through a suffix tree. We now want to know what can be done with a suffix tree iterator. As all iterators are specializations of the general VSTree Iterator class, they inherit all of its functions. There are various functions to access the node the iterator points at (some we have already seen), so we concentrate on the most important ones.

representative returns the substring that represents the current node, i.e. the concatenation of substrings on the path from the root to the current node

getOccurrence returns a position where the representative occurs in the text

getOccurrences returns a string of all positions where the representative occurs in the text

isRightTerminal tests if the representative is a suffix in the text (corresponds to the shaded nodes in the Suffix Tree figures)

isLeaf tests if the current node is a tree leaf

parentEdgeLabel returns the substring that represents the edge from the current node to its parent (only TopDown-History Iterator)

Important: There is a difference between the functions isLeaf and isRightTerminal. In a relaxed suffix tree (see Suffix Tree) a leaf is always a suffix, but not vice versa, as there can be internal nodes a suffix ends in. For them isLeaf returns false and isRightTerminal returns true.

Property Maps

Some algorithms require to store auxiliary information (e.g. weights, scores) to the nodes of a suffix tree. To attain this goal SeqAn provides so-called property maps, simple Strings of a property type. Before storing a property value, these strings must first be resized with resizeVertexMap. The property value can then be assigned or retrieved via assignProperty, getProperty, or property. It is recommended to call resizeVertexMap prior to every call of assignProperty to ensure that the property map has sufficient size. The following example iterates over all nodes in preorder dfs and recursively assigns the node depth to each node. First we create a String of int to store the node depth for each suffix tree node.

```c++
int main ()
{
    String<char> myString = "abracadabra";

    typedef Index< String<char>, IndexWord<> > TMyIndex;
    TMyIndex myIndex(myString);
    String<int> propMap;
```
The main loop iterates over all nodes in preorder DFS, i.e. parents are visited prior children. The node depth for the root node is 0 and for all other nodes it is the parent node depth increased by 1. The functions assignProperty, getProperty and property must be called with a VertexDescriptor. The vertex descriptor of the iterator node is returned by value and the descriptor of the parent node is returned by nodeUp.

```cpp
Iterator< TMyIndex, TopDown< ParentLinks<Preorder> > >::Type myIterator(myIndex);

int depth;
while (!atEnd(myIterator))
{
    if (isRoot(myIterator))
        depth = 0;
    else
        depth = getProperty(propMap, nodeUp(myIterator)) + 1;

    resizeVertexMap(myIndex, propMap);
    assignProperty(propMap, value(myIterator), depth);

    ++myIterator;
}
```

At the end we again iterate over all nodes and output the calculated node depth.

```cpp
goBegin(myIterator);
while (!atEnd(myIterator))
{
    std::cout << getProperty(propMap, value(myIterator)) << '\t' << representative(myIterator) << std::endl;
    ++myIterator;
}
```

Program output:

```
0
  1  a
  2  abra
  3  abracadabra
  2  acadabra
  2  adabra
  1  bra
  2  bracadabra
  1  cadabra
  1  dabra
  1  ra
  2  racadabra
```

**Tip:** In SeqAn there is already a function nodeDepth defined to return the node depth.

**Additional iterators**

By now, we know the following iterators ($n =$ text size, $\sigma =$ alphabet size, $d =$ tree depth):
### Iterator specialization

<table>
<thead>
<tr>
<th>Iterator specialization</th>
<th>Description</th>
<th>Space</th>
<th>Index tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottomUpIterator</td>
<td>postorder dfs</td>
<td>(O(d))</td>
<td>SA, LCP</td>
</tr>
<tr>
<td>TopDownIterator</td>
<td>can go down and go right</td>
<td>(O(1))</td>
<td>SA, Lcp, Childtab</td>
</tr>
<tr>
<td>TopDownHistoryIterator</td>
<td>can also go up, preorder/postorder dfs</td>
<td>(O(d))</td>
<td>SA, Lcp, Childtab</td>
</tr>
</tbody>
</table>

Besides the iterators described above, there are some application-specific iterators in SeqAn:

<table>
<thead>
<tr>
<th>Iterator specialization</th>
<th>Description</th>
<th>Space</th>
<th>Index tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxRepeatsIterator</td>
<td>maximal repeats</td>
<td>(O(n))</td>
<td>SA, Lcp, Bwt</td>
</tr>
<tr>
<td>SuperMaxRepeatsIterator</td>
<td>supermaximal repeats</td>
<td>(O(d + \sigma))</td>
<td>SA, Lcp, Childtab, Bwt</td>
</tr>
<tr>
<td>SuperMaxRepeatsFastIterator</td>
<td>supermaximal repeats (optimized for ESA)</td>
<td>(O(\sigma))</td>
<td>SA, Lcp, Bwt</td>
</tr>
<tr>
<td>MumsIterator</td>
<td>maximal unique matches</td>
<td>(O(d))</td>
<td>SA, Lcp, Bwt</td>
</tr>
<tr>
<td>MultiMemsIterator</td>
<td>multiple maximal exact matches (w.i.p.)</td>
<td>(O(n))</td>
<td>SA, Lcp, Bwt</td>
</tr>
</tbody>
</table>

Given a string \(s\) a repeat is a substring \(r\) that occurs at 2 different positions \(i\) and \(j\) in \(s\). The repeat can also be identified by the triple \((i, j, |r|)\). A maximal repeat is a repeat that cannot be extended to the left or to the right, i.e. \(s[i-1]s[j-1]\) and \(s[i+|r|]s[j+|r|]\). A supermaximal repeat \(r\) is a maximal repeat that is not part of another repeat. Given a set of strings \(s_1, ..., s_m\) a MultiMEM (multiple maximal exact match) is a substring \(r\) that occurs in each sequence \(s_i\) at least once and cannot be extended to the left or to the right. A MUM (maximal unique match) is a MultiMEM that occurs exactly once in each sequence. The following examples demonstrate the usage of these iterators:

- Demo Maximal Unique Matches
- Demo Supermaximal Repeats
- Demo Maximal Repeats

### 2.1.14 Q-gram Index

**Learning Objective** You will know the features of the q-gram Index, how it can be used for searching and how to access the different fibres.

**Difficulty** Average

**Duration** 1 h

**Prerequisites** Sequences, Iterators

#### The Q-gram Index

A q-gram index can be used to efficiently retrieve all occurrences of a certain q-gram in the text. It consists of various tables, called fibres (see Accessing Index Fibres), to retrieve q-gram positions, q-gram counts, etc. However, it has...
no support for suffix tree iterators. A q-gram index must be specialized with a Shape type. A Shape defines q, the number of characters in a q-gram and possibly gaps between these characters. There are different specializations of Shape available:

<table>
<thead>
<tr>
<th>Specialization</th>
<th>Modifiable</th>
<th>Number of Gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UngappedShape</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>SimpleShape</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>OneGappedShape</td>
<td>+</td>
<td>0/1</td>
</tr>
<tr>
<td>GappedShape</td>
<td>-</td>
<td>any</td>
</tr>
<tr>
<td>GenericShape</td>
<td>+</td>
<td>any</td>
</tr>
</tbody>
</table>

- - fixed at compile time, + can be changed at runtime

Each shape evaluates a gapped or ungapped sequence of q characters to a hash value by the Functions hash, hashNext, etc. For example, the shape 1101 represents a 3-gram with one gap of length 1. This shape overlayed with the Dna text "GATTACA" at the third position corresponds to "TT-C". The function hash converts this 3-gram into $61 = (3 \cdot 4 + 3 \cdot 4 + 1 \cdot 4$ is the alphabet size in this example (see ValueSize).

With hash and hashNext, we can compute the hash values of arbitrary / adjacent q-grams and a loop that outputs the hash values of all overlapping ungapped 3-grams could look as follows:

```cpp
dna_string text = "AAAACACAGTTTGA";
shape<Dna, UngappedShape<3> > myShape;
std::cout << hash(myShape, begin(text)) << '	';
for (unsigned i = 1; i < length(text) - length(myShape) + 1; ++i)
    std::cout << hashNext(myShape, begin(text) + i) << '	';
```

Note that the shape not only stores the length and gaps of a q-gram shape but also stores the hash value returned by the last hash/hashNext call. This hash value can be retrieved by calling value on the shape. However, one drawback of the example loop above is that the first hash value must be computed with hash while the hash values of the following overlapping q-grams can more efficiently be computed by hashNext. This complicates the structure of algorithms that need to iterate all hash values, as they have to handle this first hash differently. As a remedy, the hashInit function can be used first and then hashNext on the first and all following text positions in the same way:

```cpp
hashInit(myShape, begin(text));
for (unsigned i = 0; i < length(text) - length(myShape) + 1; ++i)
    std::cout << hashNext(myShape, begin(text) + i) << '	';
```

The q-gram index offers different functions to search or count occurrences of q-grams in an indexed text, see getOccurrences, countOccurrences. A q-gram index over a StringSet stores occurrence positions in the same way as the ESA index and in the same fibre (FibreSA). If only the number of q-grams per sequence are needed the QGramCounts and QGramCountsDir fibres can be used. They store pairs (seqNo, count), count>0, for each q-gram that occurs counts times in sequence number seqNo.

To efficiently retrieve all occurrence positions or all pairs (seqNo, count) for a given q-gram, these positions or pairs are stored in contiguous blocks (in QGramSA, QGramCounts fibres), called buckets. The begin position of bucket i is stored in directory fibres (QGramDir, QGramCountsDir) at position i, the end position is the begin positions of the bucket i+1. The default implementation of the IndexQGram index maps q-gram hash values 1-to-1 to bucket numbers. For large q or large alphabets the Open Adressing QGram Index can be more appropriate as its directories are additionally bound by the text length. This is realized by a non-trivial mapping from q-gram hashes to bucket numbers that requires an additional fibre (QGramBucketMap).

For more details on q-gram index fibres see Accessing Index Fibres or QGram Index Fibres.
Example

We want to construct the q-gram index of the string "CATGATTACATA" and output the occurrences of the ungapped 3-gram "CAT". As 3 is fixed at compile-time and the shape has no gaps we can use a UngappedShape which is the first template argument of IndexQGram, the second template argument of Index. Next we create the string "CATGATTACATA" and specialize the first index template argument with the type of this string. The string can be given to the index constructor.

```cpp
int main ()
{
    typedef Index<DnaString, IndexQGram< UngappedShape<3> > > TIndex;
    TIndex index("CATGATTACATA");
}
```

To get all occurrences of a q-gram, we first have to hash it with a shape of the same type as the index shape (we can even use the index shape returned by indexShape). The hash value returned by hash or hashNext is also stored in the shape and is used by the function getOccurrences to retrieve all occurrences of our 3-gram.

```cpp
hash(indexShape(index), "CAT");
for (unsigned i = 0; i < length(getOccurrences(index, indexShape(index))); ++i)
    std::cout << getOccurrences(index, indexShape(index))[i] << std::endl;
    return 0;
}
```

Program output:

```
0
8
```

Assignment 1

Type Review

Objective Write a program that outputs all occurrences of the gapped q-gram “AT-A” in “CATGATTACATA”.

Solution Before we can create a DnaString index of “CATGATTACATA”, we have to choose an appropriate Shape. Because our shape 1101 is known at compile-time and contains only one gap we could choose OneGappedShape, GappedShape, or GenericShape (see the commented-out code). Although the GenericShape could be used for every possible shape, it is a good idea to choose a Shape with restrictions as its hash functions are more efficient in general.

```cpp
int main ()
{
    Index<DnaString, IndexQGram<OneGappedShape> > index("CATGATTACATA");
    stringToShape(indexShape(index), "1101");
}
```

Please note that the Shape object that corresponds to the IndexQGram index is empty initially and has to be set by stringToShape or resize. This initialization is not necessary for Shape that are defined at compile-time, i.e. UngappedShape and GappedShape. To search for “AT-A” we first have to hash it with the index shape or any other Shape with the same bitmap. The we can use getOccurrences to output all matches.

```cpp
hash(indexShape(index), "ATCA");
for (unsigned i = 0; i < length(getOccurrences(index, indexShape(index))); ++i)
    std::cout << getOccurrences(index, indexShape(index))[i] << std::endl;
    return 0;
}
```
SeqAn Manual, Release 1.4.2

Tip: Instead of length(getOccurrences(...)) we could have used countOccurrences. But beware
that countOccurrences requires only the QGram_Dir fibre, whereas getOccurrences requires both QGram_Dir
and QGram_SA, see Accessing Index Fibres. Because QGram_SA can be much more efficiently constructed
during the construction of QGram_Dir, QGram_Dir would be constructed twice.
Program output:
1
4

Assignment 2

Type Review
Objective Create and output a matrix M where M(i,j) is the number of common ungapped 5-grams between sequence
i and sequence j for 3 random Dna sequences, each not longer than 200 characters. Optional: Run the matrix
calculation twice, once for an IndexQGram and once for an Open Adressing QGram Index and output the
directory sizes (QGram_Dir, QGram_CountsDir fibre).
Hint A common q-gram that occurs 𝑎 times in one and 𝑏 times in the other sequence counts for min(𝑎, 𝑏).
Solution For generating random numbers we use the MersenneTwisterRng which is a specialization of the random
number generator class Rng. The random numbers returned by pickRandomNumber are arbitrary unsigned
int values which we downscale to values between 0 and 3 and convert into Dna characters. The 3 generated
strings are of random length and appended to a StringSet. The main algorithmus is encapsulated in a template
function qgramCounting to easily switch between the two IndexQGram specializations.
int main ()
{
// for the sake of reproducibility
Rng<MersenneTwister> rng;
// create StringSet of 3 random sequences
StringSet<DnaString> stringSet;
reserve(stringSet, 3);
for (int seqNo = 0; seqNo < 3; ++seqNo)
{
DnaString tmp;
int len = pickRandomNumber(rng) % 100 + 10;
for (int i = 0; i < len; ++i)
appendValue(tmp, Dna(pickRandomNumber(rng) % 4));
appendValue(stringSet, tmp);
std::cout << ">Seq" << seqNo << std::endl << tmp << std::endl;
}
qgramCounting(stringSet, IndexQGram<UngappedShape<5> >());
qgramCounting(stringSet, IndexQGram<UngappedShape<5>, OpenAddressing>());
return 0;
}

The main function expects the StringSet and the Index specialization as a tag. First, we define lots of types we
need to iterate and access the fibres directly. We then notify the index about the fibres we require. For storing
the common q-grams we use a 2-dimensional Matrix object whose lengths have to be set with setLength for
each dimension. The matrix is initialized with zeros by resize.

148

Chapter 2. API Documentation


template <typename TStringSet, typename TIndexSpec>
void qgramCounting(TStringSet &set, TIndexSpec)
{
    typedef Index<TStringSet, TIndexSpec> TIndex;
    typedef typename Fibre<TIndex, QGramCounts>::Type TCounts;
    typedef typename Fibre<TIndex, QGramCountsDir>::Type TCountsDir;
    typedef typename Value<TCountsDir>::Type TDirValue;
    typedef typename Iterator<TCounts, Standard>::Type TIterCounts;
    typedef typename Iterator<TCountsDir, Standard>::Type TIterCountsDir;

    TIndex index(set);
    indexRequire(index, QGramCounts());

    // initialize distance matrix
    int seqNum = countSequences(index);
    Matrix<int, 2> distMat;
    setLength(distMat, 0, seqNum);
    setLength(distMat, 1, seqNum);
    resize(distMat, 0);

    std::cout << std::endl << "Length of the CountsDir fibre: " << length(indexCountsDir(index)) << std::endl;
    TIterCountsDir itCountsDir = begin(indexCountsDir(index), Standard());
    TIterCountsDir itCountsDirEnd = end(indexCountsDir(index), Standard());
    TIterCounts itCountsBegin = begin(indexCounts(index), Standard());

    The main part of the function iterates over the CountsDir fibre. Each entry in this directory represents a q-gram bucket, a contiguous interval in the Counts fibre storing for every sequence the q-gram occurs in the number of occurrences in pairs (seqNo,count). The interval begin of each bucket is stored in the directory and the interval end is the begin of the next bucket. So the inner loops iterate over all non-empty buckets and two pairs (seqNo1,count1) and (seqNo2,count2) indicate that seqNo1 and seqNo2 have a common q-gram. At the end the matrix can simply be output by shifting it to the cout stream.

    // for each bucket count common q-grams for each sequence pair
    TDirValue bucketBegin = *itCountsDir;
    for(++itCountsDir; itCountsDir != itCountsDirEnd; ++itCountsDir)
    {
        TDirValue bucketEnd = *itCountsDir;

        // q-gram must occur in at least 2 different sequences
        if (bucketBegin != bucketEnd)
        {
            TIterCounts itA = itCountsBegin + bucketBegin;
            TIterCounts itEnd = itCountsBegin + bucketEnd;
            for(; itA != itEnd; ++itA)
            {
                TIterCounts itB = itA; itB != itEnd; ++itB)
                    distMat((itA).i1, (itB).i1) += _min((itA).i2, (itB).i2);
                };
            
            bucketBegin = bucketEnd;
        }

        std::cout << std::endl << "Common 5-mers for Seq_i, Seq_j" << std::endl;
        std::cout << distMat;
    }

Please note that the open addressing q-gram index directories are smaller than the IndexQGram index directories.

Program output:
>Seq0
TCATTTTCTCGATGAAAGCGTGACCCCACATATGACTTTGTACTCCCT
>Seq1
TGATTGTTGAAACCGATACCGTACCTGTGTTAGTGAGAATTATTAAAATGATATGATATCCACCCCTATCGAGACTGCG
>Seq2
CCTAGGTGGGCTCGGCAACGAGACTGCG
Length of the CountsDir fibre: 1025

Common 5-mers for Seq_i, Seq_j
| 50 | 4 | 0 |
| 0  | 86| 5 |
| 0  | 0 | 25 |

Length of the CountsDir fibre: 259

Common 5-mers for Seq_i, Seq_j
| 50 | 4 | 0 |
| 0  | 86| 5 |
| 0  | 0 | 25 |

Contents
- Pattern Matching
  - Overview
  - Online Search
    * Exact Search
      · Assignment 1
    * Approximate Search
      · Assignment 2
  - Index Search
    * Exact Search
      · Assignment 3
    * Approximate Filtration

2.1.15 Pattern Matching

Learning Objective In this tutorial you will learn how to use the SeqAn classes finder and pattern to search a known pattern in a string or StringSet.

Difficulty Average

Duration 40 min

Prerequisites Sequences, Indices

Pattern matching is about searching a known string or StringSet (needle) in another string or StringSet (haystack). This tutorial will introduce you into the SeqAn classes finder and pattern. It will demonstrate how to use the specializations of the class finder to perform either an online search or an index based search. And you will learn how to specify the search algorithm, which can be exact or approximate.
Overview

In the case of approximate searching errors are allowed, which are either only mismatches or also indels. Additionally there are filtration algorithms which return potential matches, i.e. haystack segments possibly containing a pattern match. All searching is done by calling the function `find`, which takes at least two arguments:

1. A Finder that stores all necessary information about the haystack and the last found position of the needle within the haystack.

2. A Pattern that stores all information about the needle. Some variants of `find` support further arguments. The Finder and Pattern classes expect the underlying haystack and needle types as first template arguments. In addition, a second template argument specifies the search algorithm.

Each call of `find` finds only one match (or potential match) of the needle within the haystack. The Finder can be asked for the begin and end position of the last found match. The Pattern can be asked for the number of the found sequence if the needle is a StringSet. Subsequent calls of `find` can be used to find more occurrences of the needle, until no more occurrences can be found and `find` returns `false`.

In general, search algorithms can be divided into algorithms that preprocess the needle (online search) or preprocess the haystack (index search).

Online Search

For all online search algorithms, the Finder is an iterator that scans over the haystack. The Pattern is a search algorithm dependent data structure preprocessed from the needle. The second template argument of the Pattern selects the search algorithm.

Exact Search

The following code snippet illustrates the usage of online search algorithms in SeqAn using the example of the Horspool algorithm [Hor80]. We begin by creating two strings of type `char` containing the haystack and the needle.

```cpp
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    CharString needle = "mo";

    Finder<CharString> finder(haystack);
    Pattern<CharString, Horspool> pattern(needle);
    while (find(finder, pattern))
        std::cout << '[' << beginPosition(finder) << ', ' << endPosition(finder) << ')" << std::endl;

    return 0;
}
```

Program output:
Currently the following exact online algorithms for searching a single sequence are implemented in SeqAn:

**Simple**  Brute force algorithm

**Horspool**  [Hor80]

**Bfam**  Backward Factor Automaton Matching

**BndmAlgo**  Backward Nondeterministic DAWG Matching

**ShiftAnd**  Exact string matching using bit parallelism

**ShiftOr**  Exact string matching using bit parallelism

... and for multiple sequences:

**WuManber**  Extension of Horspool.

**MultiBfam**  Multiple version of Bfam, uses an automaton of reversed needles.

**SetHorspool**  Another extension of Horspool using a trie of reversed needles.

**AhoCorasick**  [AC75]

**MultiShiftAnd**  Extension of ShiftAnd, should only be used if the sum of needle lengths doesn’t exceed the machine word size.

**Assignment 1**

**Type**  Review

**Objective**  Use the given code example from below. Extend the code to search the given haystack simultaneously for “mo”, “send” and “more”. For every match output the begin and end position in the haystack and which needle has been found.

**Hint**  Online search algorithms for multiple sequences simply expect needles of type `String<String<...>>`.

```cpp
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    String<CharString> needles;
    appendValue(needles, "mo");
    appendValue(needles, "send");
    appendValue(needles, "more");

    return 0;
}
```

You can use the specialization WuManber.

**Solution**  Click more... to see the solution.
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    String<CharString> needles;
    appendValue(needles, "mo");
    appendValue(needles, "send");
    appendValue(needles, "more");

    Finder<CharString> finder(haystack);
    Pattern<String<CharString>, WuManber> pattern(needles);
    while(find(finder, pattern))
    {
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << ') 	
        std::cout << position(pattern) << ' ' 	' infix(finder) << std::endl;
    }
    return 0;
}

We use a Pattern specialized with the WuManber algorithm for the search and initialize it with our needles string. For every match found by find we output the begin and end position and the match region in the haystack as well as the index of the found needle which is returned by position(pattern).

[2,4) 0 mo
[7,11) 1 send
[12,14) 0 mo
[12,16) 2 more
[17,19) 0 mo

Approximate Search

The approximate search can be used to find segments in the haystack that are similar to a needle allowing errors, such as mismatches or indels. Note that if only mismatches are allowed, the difference of the end and begin position of a match is the length of the found needle. However, in the case of indels this difference may vary and is only a rough estimate for the length. Therefore, to find a begin position for a certain end position the findBegin interface should be used. The usage is similar to find and is shown in the next example. We want to find all semi-global alignments of a needle "more" with a SimpleScore of at least -2 using the scoring scheme (0,-2,-1) (match,mismatch,gap).

Again, we create haystack and needle strings first:

#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    CharString needle = "more";

    We then create Finder and Pattern objects of these strings and choose DPSearch as the specialization in the second template argument of Pattern. DPSearch expects the scoring function as the first template argument which is SimpleScore in our example. The pattern is constructed using the needle as a template and our scoring object is initialized
with the appropriate scores for match, mismatch and gap. As in the previous example, the main iteration uses find to iterate over all end positions with a minimum best score of -2. If such a semi-global alignment end position is found the begin position is searched via findBegin. Please note that we have to set the minimum score to the score of the match found (getScore) in order to find the begin of a best match. We then output all begin and end positions and the corresponding haystack segment for each match found.

Finder<CharString> finder(haystack);
Pattern<CharString, DPSearch<SimpleScore>> pattern(needle, SimpleScore(0, -2, -1));
while (find(finder, pattern, -2))
  while (findBegin(finder, pattern, getScore(pattern)))
    std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << '"

return 0;
}

Program output:
[2,4) mo
[12,14) mo
[12,15) mor
[12,16) more
[12,17) more
[12,18) more m
[17,19) mo
[17,21) mone

The following specializations are available:

Specialization DPSearch Dynamic programming algorithm for many kinds of scoring scheme
Specialization Myers [Mye99], [Ukk85]
Specialization Pex [BYN99]
Specialization AbndmAlgo Approximate Backward Nondeterministic DAWG Matching, adaption of AbndmAlgo

Assignment 2

Type Application

Objective Use the example from above. Modify the code to search with the Myers algorithm for matches of "more" with an edit distance of at most 2.

Solution Click more... to see the solution.

#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
  CharString haystack = "Simon, send more money!";
  CharString needle = "more";

  Finder<CharString> finder(haystack);
  Pattern<CharString, Myers<> > pattern(needle);
  while (find(finder, pattern, -2))
    while (findBegin(finder, pattern, getScore(pattern)))
      std::cout << '(' << beginPosition(finder) << ',', << endPosition(finder) << 

return 0;
}

We again set the needle to "more". We then change the specialization tag of the Pattern to Myers with default arguments. As Myers algorithm is only applicable to edit distance searches it cannot be specialized or initialized with a scoring scheme. In SeqAn, edit distance corresponds to the scoring scheme (0,-1,-1) (match, mismatch, gap) and an edit distance of 2 corresponds to a minimum score of -2 given to the find function.

The program’s output is as follows.

\[
\begin{align*}
&[2,4) \text{ mo} \\
&[2,5) \text{ mon} \\
&[2,6) \text{ mon,} \\
&[12,14) \text{ mo} \\
&[12,15) \text{ mor} \\
&[12,16) \text{ more} \\
&[12,17) \text{ more} \\
&[12,18) \text{ more m} \\
&[17,19) \text{ mo} \\
&[17,20) \text{ mon} \\
&[17,21) \text{ mone} \\
&[17,22) \text{ money}
\end{align*}
\]

Index Search

Exact Search

For the index based search the Finder needs to be specialized with an Index of the haystack in the first template argument. The index itself requires two template arguments, the haystack type and a index specialization. In contrast, since the needle is not preprocessed the second template argument of the Pattern has to be omitted. The following source illustrates the usage of an index based search in SeqAn using the example of the IndexEsa index (an enhanced suffix array index). This is the default index specialization if no second template argument for the index is given. We begin to create an index object of our haystack "toboeornottobe" and a needle "be".

\[
\begin{align*}
\text{int main()} \\
&\{ \\
&\quad \text{Index<CharString> index("toboeornottobe");} \\
&\quad \text{CharString needle = "be";} \\
&\quad \text{Finder<Index<CharString> > finder(index);} \\
&\}
\]

We proceed to create a Pattern of the needle and conduct the search in the usual way.

\[
\begin{align*}
&\text{Pattern<CharString> pattern(needle);} \\
&\text{while (find(finder, pattern))} \\
&\quad \text{std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << "\t" << infix(finder) << std::endl;}
\end{align*}
\]

Instead of creating and using a pattern solely storing the needle we can pass the needle directly to find. Please note that an Index based Finder has to be reset with clear before conducting another search.

\[
\begin{align*}
&\text{clear(finder);} \\
&\text{while (find(finder, "be"))} \\
&\quad \text{std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << "\t" << std::endl;}
\end{align*}
\]

Program output:
All indices also support StringSet texts and can therefore be used to search multiple haystacks as the following example shows. We simply exchange the CharString of the haystack with a StringSet of CharString and append some strings to it.

```cpp
int main()
{
    typedef StringSet<CharString> THaystacks;
    THaystacks haystacks;
    appendValue(haystacks, "tobeornottobe");
    appendValue(haystacks, "thebeenonthecomb");
    appendValue(haystacks, "beingjohnmalkovich");

    Index<THaystacks> index(haystacks);
    Finder<Index<THaystacks>> finder(haystacks);

    The rest of the program remains unchanged.
    clear(finder);
    while (find(finder, "be"))
    {
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << "\t" << infix(finder) << std::endl;
    }
    return 0;
}
```

The following index specializations support the Finder interface as described above.

**Specialization IndexEsa** Enhanced suffix array based index. Supports arbitrary needles.

**Specialization IndexQGram** Q-gram index. Needle mustn’t exceed the size of the q-gram.

**Specialization Open Adressing QGram Index** Q-gram index with open addressing. Supports larger q-grams. Needle and q-gram must have the same size.

Besides the find interface there is another interface for indices using suffix tree iterators to search exact needle occurrences described in the tutorial Indices.

### Assignment 3

**Type** Application

**Objective** Modify the example above to search with a Open Adressing QGram Index q-gram index for matches of "tobe" in "tobeornottobe".

**Solution** Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/index.h>

using namespace seqan;
```
```cpp
int main()
{
    typedef Index<CharString, IndexQGram<UngappedShape<4>, OpenAddressing> > TIndex;
    TIndex index("tobeornottobe");
    Finder<TIndex> finder(index);

    while (find(finder, "tobe"))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) << '"

    return 0;
}
```

We simply add a second template argument to the definition of the Index as described in the documentation of the Open Adressing QGram Index. As shape we can use an UngappedShape of length 4.

Program output:
```
[0,4) tobe
[9,13) tobe
```

**Approximate Filtration**

Currently there are no indices directly supporting an approximate search. But nevertheless, there are approximate search filters available that can be used to filter out regions of the haystack that do not contain an approximate match, see SwiftFinder and SwiftPattern. The regions found by these filters potentially contain a match and must be verified afterwards. `beginPosition`, `endPosition` and `infix` can be used to return the boundaries or sequence of such a potential match. For more details on using filters, see the article *Filtering Similar Sequences*.

**ToC**

**Contents**
- Graphs
  - Graph Basics
    * Assignment 1
    * Assignment 2
    * Assignment 3
  - Property Maps And Iterators
    * Assignment 4
  - Graph Algorithms
    * Assignments 5
    * Assignment 6

**2.1.16 Graphs**

**Learning Objective** This tutorial shows how to use graphs in SeqAn and their functionality.

**Difficulty** Average

**Duration** 1 h

**Prerequisites** *Sequences, Alignment Representation, Pairwise Sequence Alignment*
A graph in computer science is an ordered pair \( G = (V, E) \) of a set of vertices \( V \) and a set of edges \( E \). SeqAn provides different graph types and the most well-known graph algorithms as well as some specialized alignment graph algorithms. In this part of the tutorial, we demonstrate how to construct a graph in SeqAn and show the usage of some algorithms. Alignment graphs are described in the tutorial Alignment Representation.

Let us follow a simple example. We have given the following network of five cities and in this network we want to compute the shortest path from Hannover to any other city.

In the section Graph Basics, we will create the network and write the graph to a .dot file. The section Property Maps and Iterators assigns city names to the vertices and demonstrates the usage of a vertex iterator. Finally, in Graph Algorithms we will compute the shortest path by calling a single function.

After having worked through these sections you should be familiar with the general usage of graphs in SeqAn.

**Graph Basics**

The general header file for all types of graphs is `<seqan/graph_types.h>`. It comprises the class Graph and its specializations, all functions for basic graph operations, and different iterators. Later, for computing the shortest path we will also need `<seqan/graph_algorithms.h>` which includes the implementations of most of SeqAn’s graph algorithms.
We want to model the network of cities as an undirected graph and label the edges with distances. Before we start creating edges and vertices, we need some typedefs to specify the graph type.

SeqAn offers different specializations of the class Graph:, Undirected Graph, DirectedGraph, Tree, Automaton, HmmGraph, and Alignment Graph. For our example, an undirected graph will be sufficient, so we define our own graph type TGraph with the specialization Undirected Graph of the class Graph. Luckily, this specialization has an optional cargo template argument, which attaches any kind of object to the edges of the graph. This enables us to store the distances between the cities, our edge labels, using the cargo type TCargo defined as unsigned int. Using the cargo argument, we have to provide a distance when adding an edge. And when we remove an edge we also remove the distance.

```cpp
int main ()
{
    typedef unsigned int TCargo;
    typedef Graph<Undirected<TCargo> > TGraph;
    typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
```

Each vertex and each edge in a graph is identified by a so-called descriptor. The type of the descriptors is returned by the metafunction VertexDescriptor. In our example, we define a type TVertexDescriptor by calling VertexDescriptor on our graph type. Analogously, there is the metafunction EdgeDescriptor for edge descriptors.

We can now create the graph g of our type TGraph.

```cpp
TGraph g;
```

For our example, we add five vertices for the five cities, and six edges connecting the cities.

Vertices can be added to g by a call to the function addVertex. The function returns the descriptor of the created vertex. These descriptors are needed to add the edges afterwards.

```cpp
TVertexDescriptor vertBerlin = addVertex(g);
TVertexDescriptor vertHamburg = addVertex(g);
TVertexDescriptor vertHannover = addVertex(g);
TVertexDescriptor vertMainz = addVertex(g);
TVertexDescriptor vertMuenchen = addVertex(g);
```

The function addEdge adds an edge to the graph. The arguments of this function are the graph to which the edge is added, the vertices that it connects, and the cargo (which is in our case the distance between the two cities).

```cpp
addEdge(g, vertBerlin, vertHamburg, 289);
addEdge(g, vertBerlin, vertHannover, 286);
addEdge(g, vertBerlin, vertMainz, 573);
addEdge(g, vertBerlin, vertMuenchen, 586);
addEdge(g, vertHannover, vertMuenchen, 572);
addEdge(g, vertHamburg, vertMainz, 521);
```

Once we have created the graph we may want to have a look at it. SeqAn offers the possibility to write a graph to a dot file. With a tool like Graphviz you can then visualize the graph.

The only thing that we have to do is to call the function write on a file stream with the tag DotDrawing() and pass over our graph g.

```cpp
FILE* strmWrite = fopen("graph.dot", "w");
write(strmWrite, g, DotDrawing());
fclose(strmWrite);
```
After executing this example, there should be a file `graph.dot` in your directory.
Alternatively, you can use the standard output to print the graph to the screen:

```cpp
std::cout << g << '\n';
```

### Assignment 1

**Type** Review

**Objective** Copy the code from above and adjust it such that a road trip from Berlin via Hamburg and Hannover to Munich is simulated.

**Hints** Use directed Edges

**Solution** Click more... to see the solution.

```cpp
#include <iostream>
#include <seqan/graph_types.h>
#include <seqan/graph_algorithms.h>
using namespace seqan;

int main ()
{
    typedef unsigned int TCargo;
    typedef Graph<Directed<TCargo> > TGraph;
    typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
    TGraph g;

    TVertexDescriptor vertBerlin = addVertex(g);
    TVertexDescriptor vertHamburg = addVertex(g);
    TVertexDescriptor vertHannover = addVertex(g);
    TVertexDescriptor vertMuenchen = addVertex(g);

    addEdge(g, vertBerlin, vertHamburg, 289u);
    addEdge(g, vertHamburg, vertHannover, 289u);
    addEdge(g, vertHannover, vertMuenchen, 572u);

    FILE* strmWrite = fopen("graph.dot", "w");
    write(strmWrite, g, DotDrawing());
    fclose(strmWrite);

    typedef Iterator<TGraph, VertexIterator>::Type TVertexIterator;
    TVertexIterator itV(g);
    std::cout << g << std::endl;
    return 0;
}
```

### Assignment 2

**Type** Application

**Objective** Write a program which creates a directed graph with the following edges: (1,0), (0,4), (2,1), (4,1), (5,1), (6,2), (3,2), (2,3), (7,3), (5,4), (6,5), (5,6),...
Use the function `addEdges` instead of adding each edge separately. Output the graph to the screen.

**Solution**  Click more... to see the solution.

We first have to include the corresponding header file for graphs. Instead of `<seqan/graph_types.h>`, we can also include `<seqan/graph_algorithms.h>` as it already includes `<seqan/graph_types.h>`.

```cpp
#include <iostream>
#include <seqan/graph_algorithms.h>
using namespace seqan;
```

This time we define a `DirectedGraph` without cargo at the edges.

```cpp
int main() {
    typedef Graph<Directed<> > TGraph;
    typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
    typedef EdgeDescriptor<TGraph>::Type TEdgeDescriptor;
    typedef Size<TGraph>::Type TSize;

    TSize numEdges = 14;
    TVertexDescriptor edges[] = {1,0, 0,4, 2,1, 4,1, 5,1, 6,2, 3,2, 2,3, 7,3, 5,4, 6,5, 5,6, 7,6, 7,7};
    TGraph g;
    addEdges(g, edges, numEdges);
    std::cout << g << std::endl;
}
```

The screen output of the graph consists of an adjacency list for the vertices and an edge list:

Adjacency list:
0 -> 4,
1 -> 0,
2 -> 3,1,
3 -> 2,
4 -> 1,
5 -> 6,4,1,
6 -> 5,2,
7 -> 7,6,3,

Edge list:
Source: 0, Target: 4 (Id: 1)
Source: 1, Target: 0 (Id: 0)
Source: 2, Target: 3 (Id: 7)
Source: 2, Target: 1 (Id: 2)
Source: 3, Target: 2 (Id: 6)
Source: 4, Target: 1 (Id: 3)
Source: 5, Target: 6 (Id: 11)
Source: 5, Target: 4 (Id: 9)
Source: 5, Target: 1 (Id: 4)
Source: 6, Target: 5 (Id: 10)
Source: 6, Target: 2 (Id: 5)
Source: 7, Target: 7 (Id: 13)
Source: 7, Target: 6 (Id: 12)
Source: 7, Target: 3 (Id: 8)

**Assignment 3**

Type  Transfer
Objective Write a program which defines an HMM for DNA sequences:

- Define an exon, splice, and intron state.
- Consider to use the type \texttt{LogProb<}> from \texttt{seqan/basic/basic_logvalue.h} for the transition probabilities. Sequences always start in the exon state. The probability to stay in an exon or intron state is 0.9. There is exactly one switch from exon to intron. Between the switch from exon to intron state, the HMM generates exactly one letter in the splice state. The sequence ends in the intron state with a probability of 0.1.
- Output the HMM to the screen.
- Use the following emission probabilities.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>exon state</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>splice state</td>
<td>0.05</td>
<td>0.0</td>
<td>0.95</td>
<td>0.0</td>
</tr>
<tr>
<td>intron state</td>
<td>0.4</td>
<td>0.1</td>
<td>0.1</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Solution The program starts with the inclusion of \texttt{seqan/graph_algorithms.h} and \texttt{seqan/basic/basic_logvalue.h}. In this example you could include \texttt{seqan/graph_types.h} instead of the algorithms header file. However, it is likely that if you define a graph, you will call a graph algorithm as well.

```cpp
#include <iostream>
#include <seqan/graph_algorithms.h>
#include <seqan/basic/basic_math.h>
using namespace seqan;

Next, we define our types. The most interesting type here is \texttt{THmm}. It is a \texttt{Graph} with the specialization \texttt{HmmGraph}. The specialization takes itself three template arguments: the alphabet of the sequence that the HMM generates, the type of the transitions, and again a specialization. In our case, we define the transitions to be the logarithm of the probabilities (\texttt{LogProb}) and hereby simplify multiplications to summations. For the specialization we explicitly use the \texttt{Default} tag.

```cpp
int main() {
    typedef LogProb<> TProbability;
    typedef Dna TAlphabet;
    typedef Size<TAlphabet>::Type TSize;
    typedef Graph<Hmm<TAlphabet, TProbability, Default> > THmm;
    typedef VertexDescriptor<THmm>::Type TVertexDescriptor;
    typedef EdgeDescriptor<THmm>::Type TEdgeDescriptor;

After that, we define some variables, especially one of our type \texttt{THmm}.

```cpp
    Dna dnaA = Dna('A');
    Dna dnaC = Dna('C');
    Dna dnaG = Dna('G');
    Dna dnaT = Dna('T');

    THmm hmm;
```

Now we can start with defining the states. States are represented by the vertices of the HMM-specialized graph. The initial and terminating states of an HMM in SeqAn are always silent, i.e. they do not generate characters. That is why we have to define an extra begin state and tell the program that this is the initial state of the HMM. The latter is done by calling the function \texttt{assignBeginState}.

```cpp
    TVertexDescriptor begState = addVertex(hmm);
    assignBeginState(hmm, begState);
```
For our three main states we also add a vertex to the HMM with `addVertex`. Additionally, we assign the emission probabilities for all possible characters of our alphabet using `emissionProbability`.

```cpp
tvertexDescriptor exonState = addVertex(hmm);
emissionProbability(hmm, exonState, dnaA) = 0.25;
emissionProbability(hmm, exonState, dnaC) = 0.25;
emissionProbability(hmm, exonState, dnaG) = 0.25;
emissionProbability(hmm, exonState, dnaT) = 0.25;

tvertexDescriptor spliceState = addVertex(hmm);
emissionProbability(hmm, spliceState, dnaA) = 0.05;
emissionProbability(hmm, spliceState, dnaC) = 0.0;
emissionProbability(hmm, spliceState, dnaG) = 0.95;
emissionProbability(hmm, spliceState, dnaT) = 0.0;

tvertexDescriptor intronState = addVertex(hmm);
emissionProbability(hmm, intronState, dnaA) = 0.4;
emissionProbability(hmm, intronState, dnaC) = 0.1;
emissionProbability(hmm, intronState, dnaG) = 0.1;
emissionProbability(hmm, intronState, dnaT) = 0.4;
```

Finally, we need to define the end state and call `assignEndState`.

```cpp
tvertexDescriptor endState = addVertex(hmm);
assignEndState(hmm, endState);
```

For the HMM, only the transition probabilities are still missing. A transition is represented by an edge of our HMM graph type. The cargo on these edges correspond to the transition probabilities.

Since the sequences always start with an exon, we set the transition probability from the begin state to the exon state to 1.0 calling the already well-known function `addEdge`. And also the other transitions can be defined in the same way.

```cpp
addEdge(hmm, begState, exonState, 1.0);
addEdge(hmm, exonState, exonState, 0.9);
addEdge(hmm, exonState, spliceState, 0.1);
addEdge(hmm, spliceState, intronState, 1.0);
addEdge(hmm, intronState, intronState, 0.9);
addEdge(hmm, intronState, endState, 0.1);
```

To check the HMM we can simply output it to the screen:

```cpp
std::cout << hmm << ' \n';
```

This should yield the following:

```
Alphabet:
{A,C,G,T}
States:
{0 (Silent),1,2,3,4 (Silent)}
Begin state: 0
End state: 4
Transition probabilities:
0 -> 1 (1.000000)
1 -> 2 (0.100000) ,1 (0.900000)
2 -> 3 (1.000000)
3 -> 4 (0.100000) ,3 (0.900000)
4 ->
Emission probabilities:
1: A (0.250000) ,C (0.250000) ,G (0.250000) ,T (0.250000)
```
Property Maps And Iterators

So far, the vertices in our graph can only be distinguished by their vertex descriptor. We will now see how to associate the city names with the vertices.

SeqAn uses External Property Map to attach auxiliary information to the vertices and edges of a graph. The cargo parameter that we used above associated distances to the edges. In most scenarios you should use an external property map to attach information to a graph. Be aware that the word external is a hint that the information is stored independently of the graph and functions like removeVertex do not affect the property map. Property maps are simply Strings of a property type and are indexed via the already well-known vertex and edge descriptors.

Let's see how we can define a vertex property map for the city names. Our property type is a String of a city name type, a char string. We only have to create and resize this map so that it can hold information on all vertices.

```cpp
typedef String<char> TCityName;
typedef String<TCityName> TProperties;
TProperties cityNames;
resizeVertexMap(g, cityNames);
```

Next, we can enter the city names for each vertex. Note that this is completely independent from our graph object $g$.

```cpp
assignProperty(cityNames, vertBerlin, "Berlin");
assignProperty(cityNames, vertHamburg, "Hamburg");
assignProperty(cityNames, vertMuenchen, "Munich");
assignProperty(cityNames, vertMainz, "Mainz");
assignProperty(cityNames, vertHannover, "Hannover");
```

If we now want to output all vertices including their associated information we can iterate through the graph and use the iterators value to access the information in the property map.

But let us first have a quick look at iterators for graph types. SeqAn provides six different specializations for graph iterators: Vertex Iterator, Adjacency Iterator, Dfs Preorder Iterator, and Bfs Iterator for traversing vertices, and Edge Iterator and Out-edge Iterator for traversing edges. Except for the Vertex Iterator and the Edge Iterator they depend additionally to the graph on a specified edge or vertex.

To output all vertices of our graph in an arbitrary order, we can define an iterator of the specialization Vertex Iterator and determine its type with the metafunction Iterator. The functions atEnd and goNext also work for graph iterators as for all other iterators in SeqAn.

The value of any type of vertex iterator is the vertex descriptor. To print out all city names we have to call the function getProperty on our property map cityNames with the corresponding vertex descriptor that is returned by the value function.

```cpp
typedef Iterator<TGraph, VertexIterator>::Type TVertexIterator;
TVertexIterator itV(g);
for(;!atEnd(itV);goNext(itV)) {
    std::cout << value(itV) << ':' << getProperty(cityNames, value(itV)) << std::endl;
}
```

The output of this piece of code should look as follows:

0:Berlin
1:Hamburg
2:Hannover
3:Mainz
4:Munich
Assignment 4

Type Application

Objective Add a vertex map to the program from task 2:

1. The map shall assign a lower-case letter to each of the seven vertices. Find a way to assign the properties to all vertices at once in a single function call (without using the function assignProperty for each vertex separately).

2. Show that the graph is not connected by iterating through the graph in depth-first-search ordering. Output the properties of the reached vertices.

Solution Our aim is not to assign all properties at once to the vertices. Therefore, we create an array containing all the properties, the letters 'a' through 'h'.

The function assignVertexMap does not only resize the vertex map (as resizeVertexMap does) but also initializes it. If we specify the optional parameter prop, the values from the array prop are assigned to the items in the property map.

```cpp
String<char> nameMap;
char names[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
assignVertexMap(g, nameMap, names);
```

To iterate through the graph in depth-first-search ordering we have to define an Iterator with the specialization DfsPreorderIterator.

The vertex descriptor of the first vertex is 0 and we choose this vertex as a starting point for the depth-first-search through our graph g with the iterator dfsIt:

```cpp
TVertexDescriptor start = 0;
typedef Iterator<TGraph, DfsPreorder>::Type TDfsIterator;
TDfsIterator dfsIt(g, start);
std::cout << "Iterate from '/' in depth-first-search ordering: " << getProperty(nameMap, start) << " in depth-first-search ordering: ";
while (!atEnd(dfsIt)) {
    std::cout << getEntry(nameMap, getValue(dfsIt)) << ", ";
goNext(dfsIt);
} std::cout << std::endl;
```

For the chosen starting point, only two other vertices can be reached:

Iterate from 'a' in depth-first-search ordering: a, e, b,

Graph Algorithms

Now that we completed creating the graph we can address the graph algorithms. Here is an overview of some graph algorithms currently available in SeqAn:

Elementary Graph Algorithms

- Breadth-First Search (breadthFirstSearch)
- Depth-First Search (depthFirstSearch)
- Topological Sort (topologicalSort)
- Strongly Connected Components (stronglyConnectedComponents)

Minimum Spanning Tree
• Prim’s Algorithm (primAlgorithm)
• Kruskal’s Algorithm (kruskalAlgorithm)

Single-Source Shortest Path
• DAG Shortest Path (dagShortestPath)
• Bellman-Ford (bellmanFordAlgorithm)
• Dijkstra (dijkstra)

All-Pairs Shortest Path
• All-Pairs Shortest Path (allPairsShortestPath)
• Floyd Warshall (floydWarshallAlgorithm)

Maximum Flow
• Ford-Fulkerson (fordFulkersonAlgorithm)

Transitive Closure
• Transitive Closure (transitiveClosure)

Bioinformatics Algorithms
• Needleman-Wunsch (globalAlignment)
• Gotoh (globalAlignment)
• Hirschberg with Gotoh (globalAlignment)
• Smith-Waterman (localAlignment)
• Multiple Sequence Alignment (globalMsaAlignment)
• UPGMA (upgmaTree)
• Neighbor Joining (njTree)

The biological algorithms use heavily the alignment graph. Most of them are covered in the tutorial Alignment Representation. All others use the appropriate standard graph. All algorithms require some kind of additional input, e.g., the Dijkstra algorithm requires a distance property map, alignment algorithms sequences and a score type and the network flow algorithm capacities on the edges.

Generally, only a single function call is sufficient to carry out all the calculations of a graph algorithm. In most cases you will have to define containers that store the algorithms results prior to the function call.

In our example, we apply the shortest-path algorithm of Dijkstra. It is implemented in the function dijkstra. Let’s have a look at the input parameters. The first parameter is of course the graph, g. Second, you will have to specify a vertex descriptor. The function will compute the distance from this vertex to all vertices in the graph. The last input parameter is an edge map containing the distances between the vertices. One may think that the distance map is already contained in the graph. Indeed this is the case for our graph type but it is not in general. The cargo of a graph might as well be a string of characters or any other type. So, we first have to find out how to access our internal edge map. We do not need to copy the information to a new map. Instead we can define an object of the type InternalMap of our type TCargo. It will automatically find the edge labels in the graph when the function property or getProperty is called on it with the corresponding edge descriptor.

The output containers of the shortest-path algorithm are two property maps, predMap and distMap. The predMap is a vertex map that determines a shortest-paths-tree by mapping the predecessor to each vertex. Even though we are not interested in this information, we have to define it and pass it to the function. The distMap indicates the length of the shortest path to each vertex.
typedef Size<TGraph>::Type TSize;
InternalMap<TCargo> cargoMap;
String<TVertexDescriptor> predMap;
String<TSize> distMap;

Having defined all these property maps, we can then call the function `dijkstra`:

dijkstra(g, vertHannover, cargoMap, predMap, distMap);

Finally, we have to output the result. Therefore, we define a second vertex iterator `itV2` and access the distances just like the city names with the function `property` on the corresponding property map.

```cpp
TVertexIterator itV2(g);
while (!atEnd(itV2)) {
    std::cout << "Shortest path from " << property(cityNames, vertHannover) << " to " <<
    std::cout << property(distMap, value(itV2)) << std::endl;
    goNext(itV2);
}
return 0;
```

## Assignments 5

### Type

**Application**

**Objective** Write a program which calculates the connected components of the graph defined in task 1. Output the component for each vertex.

**Solution** SeqAn provides the function `stronglyConnectedComponents` to compute the connected components of a directed graph. The first parameter of this function is of course the graph. The second parameter is an output parameter. It is a vertex map that will map a component id to each vertex. Vertices that share the same id are in the same component.

```cpp
String<unsigned int> component;
stronglyConnectedComponents(g, component);
```

Now, the only thing left to do is to walk through our graph and output each vertex and the corresponding component using the function `getProperty`. One way of doing so is to define a `VertexIterator`.

```cpp
std::cout << "Strongly Connected Components: " << std::endl;
typedef Iterator<TGraph, VertexIterator>::Type TVertexIterator;
TVertexIterator it(g);
while (!atEnd(it)) {
    std::cout << "Vertex " << getProperty(nameMap, getValue(it)) << ": ";
    std::cout << "Component = " << getProperty(component, getValue(it)) << std::endl;
    goNext(it);
}
return 0;
```

The output for the graph defined in the Assignment 4 looks as follows:

```
Strongly Connected Components:
Vertex a: Component = 3
Vertex b: Component = 3
Vertex c: Component = 2
Vertex d: Component = 2
```
Vertex e: Component = 3
Vertex f: Component = 1
Vertex g: Component = 1
Vertex h: Component = 0

The graph consists of four components. The first contains vertex a, b, and e, the second contains vertex c and d, the third contains vertex f and g and the last contains only vertex h.

Assignment 6

Type Application

Objective Extend the program from the Assignment 5. Given the sequence s = "CTTCATGTGAAAGCAGACGTAAGTCA".

1. calculate the Viterbi path of s and output the path as well as the probability of the path and
2. calculate the probability that the HMM generated s with the forward and backward algorithm.

Solution In Assignment 3 we defined an HMM with three states: exon, splice, and intron.

The Viterbi path is the sequence of states that is most likely to produce a given output. In SeqAn, it can be calculated with the function viterbiAlgorithm. The produced output for this assignment is the DNA sequence s.

The first parameter of the function viterbiAlgorithm is of course the HMM, and the second parameter is the sequence s. The third parameter is an output parameter that will be filled by the function. Since we want to compute a sequence of states, this third parameter is a String of VertexDescriptors which assigns a state to each character of the sequence s.

The return value of the function viterbiAlgorithm is the overall probability of this sequence of states, the Viterbi path.

The only thing left is to output the path. The path is usually longer than the given sequence. This is because the HMM may have silent states, e.g. the begin and end state. To check if a state is silent SeqAn provides the function isSilent.

```
String<Dna> sequence = "CTTCATGTGAAAGCAGACGTAAGTCA";
String<TVertexDescriptor> path;
TProbability p = viterbiAlgorithm(hmm, sequence, path);
std::cout << "Viterbi algorithm" << std::endl;
std::cout << "Probability of best path: " << p << std::endl;
for (TSize i = 0; i < length(sequence); ++i) std::cout << sequence[i] << ',';
std::cout << std::endl;
for (TSize i = 0; i < length(path); ++i) {
    std::cout << path[i];
    if (isSilent(hmm, path[i])) std::cout << " (Silent)";
    if (i < length(path) - 1) std::cout << ',';
}
std::cout << std::endl;
```

The output of the above piece of code is:

```
Viterbi algorithm
Probability of best path: 1.25465e-18
Sequence:
State path:
0 (Silent),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,3,3,3,3,3,3,3,4 (Silent)
```
It is even simpler to use the forward algorithm in SeqAn since it needs only the HMM and the sequence as parameters and returns a single probability. This is the probability of the HMM to generate the given sequence. The corresponding function is named `forwardAlgorithm`.

```cpp
std::cout << "Forward algorithm" << std::endl;
p = forwardAlgorithm(hmm, sequence);
std::cout << "Probability that the HMM generated the sequence: " << p << std::endl;
```

Analogously, the function `backwardAlgorithm` implements the backward algorithm in SeqAn.

```cpp
std::cout << "Backward algorithm" << std::endl;
p = backwardAlgorithm(hmm, sequence);
std::cout << "Probability that the HMM generated the sequence: " << p << std::endl;
return 0;
```

The output of these two code fragments is:

Forward algorithm
Probability that the HMM generated the sequence: 2.71585e-18
Backward algorithm
Probability that the HMM generated the sequence: 2.71585e-18

---

2.1.17 Basic Sequence I/O

**Learning Objective** You will learn how to read and write sequence files (FASTA and FASTQ) using a simple, high-level API in the SeqAn library. This includes reading and writing compressed files.

**Difficulty** Basic

**Duration** 30 min

**Prerequisites** Sequences

This tutorial explains how to read and write sequence files using the `SequenceStream` class.
The SeqAn library’s I/O functionality (i.e., reading from and writing to files) is organized in layers and SequenceStream is in the highest layer. It provides an easy-to-use API for reading and writing sequence files, automatically compressing and decompressing data, and support for different sequence formats. This flexibility comes at a slight cost of performance, compared to using the more low-level APIs.

The lower layers are responsible for providing raw file I/O functionality and adding parsing functionality. Their usage is not part of this tutorial and is explained in the Tutorials File I/O, Sequence File I/O, and Parsing.

After completing the tutorial, you will be able to read and write sequence files in the formats supported by the SequenceStream class.

A First Working Example

Let us start out with a minimal working example. The following small program will read the file example.fa (which we will create later) from the current directory and print out the identifier and the sequence of the first record.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main()
{
    seqan::CharString id;
    seqan::Dna5String seq;

    seqan::SequenceStream seqStream("example.fa");
    readRecord(id, seq, seqStream);
    std::cout << id << '	' << seq << '
';

    return 0;
}
```

We use the SequenceStream constructor with one parameter, the path to the file we want to read. This will open the file and guess the file format from the file contents. The class will read the first some thousand characters and try to guess what format the file is. Note that you can also pass "-" as the file name. This will open the standard input when reading and the standard output when writing.

After construction, the seqStream object is ready for reading. We use the function readRecord to read the first record from the file. readRecord always reads the next record in the file. readRecord always reads the next record in the file.

**Tip:** FASTA/FASTQ and Record-Based Files

Most files in bioinformatics have a record-based structure. Often, a file format requires or allows for a header that contains information about the file format. Then, the file contains a list of records, one after another.

The FASTA and FASTQ formats do not have a header but only contain lists of records. For example, a FASTQ record contains the sequence id, the sequence characters, and a quality value for each character.

Note that we do not have to close the file manually. The SequenceStream object will automatically close any open files when it goes out of scope and it is destructred. If you want to force a file to be closed, you can use the function close.

Adding Error Handling

Now, create a new FASTA file named example.fa in a directory of your choice with the following content:
>seq1
CCCCCCCCCCCCCCC
>seq2
CGATCGATC
>seq3
TTTTTTT

Then, copy the program above into new application basic_seq_io_example, adjust the path "example.fa" to the just created FASTA file, compile the program, and run it. For example, if you stored the file example.fa in /home/username/example.fa, you replace the line seqan::SequenceStream seqStream("example.fa"); from above with seqan::SequenceStream seqStream("/home/username/example.fa");. You should see the following output:

# basic_seq_io
seq1 CCCCCCCCCCCCCC

Assignment 1

Type :: Review

Objective :: Adjust the program above to use the first command line parameter argv[1], i.e. the first argument.

Check that there actually is such an argument (argc >= 2) and let main() return 1 otherwise.

Solution ::

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n"
        return 1;
    }

    seqan::CharString id;
    seqan::Dna5String seq;

    seqan::SequenceStream seqStream(argv[1]);
    readRecord(id, seq, seqStream);
    std::cout << id << \t << seq << \n;

    return 0;
}
```

Our program is very simple but there is one large problem. Anything can go wrong during file I/O and have not used any means to handle such errors. Possible errors include: the file permissions forbid a certain operations, the file does not exist, there is a disk reading error, a file read from a remote location gets deleted while we are reading from it, or there is a physical error in the hard disk.

Let us add some error handling. At the very least, we should detect errors. If possible, we should try to recover from the error (sometimes it is possible to return default values instead of loading values from a file) or otherwise stop the current task in an organized fashion and notify the user about the problem.

2.1. Tutorial

171
We can use the Function `isGood` to check whether the `SequenceStream` object is ready for any more reading. After the creation of the object, this function indicates whether the file could be opened successfully by returning `true`. The function `readRecord` returns an `int` that indicates whether the reading was successful. If everything went fine, it returns 0, and a different value otherwise.

Note that `isGood` queries the state of the stream and returns a `bool` indicating whether the stream is ready for reading/writing (true for “is good” and false for “is not good”). `readRecord`, on the other hand, returns an `int` indicating whether there was any error (0 for “is good” and a non-0 value for “is not good”, as it is customary in Unix programming).

The program will now read as follows:

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main()
{
    seqan::CharString id;
    seqan::Dna5String seq;

    seqan::SequenceStream seqStream("example.fa");
    if (!isGood(seqStream))
    {
        std::cerr << "ERROR: Could not open the file.\n";
        return 1;
    }
    if (readRecord(id, seq, seqStream) != 0)
    {
        std::cerr << "ERROR: Could not read from example.fa!\n";
        return 1;
    }

    std::cout << id << '	' << seq << '\n';
    return 0;
}
```

**Assignment 2**

**Type Review**

**Objective** Change your program from above to perform these checks, too.

**Solution**

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n";
        return 1;
    }
```
Assignment 3

Type Application

Objective Change your program from above to loop over all sequences and print them in the same fashion.

Hint You can use the function atEnd to check whether a SequenceStream object is at the end of the file.

Solution

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n";
        return 1;
    }

    seqan::CharString id;
    seqan::Dna5String seq;

    seqan::SequenceStream seqStream(argv[1]);
    if (!isGood(seqStream))
    {
        std::cerr << "ERROR: Could not open the file.\n";
        return 1;
    }

    while (!atEnd(seqStream))
    {
        if (readRecord(id, seq, seqStream) != 0)
        {
            std::cerr << "ERROR: Could not read from example.fa!\n";
            return 1;
        }

        std::cout << id << \t << seq << \n;
    }

    return 0;
}
```
After completing Assignment 3, you should be able to run your program on the example file we created above and see the following output:

```
# basic_seq_io_example example.fa
seq1 CCCCCCCCCCCCCCC
seq2 CGATCGATC
seq3 TTTTTTT
```

**The Interface for Reading**

There are three major usage patterns for sequence I/O:

1. **We want to read all records** from the file into memory, for example for building an index.

2. **We want to read the file into memory record by record**, so the memory usage is minimal. We could then perform some computation on each record, e.g. search it in an index.

3. **We want to read a batch of records** into memory, e.g. 100k records at a time. Then, we perform some computation on the records, for example in parallel with 4 threads on 25k records each.

These use cases are supported by the functions `readAll`, `readRecord`, and `readBatch`.

Each of these functions is available in two variants. The first accepting only the sequence identifier and sequence characters besides the `SequenceStream` object and the second also accepting the a `CharString` for the PHRED base qualities. If a file does not contain any qualities and the function variant with quality values is used then the quality strings are returned as empty. When writing a file with qualities and the function variant without quality values is used then the qualities are written out as ‘I’, i.e. PHRED score 40.

When `DnaQ` or `Dna5Q` are used, then you should use the function variant without a parameter for qualities. The qualities are simply stored directly in the sequence characters.

As to be expected, when there are characters in the file that are not valid characters in the `String` then the alphabet-dependent conversion is performed. For example, for `Dna` and `Rna` this means a conversion of the invalid character to ‘A’, and for `Dna5` and `Rna5` this means a conversion to ‘N’.

Here is an example for using `readRecord`:

```cpp
seqan::CharString id;
seqan::Dna5String seq;
seqan::CharString qual;
int res = 0;

seqan::SequenceStream seqStream("in.fq");

res = readRecord(id, seq, seqStream);
res = readRecord(id, seq, qual, seqStream);
```

The functions `readAll` and `readBatch` use `StringSet` instead of `String`. The function `readBatch` reads up to the given number of records. It is not an error if there are less records.
Assignment 4

Type Application

Objective Change your result of Assignment 3 to use the variant of readRecord that also reads in the qualities and writes them next to the sequences. Create the following FASTQ file example.fq.

@seq1
CCCCCCCCCC
+
IIIIIIIIIIIIIIII
@seq2
CGATCGATC
+
IIIIIIII
@seq3
TTTTTT
+
IIIIHHG

When your program is called on this file, the result should look as follows.

# basic_seq_io_example example.fq
seq1  CCCCCCCCCCC   IIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
seq2  CGATCGATC   IIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
seq3  TTTTTTT   IIIIIHHG

Solution

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n";
        return 1;
    }

    seqan::CharString id;
    seqan::Dna5String seq;
    seqan::CharString qual;
    seqan::SequenceStream seqStream("in.fq");

    res = readAll(ids, seqs, seqStream);
    res = readAll(ids, seqs, quals, seqStream);

    res = readBatch(ids, seqs, seqStream, 10);
    res = readBatch(ids, seqs, quals, seqStream, 10);
}
```c++
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv) {
  if (argc < 2) {
    std::cerr << "USAGE: basic_seq_io_example FILENAME\n";
    return 1;
  }

  seqan::SequenceStream seqStream(argv[1], seqan::SequenceStream::WRITE);
  if (!isGood(seqStream)) {
    std::cerr << "ERROR: Could not open the file.\n";
    return 1;
  }

  seqan::CharString id = "seq1";
  seqan::Dna5String seq = "CGAT";
  if (writeRecord(seqStream, id, seq) != 0) {
    std::cerr << "ERROR: Could not write to file!\n";
    return 1;
  }

  return 0;
}
```

**The Interface for Writing**

Now that you know how to read sequence files, writing them will come easy to you. We can open files for writing by giving `seqan::SequenceStream::WRITE` as the second parameter to the `SequenceStream` constructor. Create a new SeqAn app `basic_seq_io_example2` in your sandbox and change the C++ file `basic_seq_io_example2.cpp` in this application to have the content below. This program already has all the bells and whistles for error checking.

```c++
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv) {
  if (argc < 2) {
    std::cerr << "USAGE: basic_seq_io_example FILENAME\n";
    return 1;
  }

  seqan::SequenceStream seqStream(argv[1], seqan::SequenceStream::WRITE);
  if (!isGood(seqStream)) {
    std::cerr << "ERROR: Could not open the file.\n";
    return 1;
  }

  seqan::CharString id = "seq1";
  seqan::Dna5String seq = "CGAT";
  if (writeRecord(seqStream, id, seq) != 0) {
    std::cerr << "ERROR: Could not write to file!\n";
    return 1;
  }

  return 0;
}
```
The first lines are similar to those in the solution to Assignment 4. However, instead of opening the file using `seqan::SequenceStream seqStream(argv[1]);`, we use `seqan::SequenceStream seqStream(argv[1], seqan::SequenceStream::WRITE);` this opens the file with the name in `argv[1]` for writing instead of for reading. Also, instead of reading records, we write one record.

The program writes out one sequence with id “seq1” and the contents “CGAT” to the file given on the command line. Note that `SequenceStream` will guess the format from the file name. A file ending in `.fa` and `.fasta` mean FASTA, `.fq` and `.fastq` means FASTQ. Optionally, you can force to use any file format with the third parameter to the `SequenceStream` constructor.

Let us try out the program from above:

```
# basic_seq_io_example2 out.fa
# cat out.fa
>seq1
CGAT
```

```
# basic_seq_io_example2 out.fq
# cat out.fq
@seq
CGAT
+
```

Assignment 5

**Type** Reproduction

**Objective** Change the program from above to write out a second sequence.

**Solution**

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n"
                   "return 1;"
        return 1;
    }

    seqan::SequenceStream seqStream(argv[1], seqan::SequenceStream::WRITE);
    if (!isGood(seqStream))
    {
        std::cerr << "ERROR: Could not open the file.\n"
                   "return 1;"
        return 1;
    }

    seqan::CharString id = "seq1";
    seqan::Dna5String seq = "CGAT";

    if (writeRecord(seqStream, id, seq) != 0)
```cpp
std::cerr << "ERROR: Could not write to file!\n";
return 1;
}

id = "seq1";
seq = "TTTT";
if (writeRecord(seqStream, id, seq) != 0)
{
    std::cerr << "ERROR: Could not write to file!\n";
    return 1;
}
return 0;
}
```

There are two functions for writing to sequence files using `SequenceStream`. One, `writeRecord`, for writing one sequence record from `Strings`, and another one, `writeAll`, for writing all sequences from `StringSets`.

Again, they come in one variant with and another variant without base qualities. When writing to a FASTQ file using the function without qualities, the PHRED score 40 is written for each character (‘I’) and when writing to a FASTA file with the variant with qualities, the qualities are ignored. When using `DnaQ` or `Dna5Q`, the variant without qualities parameter writes out the qualities stored in the sequence characters themselves.

Here is an example for using `writeRecord`:

```cpp
seqan::CharString id;
seqan::Dna5String seq;
seqan::CharString qual;

seqan::SequenceStream seqStream("out.fq", seqan::SequenceStream::WRITE);
res = writeRecord(seqStream, id, seq);
res = writeRecord(seqStream, id, seq, qual);
```

And here is an example for using `writeAll`:

```cpp
seqan::StringSet<seqan::CharString> ids;
seqan::StringSet<seqan::Dna5String> seqs;
seqan::StringSet<seqan::CharString> quals;

seqan::SequenceStream seqStream("out.fq", seqan::SequenceStream::WRITE);
res = writeAll(seqStream, ids, seqs);
res = writeAll(seqStream, ids, seqs, quals);
```

**Assignment 6**

**Type** Application

**Objective** Change the result of Assignment 5 to store the data for the two records in `StringSets` and write them out using `writeAll`.

**Solution**

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>
```
int main(int argc, char const ** argv)
{
    if (argc < 2)
    {
        std::cerr << "USAGE: basic_seq_io_example FILENAME\n"
        return 1;
    }

    seqan::SequenceStream seqStream(argv[1], seqan::SequenceStream::WRITE);
    if (!isGood(seqStream))
    {
        std::cerr << "ERROR: Could not open the file.\n"
        return 1;
    }

    seqan::StringSet<seqan::CharString> ids;
    appendValue(ids, "seq1");
    appendValue(ids, "seq2");
    seqan::StringSet<seqan::Dna5String> seqs;
    appendValue(seqs, "CGAT");
    appendValue(seqs, "TTTT");

    if (writeAll(seqStream, ids, seqs) != 0)
    {
        std::cerr << "ERROR: Could not write to file!\n"
        return 1;
    }

    return 0;
}

Compressed Files

Using compressed files is simple. When opening a file for reading, SequenceStream will automatically detect whether
the file is compressed or not, the same it detects the sequence file format for you. If you run into problems here, make
sure that you have zlib and/or libbz2 installed (see Dependencies on Compression Libraries below).

When opening a file for writing, SequenceStream will infer the compression type (gzip, bzip2, or plain text only) and
the file format (FASTA or FASTQ) from the file ending. First, the file type is guessed: A file ending in .gz means
“gzip-compressed”, one ending in .bz2 means “bzip2-compressed”. Then, the .gz or .bz2 suffix is ignored when
guessing the file format. A path ending in .fa and .fasta mean FASTA, .fq and .fastq mean FASTQ. Since
the suffixes .gz and .bz2 are ignored, .fa.gz, .fa.bz2, ... mean FASTA too and .fq.gz,.fq.bz2,... mean
FASTQ.

File type detection from standard input is currently limited to either gzip-compressed or plain-text data.

Note that you can also use additional parameters in the SequenceStream constructor to force a certain file type and file
format when writing. You can also force a certain file type and format when reading but this is only helpful in the few
instances where the automatic detection fails.

This means that all the examples and your solutions to the assignments from above already have compression support
built-in, if the compression libraries are available.

Dependencies on Compression Libraries

For accessing compressed files, you need to have zlib installed for reading .gz files and libbz2 for reading .bz2 files.
If you are using Linux or Mac Os X and you followed the Getting Started tutorial closely then you should have already installed the necessary libraries. On Windows, you will need to follow Installing Contribs On Windows to get the necessary libraries.

You can check whether you have installed the libraries to use zlib and libbz2 by running CMake again. Simply call cmake . in your build directory. At the end of the output, there will be a section “SeqAn Features”. If you can read ZLIB - FOUND and BZIP2 - FOUND then you can use zlib and libbz2 in your programs.

Congratulations, you have now learned to write simple and robust sequence I/O code using SeqAn!

Next Steps

• Read the Wikipedia articles about the FASTA file format and the FASTQ file format and quality values to refresh your knowledge.
• Read the Indexed FASTA I/O tutorial to learn how to read FASTA files efficiently in a random-access fashion.
• Continue with the Tutorial.

2.18 Indexed FASTA I/O

Learning Objective :: In this tutorial, you will learn how to use a FASTA Index file (.fai) for indexed random-access to FASTA files. This is useful for retrieving regions (e.g. chr1:123-10004) or single sequences (e.g. chr1) from FASTA files quickly.

Difficulty Average

Duration 30 min

Prerequisites Sequences

The idea of FASTA index files (FAI) comes from the samtools program by Heng Li. The program provides a command samtools faidx for rapidly accessing parts of a large FASTA file (e.g. querying for the first chromosome by the identifier “chr1” or querying for 900 characters starting from character 100 (1-based) by chr1:100-1,000). To do this, the program creates an index file that contains one entry for each sequence. If the FASTA file is named path/sequence.fasta, the index file is usually named path/sequence.fasta.fai.

Using such index files, it is possible to rapidly read parts of the given sequence file. The module <seqan/seq_io.h> allows to create and read such .fai index files and exposes an API to read parts randomly of FASTA file.

Note: FASTA/FASTQ Meta Data and Sequence Ids
FASTA and FASTQ files have one meta data record for each sequence. This usually contains the sequence name but sometimes a lot of additional information is stored. There is no consensus for the meta data.

It is common, however, to store the identifier (id) of the sequence at the beginning of the meta data field before the first space. The id is unique to the whole file and often identifies the associate sequence uniquely in a data base (see section Sequence Identifiers on the Wikipedia FASTA format page).

While not documented anywhere explicite, only the characters up to the first space are used as the identifiers by widely used tools such as BWA. Only the identifier is carried over into files generated from the input files (BWA uses the sequence id from the genome FASTA to identify the contig/chromosome and the read id as the read name in the SAM output).

How Does It Work?

There are two requirements that a FASTA file has to fulfill to work with the FAI scheme. For each sequence in the FASTA file, the number of characters stored per line has to be the same and the number of bytes per line has to be the same. The first restriction speaks for itself, the second restriction means that the same line ending character has to be used and no line should contain any additional spaces.

The index file then stores records of sequence identifier, length, the offset of the first sequence character in the file, the number of characters per line, and the number of bytes per line. With this information, we can easily compute the byte offset of the i-th character of a sequence in a file by looking at its index record. We skip to this byte offset in the file and from there, we can read the necessary sequence characters.

Building the Index

The class FaiIndex allows for building and loading FAI indices. To build such an index, we use the function build of the class FaiIndex. The first parameter is the FaiIndex object, the second is the path to the FASTA file. The function returns an integer indicating the result (as usual, 0 for no errors, a value different from 0 indicating an error).

```cpp
#include <seqan/seq_io.h>

seqan::FaiIndex faiIndex;
int res = build(faiIndex, "path/to/file.fasta");
if (res != 0)
    std::cerr << "ERROR: Could not build the index!\n";
```

There is an alternative variant of this function where you can pass the path to the FAI file that is to be built as third parameter. The FAI file name will be stored in the FaiIndex.

```cpp
#include <seqan/seq_io.h>

seqan::FaiIndex faiIndex;
int res = build(faiIndex, "path/to/file.fasta", "another/path/file.fasta.fai");
if (res != 0)
    std::cerr << "ERROR: Could not build the index!\n";
```

We can write out the index after building it using the function write:

```cpp
#include <seqan/seq_io.h>

seqan::FaiIndex faiIndex;
// ... index building here ...
int res = write(faiIndex, "path/to/file.fasta.fai");
```
if (res != 0)
    std::cerr << "ERROR: Could not write the index to file!\n";

Assignment 1

Building a FAI index

Type Application

Objective Write a small program build_fai that takes one parameter from the command line, the path to a FASTA file. The program should then build a FAI index and write it out.

Hints Using the two-parameter variant of build is good enough.

Solution

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
    {
        std::cerr << "USAGE: build_fai FILE.fa\n";
        return 1;
    }

    seqan::FaiIndex faiIndex;
    if (build(faiIndex, argv[1]) != 0)
    {
        std::cerr << "ERROR: Could not build FAI index for file " << argv[1] << ".\n";
        return 1;
    }

    seqan::CharString faiFilename = argv[1];
    append(faiFilename, ".fai");

    if (write(faiIndex, toCString(faiFilename)) != 0)
    {
        std::cerr << "ERROR: Could not write the index to file!\n";
        return 1;
    }

    std::cout << "Index file " << faiFilename << " was successfully created.\n";
    return 0;
}
```

Using the Index

To load a FAI file, we use the function read: We pass the FaiIndex object as the first and the path to the FASTA file as the second parameter. The function returns an int indicating success (value 0) or failure (non-0 value).

```cpp
#include <seqan/seq_io.h>

seqan::FaiIndex faiIndex;
int res = read(faiIndex, "path/to/file.fasta");
```
if (res != 0)
    std::cerr << "ERROR: Could not read FAI index path/to/file.fasta.fai\n";

In the example above, the FAI file "path/to/file.fasta.fai" would be loaded. Optionally, we can specify an extra path to the FAI file:

#include <seqan/seq_io.h>

seqan::FaiIndex faiIndex;
int res = read(faiIndex, "path/to/file.fasta", "path/to/index.fai");
if (res != 0)
    std::cerr << "ERROR: Could not load FAI index path/to/index.fai\n";

After loading the index, we can then use the index to map a sequence id to its (zero-based) position (a position \( i \) meaning that it is the \( i \)-th sequence) in the FASTA file using getIdByName. The function gets the FaiIndex to use, the id of the sequence, and an unsigned position as parameters. It returns a bool indicating whether the mapping was successful (true on success, false on failure).

unsigned idx = 0;
if (getIdByName(faiIndex, "chr1", idx))
    std::cerr << "ERROR: FAI index has no entry for chr1.\n";

Once we have the index for the sequence in the FASTA file, we can then query the FailIndex for the length of the sequence using sequenceLength, get the whole sequence using readSequence, or get just a part of the sequence using readRegion.

unsigned seqLength = sequenceLength(faiIndex, idx);

// Load first 1000 characters of chr1.
seqan::CharString seqChr1Prefix;
if (readRegion(seqChr1Prefix, faiIndex, idx, 0, 1000) != 0)
    std::cerr << "ERROR: Could not load chr1.\n";

// Load all of chr1.
seqan::CharString seqChr1;
if (readSequence(seqChr1, faiIndex, idx) != 0)
    std::cerr << "ERROR: Could not load chr1.\n";

The sequence length can be determined by only looking at the index. When loading the sequence or a sequence infix, only the relevant part of the file will be touched. Thus, only the minimal amount of memory, time, and disk I/O is used.

Assignment 2

Using the FAI index

Type  Application

Objective  Write a small program query_fai that takes four parameters from the command line: A path to a FASTA file, the id of the sequence, a begin and an end position. The program should then read the given infix of the given sequence from the file and print it to stdout.

Hint  Use the function lexicalCast2 to convert strings of numbers into integers.

Solution  The program appears to be very long, but most is error handling, as usual with robust I/O code.

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>
```cpp
#include <seqan/stream.h>

int main(int argc, char const ** argv) {
    if (argc != 5) {
        std::cerr << "USAGE: build_fai FILE.fa SEQ BEGIN END\n"
                   "\n"
        return 1;
    }

    // Try to load index and create on the fly if necessary.
    seqan::FaiIndex faiIndex;
    if (seqan::read(faiIndex, argv[1]) != 0) {
        if (build(faiIndex, argv[1]) != 0) {
            std::cerr << "ERROR: Index could not be loaded or built.\n"
                       "\n"
            return 1;
        }

        if (write(faiIndex) != 0) // Name is stored from when reading.
            std::cerr << "ERROR: Index could not be written to disk.\n"
                       "\n"
            return 1;
    }

    // Translate sequence name to index.
    unsigned idx = 0;
    if (!getIdByName(faiIndex, argv[2], idx)) {
        std::cerr << "ERROR: Index does not know about sequence " << argv[2] << "\n"
                   "\n"
        return 1;
    }

    // Convert positions into integers.
    unsigned beginPos = 0, endPos = 0;
    if (!seqan::lexicalCast2(beginPos, argv[3])) {
        std::cerr << "ERROR: Cannot cast " << argv[3] << " into an unsigned.\n"
                   "\n"
        return 1;
    }

    if (!seqan::lexicalCast2(endPos, argv[4])) {
        std::cerr << "ERROR: Cannot cast " << argv[4] << " into an unsigned.\n"
                   "\n"
        return 1;
    }

    // Make sure begin and end pos are on the sequence and begin <= end.
    if (beginPos > sequenceLength(faiIndex, idx))
        beginPos = sequenceLength(faiIndex, idx);
    if (endPos > sequenceLength(faiIndex, idx))
        endPos = sequenceLength(faiIndex, idx);
    if (beginPos > endPos)
        endPos = beginPos;

    // Finally, get infix of sequence.
    seqan::Dna5String sequenceInfix;
    if (readRegion(sequenceInfix, faiIndex, idx, beginPos, endPos) != 0)
```
Next Steps

- Read the Wikipedia articles about the FASTA file format and the FASTQ file format and quality values to refresh your knowledge.
- Read the API documentation of the GenomicRegion class for storing regions (sequence identifier, start and end position). There also is functionality for parsing strings like chr1:2,032–3,212 into GenomicRegion objects.
- Continue with the Tutorial.

2.1.19 Basic SAM and BAM I/O

Learning Objective In this tutorial, you will learn how to use the high-level interface BamStream class to read and write SAM and BAM files.

Difficulty Average

Duration 1 h (45 min if you know the SAM format)

Prerequisites Sequences, Basic Sequence I/O, Exposure to the SAM format

This tutorial deals with how to easily read and write SAM and BAM files using the BamStream class. It starts out with a quick reminder on the structure of SAM (and also BAM) files and will then continue with how to read and write SAM/BAM files and access the tags of a record.

Important: Note that this tutorial is targeted at readers that already know about the SAM format. If you do not know
Both SAM and BAM file store multi-read alignments. Storing alignments of longer sequences such as contigs from assemblies is also possible, but less common. Here, we will focus on multi-read alignments.

SAM files are text files, having one record per line. BAM files are just binary, compressed versions of SAM files that have a stricter organization and aim to be more efficiently useable by programs and computers. The nuts and bolts of the formats are described in the SAM Format Specification.

The SAM and BAM related I/O functionality in SeqAn focuses on allowing access to these formats in SeqAn with thin abstractions. The Fragment Store Tutorial shows how to get a more high-level abstraction for multi-read alignments.

**Important:** SAM/BAM I/O vs. Fragment Store

The Fragment Store provides a high-level view of multi-read alignments. This is very useful if you want to do SNP or small indel detection because you need to access the alignment of the reads around your candidate regions. However, storing the whole alignment of a 120GB BAM file obviously is not a good idea.

The SAM/BAM I/O functionality in SeqAn is meant for sequentially reading through SAM and BAM files. Jumping within BAM files using BAI indices is described in the SAM and BAM I/O tutorial.

**SAM / BAM File Structure**

We will give an quick overview of the SAM and BAM formats here. Note that this overview serves more to remind you what the formats are about and are not meant to teach how to use the SAM and BAM format.

The following shows an example of a SAM file.

```bash
@HD VN:1.3 SO:coordinate
@SQ SN:ref LN:45
@SQ SN:ref2 LN:40
r001 163 ref 7 30 8M4I4M1D3M = 37 39 TTAGATAAGGGATACTG * XX:B:S,12561,2,20,112
r002  0 ref 9 30 1S2I6M1P1I1P1I4M2I * 0 0 AAAAGATAAGGGATAAA *
r003  0 ref 9 30 5H6M * 0 0 AGCTAA *
r004  0 ref 16 30 6M14N1I5M * 0 0 ATAGCTCTCAGC *
```

SAM files are TSV (tab-separated-values) files and begin with an optional header. The header consists of multiple lines, starting with an `@` character, each line is a record. Each record starts with its identifier and is followed by tab-separated tags. Each tag in the header consists of a two-character identifier, followed by `:` followed by the value.

If present, the `@HD` record must be the first record and specifies the SAM version (tag VN) used in this file and the sort order (SO). The optional `@SQ` header records give the reference sequence names (tag SN) and lengths (tag LN). There also are other header record types.

The optional header section is followed by the alignment records. The alignment records are again tab-separated. There are 11 mandatory columns.
### Notes:

- The SAM standard talks about “queries”. In the context of read mapping, where the format originates, queries are reads.
- The SAM standard talks about “templates” and “segments”. In the case of paired-end and mate-pair mapping the template consists of two segments, each is one read. The template length is the insert size.
- Paired-end reads are stored as two alignments records with the same QNAME. The first and second mate are discriminated by the FLAG values.
- When the FLAG indicates that SEQ is reverse-complemented, then QUAL is reversed.
- Positions in the SAM file are 1-based. When read into a BamAlignmentRecord (see below), the positions become 0-based.
- The qualities must be stored as ASCII PHRED-encoded qualities.
- The query and reference names must not contain whitespace. It is common to trim query and reference ids at the first space.

There are many ambiguities, recommendations, and some special cases in the formats that we do not describe here. We recommend that you follow this tutorial, start working with the SAM and BAM formats and later read the SAM specification “on demand” when you need it.

The 11 mandatory columns are followed by an arbitrary number of optional tags. Tags have a two-character identifier followed by " :${TYPE} :", followed by the tag’s value.

BAM files store their header as plain-text SAM headers. However, they additionally store the name and length information about the reference sequences. This information is mandatory since in BAM, the alignment records only contain the numeric ids of the reference sequences. Thus, the name is stored outside the record in the header.

### A First Working Example

The following example shows an example of a program that reads the file with the path example.sam and prints its contents back to the user on stdout. If you want to try out this program then create a file with the sample SAM content from above and adjust the path "example.sam" in the program below to the path to your SAM file (e.g. "path/to/my_example.sam”).

```cpp
#include <iostream>
#include <seqan/bam_io.h>

int main()
{
    // Open input stream, BamStream can read SAM and BAM files.
    seqan::BamStream bamStreamIn("example.sam");
    // Open output stream, "-" means stdin on if reading, else stdout.
    return 0;
}
```
The program first opens a **BamStream** for reading, then one for writing. Note that **BamStream** automatically guesses the file type from the file contents when reading and from the file name when writing. You can also force a format using **BamStream’s constructor**. You can read from stdin and write to stdout using "-" as the file name.

The header is automatically read when a **BamStream** is opened. After the header has been read, it is copied over into the output stream. Then, the input stream is read record by record and written out to the output stream. Note that the header is written out automatically before the first alignment record is written.

The alignment records are read into **BamAlignmentRecord** objects which we will focus on below.

Note that the example above is missing error handling. This means that if the input format is ill-formed, error return codes are not handled appropriately and the program might do something unexpected in the case of an error.

For example, if the file contains trailing empty lines, the program will loop indefinitely as can be seen in the shell output below:

```
# tutorial_basic_sam_bam_io_example1
@HD VN:1.3 SO:coordinate
@SQ SN:ref LN:45
@SQ SN:ref2 LN:40
r001 163 ref 7 30 8M4I4M1D3M = 37 39 TTAGATAAGAGGATACTG * XX:B:S,12561,2,20,112
r002 0 ref 9 30 1S2I6M1P111P114M21 * 0 0 AAAAGATAAGGGATAAA *
...```

We can fix this problem by introducing error handling. The `readRecord` call returns a status code different from 0, indicating an error because an empty line does not form a valid SAM record line. However, it stops processing as soon as an erroneous record is detected which makes the call to `atEnd` return `false` and run in an infinite loop.

In Assignment 1, we will add error handling to the program.

**Assignment 1**

**Adding Error Handling**

**Type** Review

**Objective** Add error handling using the hints below.
Hints The functions `readRecord` and `writeRecord` return a status code `int`, 0 on success, 1 on errors. The function `isGood` checks whether the state of a `BamStream` is erroneous.

Solution

```cpp
#include <iostream>
#include <seqan/bam_io.h>

int main()
{
    // Open input stream, BamStream can read SAM and BAM files.
    seqan::BamStream bamStreamIn("example.sam");
    if (!isGood(bamStreamIn))
    {
        std::cerr << "ERROR: Could not open example.sam!\n"
        return 1;
    }
    // Open output stream, "-" means stdin on if reading, else stdout.
    seqan::BamStream bamStreamOut("-", seqan::BamStream::WRITE);
    // Copy header. The header is automatically written out before
    // the first record.
    bamStreamOut.header = bamStreamIn.header;

    seqan::BamAlignmentRecord record;
    while (!atEnd(bamStreamIn))
    {
        if (readRecord(record, bamStreamIn) != 0)
        {
            std::cerr << "ERROR: Could not read record!\n"
            return 1;
        }
        if (writeRecord(bamStreamOut, record) != 0)
        {
            std::cerr << "ERROR: Could not write record!\n"
            return 1;
        }
    }
    return 0;
}
```

The Class `BamAlignmentRecord`

The class `BamAlignmentRecord` stores one alignment record in a SAM or BAM file. The class gives a in-memory representation that (1) is independent of whether it comes from/goes to a SAM or BAM file, (2) at the same time follows both formats closely, (3) allows for efficient storage and usage in C++, and (4) integrates well with the rest of the SeqAn library.

The following definition gives an overview that annotate which fields are available, the field types, and how they map to the SAM and BAM fields. Note that we use the `CigarElement` class to store entries in the CIGAR string.

```cpp
namespace seqan {

class BamAlignmentRecord
{
    public:
        CharString qName; // QNAME
        __uint16 flag; // FLAG

```
The static members INVALID_POS, INVALID_REFID, and INVALID_LEN store sentinel values for marking positions, reference sequence ids, and lengths as invalid or N/A.

An important related type is the enum BamFlags that provides constants for bit operations on the flag field. The functions hasFlagAllProper, hasFlagDuplicate, hasFlagFirst, hasFlagLast, hasFlagMultiple, hasFlagNextRC, hasFlagNextUnmapped, hasFlagQCNoPass, hasFlagRC, hasFlagSecondary, hasFlagUnmapped, and hasFlagSupplementary allow for easy reading of flags.

For example, the following loop sums up the length of the sequences that did not align:

```cpp
seqan::BamAlignmentRecord record;
unsigned lenSum = 0;
while (!atEnd(bamStreamIn))
    if (hasFlagUnmapped(record))
        lenSum += length(record.seq);
```

Counting Records

**Type Review**

**Objective** Extend the result of Assignment 1 by counting the number of unmapped reads.

**Hints** Use the function hasFlagUnmapped.

**Solution**

```cpp
#include <iostream>
#include <seqan/bam_io.h>

int main() {

    // Open input stream, BamStream can read SAM and BAM files.
    seqan::BamStream bamStreamIn("example.sam");
    if (!isGood(bamStreamIn))
        { std::cerr << "ERROR: Could not open example.sam!\n";
          return 1;
        }

    // Open output stream, "-" means stdin on if reading, else stdout.
    seqan::BamStream bamStreamOut("-", seqan::BamStream::WRITE);
    // Copy header. The header is automatically written out before
// the first record.
bamStreamOut.header = bamStreamIn.header;

unsigned numUnmappedReads = 0;
seqan::BamAlignmentRecord record;
while (!atEnd(bamStreamIn))
{
    if (readRecord(record, bamStreamIn) != 0)
    {
        std::cerr << "ERROR: Could not read record!\n"
        return 1;
    }
    if (hasFlagUnmapped(record))
        numUnmappedReads += 1;
}
std::cout << "Number of unmapped reads: " << numUnmappedReads << "\n";
return 0;
}

The Classes BamHeader and BamHeaderRecord

The header information is stored in the class BamHeader. This class gives a unified in-memory representation for SAM and BAM files.

The class has two members: records and sequenceInfos. We will focus on sequenceInfos here. sequenceInfos is a String of Pair objects. The first entry of the pair is a CharString with the sequence name and the second entry is a _int32 with the sequence length. Note that the @SQ header lines in the header and the sequenceInfos fields are not kept in sync automatically.

The following example program prints the sequences and lengths from a BAM file.

```cpp
#include <iostream>
#include <seqan/bam_io.h>

int main()
{
    seqan::BamStream bamInStream("example.bam");

    for (unsigned i = 0; i < length(bamInStream.header.sequenceInfos); ++i)
        std::cout << bamInStream.header.sequenceInfos[i].i1 << 't'
                  << bamInStream.header.sequenceInfos[i].i2 << 'n';

    return 0;
}
```

Note that this is only guaranteed to work for BAM files because this information is not mandatory in SAM files and might be missing. When writing files, you have to fill the sequenceInfos string appropriately before writing any record.

**Tip:** Building Ref-ID Mappings Using sequenceInfos.

The following example gives a typical example for using the sequenceInfos member: You want to post-process a BAM file together with the reference FASTA file. The sequences in the FASTA file are the same but their order may have changed. For example, because the FASTA file from the mapping step has been generated from the chromosomes by concatenation in a different order than the currently present one.
Assignment 3

Generating SAM From Scratch

Type Application

Objective Write a program that prints a SAM file, including headers @HD and @SQ. The content should be all 12-mers of the reference sequence "CCCGATGAGCACACGATCACACGATGACA", called "REF". The name should be "REF_${begin pos}_${end pos}". You only have to fill the members qId, rID, beginPos, cigar, and flag (set flag to 0).

Hints You can convert integers into strings using the <sstream> STL header.

The first lines of the result should read as follows:

@HD VN:1.4
@SQ SN:REF LN:29
REF_0_12 0 REF 1 * 12= * 0 * CCCGATGAGCAC
REF_1_13 0 REF 2 * 12= * 0 * CCGATGAGCAC
REF_2_14 0 REF 3 * 12= * 0 * CGATGAGCAC
REF_3_15 0 REF 4 * 12= * 0 * GATGACACC

Solution

#include <iostream>
#include <sstream>
#include <seqan/bam_io.h>
```cpp
int main()
{
    seqan::Dna5String ref = "CCGATGAGCACGATCACGATGACA";
    std::stringstream ss;

    // Build header.
    seqan::BamHeader header;

    // Fill sequenceInfos.
    resize(header.sequenceInfos, 1);
    header.sequenceInfos[0].i1 = "REF";
    header.sequenceInfos[0].i2 = length(ref);

    // Fill header records.
    resize(header.records, 2);
    // @HD header.
    header.records[0].type = seqan::BAM_HEADER_FIRST;
    resize(header.records[0].tags, 1);
    header.records[0].tags[0].i1 = "VN:";
    header.records[0].tags[0].i2 = "1.4";
    // @SQ header.
    header.records[1].type = seqan::BAM_HEADER_REFERENCE;
    resize(header.records[1].tags, 2);
    // @SQ header, tag/value: SN:REF
    header.records[1].tags[0].i1 = "SN:";
    header.records[1].tags[0].i2 = "REF";
    // @SQ header, tag/value: LN:30
    header.records[1].tags[1].i1 = "LN:";
    ss << length(ref);
    header.records[1].tags[1].i2 = ss.str();

    // Write out records.
    seqan::BamStream bamStream("-");
    bamStream.header = header;
    seqan::BamAlignmentRecord record;

    for (unsigned i = 0; i + 12 - 1 < length(ref); ++i)
    {
        clear(record);
        // Set members that are the same for all records.
        record.rID = 0;
        record.flag = 0;
        resize(record.cigar, 1);
        record.cigar[0].operation = '=';
        record.cigar[0].count = 12;

        ss.str(""");
        ss.clear();
        // The query name is REF_${START}_${END}.
        ss << "REF_" << i << "_" << (i + 12);
        record.qName = ss.str();
        // Set position.
    }
```
record.beginPos = i;
// Set sequence.
record.seq = infix(ref, i, i + 12);

writeRecord(bamStream, record);
}

return 0;
}

Accessing the Tags

As seen above, accessing the header tags is simple since it is a string of tag/value pairs. The whole header is completely read, parsed, and converted into this structure when the file is opened. The header is expected to be small, especially when compared to the rest of the file, and thus the time and memory spent is neglectable.

The alignment record tags are a different story. The tags only contain auxiliary information that are not of interest for all use cases. Always parsing the tags would not be in agreement with C++’s and SeqAn’s device “you only pay for what you use”, especially for BAM files that are expected to contain millions of records. Also, the tags of the alignment records are typed, e.g. NM:i:10 is an integer tag named "NM" with the value 10.

Thus, the following strategy is used. Alignment record tags from BAM files are copied byte-wise into the tag member of BamAlignmentRecord in a verbatim fashion. When reading from SAM, the tags are converted into format used by BAM tags.

Then, you can use the BamTagsDict class to access the the tag list of a record in a dictionary-like fashion. This class also performs the necessary casting when reading and writing tag list entries.

BamTagsDict acts as a wrapper around the tags member (which is of type CharString) of a BamAlignmentRecord:

```cpp
seqan::BamAlignmentRecord record;
seqan::BamTagsDict tagsDict(record.tags);
```

We can add a tag using the function setTagValue. When setting an already existing tag’s value, its value will be overwritten. Note that in the following, we give the tags value in SAM format because it is easier to read, although they are stored in BAM format internally.

```cpp
setTagValue(tagsDict, "NM", 2); // => tags: "NM:i:2"
setTagValue(tagsDict, "NH", 1); // => tags: "NM:i:2 NH:i:1"
setTagValue(tagsDict, "NM", 3); // => tags: "NM:i:3 NH:i:1"
```

The first parameter to setTagValue is the BamTagsDict, the second one is a two-character string with the key, and the third one is the value. Note that the type of tag entry will be taken automatically from the type of the third parameter.

Reading values is slightly more complex because we have to handle the case that the value is not present. First, we get the index of the tag in the tag list.

```cpp
unsigned myIdx = 0;
bool keyFound = findTagKey(myIdx, tagsDict, "NH");
if (keyFound)
    std::cerr << "ERROR: Unknown key!\n";
```

Then, we can read the value from the BamTagsDict using the function extractTagValue.
int valInt = 0;
bool ok = extractTagValue(valInt, tagsDict, myIdx);
if (ok)
    std::cerr << "ERROR: There was an error extracting NH from tags!\n";

The function returns a bool that is true on success and false otherwise. The extraction can fail if the index is out of bounds or the value in the dictionary cannot be cast to the type of the first parameter.

The value in the tags dictionary will be casted to the type of the first parameter (result parameter) of extractTagValue:

short valShort = 0;
extractTagValue(valShort, tagsDict, myIdx);

Assignment 4

Writing Tags

Type Review

Objective Modify the solution of Assignment 3 to also write the "NH" tag. This tag stores an int value that is the number of records for this query. In our case, the value is always 1.

The first lines of the result should read as follows:

@HD VN:1.4
@SQ SN:REF LN:29
REF_0_12 0 REF 1 * 12= * 0 * CCCGATGAGCAC * NH:i:1
REF_1_13 0 REF 2 * 12= * 0 * CCGATGAGCAC * NH:i:1
REF_2_14 0 REF 3 * 12= * 0 * CGATGAGCACAC * NH:i:1
REF_3_15 0 REF 4 * 12= * 0 * GATGAGCACACG * NH:i:1

Solution

#include <iostream>
#include <sstream>
#include <seqan/bam_io.h>

int main()
{
    seqan::Dna5String ref = "CCCGATGAGCACACGATCACACGATGACA";
    std::stringstream ss;

    // Build header.
    seqan::BamHeader header;

    // Fill sequenceInfos.
    resize(header.sequenceInfos, 1);
    header.sequenceInfos[0].i1 = "REF";
    header.sequenceInfos[0].i2 = length(ref);

    // Fill header records.
    resize(header.records, 2);
    // @HD header.
    header.records[0].type = seqan::BAM_HEADER_FIRST;
    resize(header.records[0].tags, 1);
    // @HD header, tag/value: VN:1.4.
    header.records[0].tags[0].i1 = "VN";
header.records[0].tags[0].i2 = "1.4";
// @SQ header.
header.records[1].type = seqan::BAM_HEADER_REFERENCE;
resize(header.records[1].tags, 2);
// @SQ header, tag/value: SN:REF
header.records[1].tags[0].i1 = "SN";
header.records[1].tags[0].i2 = "REF";
// @SQ header, tag/value: LN:30
header.records[1].tags[1].i1 = "LN";
ss << length(ref);
header.records[1].tags[1].i2 = ss.str();

// Write out records.
sequa::BamStream bamStream("-", sequa::BamStream::WRITE);
bamStream.header = header;

sequa::BamAlignmentRecord record;
for (unsigned i = 0; i + 12 - 1 < length(ref); ++i)
{
    clear(record);
    // Set members that are the same for all records.
    record.rID = 0;
    record.flag = 0;
    resize(record.cigar, 1);
    record.cigar[0].operation = '=';
    record.cigar[0].count = 12;
    // Write "NH" tag.
    sequa::BamTagsDict tagsDict(record.tags);
    setTagValue(tagsDict, "NH", 1);

    ss.str("\n");
    ss.clear();
    // The query name is REF_${START}_${END}.
    ss << "REF_" << i << "_" << (i + 12);
    record.qName = ss.str();
    // Set position.
    record.beginPos = i;
    // Set sequence.
    record.seq = infix(ref, i, i + 12);

    writeRecord(bamStream, record);
}

return 0;
}

Congratulations, you have now learned to read and write SAM and BAM files.

Next Steps

- Read the SAM Specification (pdf).
- Continue with the Tutorial.
2.1.20 VCF I/O

Learning Objective In this tutorial, you will learn how to use the high-level interface \texttt{VcfStream} class to read and write VCF files.

Difficulty Average

Duration 1 h (45 min if you know the VCF format)

Prerequisites \texttt{Sequences}, exposure to the VCF format

This tutorial deals with how to easily read and write VCF files using the \texttt{VcfStream} class. It starts out with a quick reminder on the structure of VCF files and will then continue with how to read and write VCF files and access the tags of a record.

Important: Note that this tutorial is targeted at readers that already know about the VCF format. If you do not know about the VCF format yet then this tutorial will be harder for your to understand. The 1000 Genomes project hosts the VCF format specification (v4.1).

The VCF format allows storing genomic variants of individuals with respect to a reference. The general file structure starts with (1) meta-information lines starting with `##`, one (2) header line giving the names of the individuals, and (3) an arbitrary number of records.

The information of (1) and (2) will be read and written together as the “header” of the file. For simple variants such as SNPs and small indels, each record corresponds to a variant. More complex variants can be stored in multiple records (see the VCF standard on “breakends” for more information).

The \texttt{vcf/io} module of SeqAn allows the record-wise reading and writing to VCF files. Since the structure of the fields in the VCF format often is very complex and the format undergoes changes in this respect, SeqAn only offers basic parsing functionality: The position is stored as a 0-based integer, reference names are stored in a reference name store (similar as in the \texttt{Basic SAM and BAM I/O Tutorial}), and the quality is stored as a \texttt{float} value.

The remaining fields have to be parsed from and composed as strings in the user’s application.

VCF File Structure

This section gives a very brief overview of the VCF file structure. For more details, see the VCF format specification (v4.1).

The following is an example of a VCF file:
The file starts with meta information lines (starting with `##`) with a key/value structure. The most important lines have the keys `contig`, `INFO`, `FILTER`, and `FORMAT`.

**contig** Lines with this key list the contigs of the reference genome.

**INFO** These lines give valid keys (and the format of the values) for the INFO column.

**FILTER** Valid values of the FILTER column.

**FORMAT** Valid entries for the INFO column.

The meta information lines are followed by the header line which gives the names of the first 9 columns which are always the same (CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, FORMAT) and a non-empty list of sample names. The columns are separated by spaces.

The header line is followed by the records which contains a value for each column in the header.

**CHROM** Name of the chromosome/reference sequence that the variant lies on.

**POS** The 1-based position of the variant.

**ID** A name of the variant. . is used if no name is available.

**REF** The value of the reference allele.

**ALT** The alternate allele values (multiple values are comma-separated).

**QUAL** Quality value of the call (float).

**FILTER** A value for the filter result (given in a FILTER meta information line).

**INFO** Information about a variant.

**FORMAT** Colon-separated list of entries that are found for each variant.

The 9 mandatory columns are followed by as many columns as there are individual. For each individual, there is a colon-separated list of values in the order given in the FORMAT cell.

**Tip:** 1-based and 0-based positions.
There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval \([1,000; 2,000]\) starts at character 1,000 and ends at character 2,000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.

2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be \([999; 2,000)\) in this case.

In text representations, such as VCF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals. When fields are reads as text, numbers are not translated, of course.

---

**A First Working Example**

The following example shows an example of a program that reads the file with the path `example.vcf` and prints its contents back to the user on stdout. If you want to try out this program then create a file with the sample VCF content from above and adjust the path "example.vcf" in the program below to the path to your VCF file (e.g. "path/to/my_example.vcf").

```c
#include <seqan/basic.h>
#include <seqan/vcf_io.h>

int main()
{
    // Open input stream.
    seqan::VcfStream vcfIn("example.vcf");
    // Open output stream, filename "-" means stdout.
    seqan::VcfStream vcfOut("-", seqan::VcfStream::WRITE);

    // Copy over header.
    vcfOut.header = vcfIn.header;

    // Read the file record by record.
    seqan::VcfRecord record;
    while (!atEnd(vcfIn))
    {
        readRecord(record, vcfIn);
        writeRecord(vcfOut, record);
    }

    return 0;
}
```

The program first opens a VcfStream for reading, then one for writing. You can read from stdin and write to stdout using "-" as the file name.

The header is automatically read when a VcfStream is opened. After the header has been read, it is copied over into the output stream. Then, the input stream is read record by record and written out to the output stream. Note that the header is written out automatically before the first variant record is written.

The alignment records are read into VcfRecord objects which we will focus on below.

Note that the example above is missing error handling. This means that if the input format is ill-formed, error return codes are not handled appropriately and the program might do something unexpected in the case of an error. We will fix this in Assignment 1.

---

2.1. Tutorial 199
You can see the output of the program below when called with the input file from above.

```plaintext
##fileformat=VCFv4.1
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb2da,species="Homo sapiens",taxonomy=x>
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">

#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA00001 NA00002 NA00003
20 14370 rs6054257 G A 29 PASS NS=3;DP=14;AF=0.5;DB;H2 GT:GQ:DP:HQ 0|0:48:1:51,51 1|0:48:8:51,51 1/1:43:5:.,.
20 17330 . T A 3 q10 NS=3;DP=11;AF=0.017 GT:GQ:DP:HQ 0|0:49:3:58,50 0|1:3:5:65,3 0/0:41:3
20 1110696 rs6040355 A G,T 67 PASS NS=2;DP=10;AF=0.333,0.667;AA=T;DB GT:GQ:DP:HQ 1|2:21:6:23,27 2|1:2:0:18,2 2/2:35:4
20 1230237 . T . 47 PASS NS=3;DP=13;AA=T GT:GQ:DP:HQ 0|0:54:7:56,60 0|0:48:4:51,51 0/0:61:2
20 1234567 microsat1 GTC G,GTCT 50 PASS NS=3;DP=9;AA=G GT:GQ:DP 0/1
```

To add error handling, we have to check return values. The `readRecor` call returns a status code different from 0, indicating an error.

In Assignment 1, we will add error handling to the program.

**Assignment 1**

**Adding Error Handling**

**Type Review**

**Objective** Add error handling using the hints below.

**Hints** The functions `readRecord` and `writeRecord` return a status code int, 0 on success, 1 on errors. The function `isGood` checks whether the state of a `VcfStream` is erroneous.

**Solution**

```c
#include <seqan/basic.h>
#include <seqan/vcf_io.h>

int main()
{
    // Open input stream
    seqan::VcfStream vcfIn("example.vcf");
    if (!isGood(vcfIn))
    {
        std::cerr << "ERROR: Could not open example.vcf\n";
        return 1;
    }
    // Open output stream, filename "-" means stdout.
    seqan::VcfStream vcfOut("-", seqan::VcfStream::WRITE);
```
// Copy over header.
vcfOut.header = vcfIn.header;

// Read the file record by record.
seqan::VcfRecord record;
while (!atEnd(vcfIn))
{
    if (readRecord(record, vcfIn) != 0)
    {
        std::cerr << "ERROR: Problem reading from example.vcf
";
        return 1;
    }
    if (writeRecord(vcfOut, record) != 0)
    {
        std::cerr << "ERROR: Problem writing to stdout.
";
        return 1;
    }
}
return 0;

The Class VcfRecord

The class VcfRecord stores one record in a VCF file. It is best explained by its definition. Note how most fields are represented by strings:

namespace seqan {

class VcfRecord
{
public:
    __int32 rID; // CHROM
    __int32 beginPos; // POS
    CharString id; // ID
    CharString ref; // REF
    CharString alt; // ALT
    float qual; // QUAL
    CharString filter; // FILTER
    CharString info; // INFO
    CharString format; // FORMAT
    StringSet<CharString> genotypeInfos; // <individual1> <individual2> ..

    // Constants for marking reference id and position as invalid.
    static const __int32 INVALID_REFID = -1;
    static const __int32 INVALID_POS = -1;
    // This function returns the float value for "invalid quality".
    static float MISSING_QUAL();
};

} // namespace seqan

The static members INVALID_POS, INVALID_REFID store sentinel values for marking positions and reference sequence ids as invalid. The static function MISSING_QUAL() returns the IEEE float “NaN” value. In C++11, there will be a std::nan() function but for now, we need this here.
Assignment 2

Counting Records

Type Review

Objective Change the result of Assignment 1 by counting the number of variants for each chromosome/contig instead of writing out the records.

Hints The header contains the sequence names in `vcfIn.header.sequenceNames`. You can use the length of this `StringSet` of `CharString` to get the number of contigs.

Solution

```cpp
#include <seqan/basic.h>
#include <seqan/vcf_io.h>

int main()
{
    // Open input stream
    seqan::VcfStream vcfIn("example.vcf");
    if (!isGood(vcfIn))
    {
        std::cerr << "ERROR: Could not open example.vcf\n"
                    " return 1;
    }
    // Open output stream, filename "-" means stdout.
    seqan::VcfStream vcfOut("-", seqan::VcfStream::WRITE);
    // Copy over header.
    vcfOut.header = vcfIn.header;
    // Get array of counters.
    seqan::String<unsigned> counters;
    resize(counters, length(vcfIn.header.sequenceNames), 0);
    // Read the file record by record.
    seqan::VcfRecord record;
    while (!atEnd(vcfIn))
    {
        if (readRecord(record, vcfIn) != 0)
        {
            std::cerr << "ERROR: Problem reading from example.vcf\n"
                        " return 1;
        }
        // Register record with counters.
        counters[record.rID] += 1;
    }
    // Print result.
    std::cout << "VARIANTS ON CONTIGS\n"
               for (unsigned i = 0; i < length(vcfIn.header.sequenceNames); ++i)
               {
                std::cout << vcfIn.header.sequenceNames[i] << '\t'
                           << counters[i] << '\n';
               }
    return 0;
}
```

The output is
The Classes VcfHeader and VcfHeaderRecord

The header information is stored in the class VcfHeader. Objects of this class store the information present in the VCF meta information and header lines.

The class has three members: sequenceNames, sampleNames, and headerRecords. sequenceNames and sampleNames are StringSets of CharStrings. The member rID of VcfRecord points into sequenceNames and gives the reference sequence. The genotypeInfos member of VcfRecord has the same number of entries as sampleNames and record.genotypeInfos[i] contains the variant information for sampleNames[i].

When writing VCF files, you have to fill these three members of VcfHeader before writing any record.

Assignment 3

Generating VCF From Scratch

Type Application

Objective Write a program that prints the VCF file from above.

Hints You can convert integers into strings using the <sstream> STL header.

```cpp
#include <sstream>
// ...
std::stringstream ss;
ss << 10;
seqan::CharString str = ss.str(); // => == "10"
// To reset ss, we need two calls:
ss.str(""); // Remove contents.
ss.clear(); // Reset any error bits.
```

Solution

```cpp
#include <seqan/basic.h>
#include <seqan/vcf_io.h>
#include <sstream>

int main()
{
    seqan::VcfStream out("-", seqan::VcfStream::WRITE);
    // Fill sequence names.
    appendValue(out.header.sequenceNames, "20");

    // Fill sample names.
    appendValue(out.header.sampleNames, "NA00001");
    appendValue(out.header.sampleNames, "NA00002");
    appendValue(out.header.sampleNames, "NA00002");

    // Write out headers.
    // This is somewhat tedious.
    appendValue(out.header.headerRecords, seqan::VcfHeaderRecord("fileformat", "VCFv4.1"));
    appendValue(out.header.headerRecords, seqan::VcfHeaderRecord("fileDate", "20090805"));
```
SeqAn Manual, Release 1.4.2

```cpp
// Write out the records.
seqan::VcfRecord record;
record.rID = 0;
record.beginPos = 14369;
record.id = "rs6054257";
record.ref = "G";
record.alt = "A";
record.qual = 29;
record.filter = "PASS";
record.info = "NS=3;DP=14;AF=0.5;DB;H2";
record.format = "GT:GQ:DP:HQ";
appendValue(record.genotypeInfos, "0|0:48:1:51,51");
appendValue(record.genotypeInfos, "1|0:48:8:51,51");
appendValue(record.genotypeInfos, "1/1:43:5:.,.");
if (writeRecord(out, record) != 0)
    std::cerr << "ERROR: Problem writing output file."
return 0;
```

Next Steps

- Continue with the Tutorial
2.1.21 BED I/O

Learning Objective In this tutorial, you will learn how to use the high-level interface BedStream class to read and write BED files.

Difficulty Average

Duration 45min

Prerequisites Exposure to the BED format is useful.

This tutorial deals with how to easily read and write BED files using the BedStream class. It starts out with a quick reminder on the structure of BED files and will then continue with how to read and write BED files.

Originally, the BED format was designed for storing annotation tracks in the UCSC genome browser. Such an annotation track consists of multiple annotation records. Each annotation adds some meta information to a genomic interval (an interval with begin/end position on a contig/chromosome) The original specification of the format can be found in the UCSC Genome Browser FAQ.

The BED format is a TSV format and contains 12 columns. The first three column specify a genomic region (contig/chromosome name, begin, and end position) and the remaining columns contain additional information. The full format will be described below.

Since genomic intervals are very useful and because there were many tools for manipulating BED files (sorting, intersecting intervals etc.), many other authors and projects created variants of the BED format. Usually, three or more columns have the same meaning as in BED and are followed by other, arbitrary tab-separated columns with additional annotation information. The “full” BED format is then called BED12, and BED3, BED4, BED5, and BED6 use the first 3-6 columns and keep the remaining information as data.

BED files can be manipulated using standard Unix tools such as sed, awk, and sort. There also is the bedtools suite with additional functionality.

The SeqAn module bed_io allows the reading and writing of BED files.

BED File Structure

The following is an example of a BED file:

<table>
<thead>
<tr>
<th>chr</th>
<th>start</th>
<th>end</th>
<th>name</th>
<th>score</th>
<th>chrom</th>
<th>start</th>
<th>end</th>
<th>id</th>
<th>name</th>
<th>score</th>
<th>chrom</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr1</td>
<td>66999</td>
<td>824</td>
<td>NM_032291</td>
<td>0</td>
<td>+</td>
<td>67000000</td>
<td>167208778</td>
<td>0</td>
<td>25</td>
<td>227,64,25,73,57,55,17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chr1</td>
<td>48998</td>
<td>526</td>
<td>NM_032785</td>
<td>0</td>
<td>-</td>
<td>4899984450</td>
<td>89489468</td>
<td>0</td>
<td>14</td>
<td>1439,27,97,163,153,11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chr1</td>
<td>16767</td>
<td>166</td>
<td>NM_018090</td>
<td>0</td>
<td>+</td>
<td>1676725616785385</td>
<td>0</td>
<td>8</td>
<td>182,101,105,82,109,17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The meaning of the columns are as follows:

ref (1) Name of the reference sequence.

beginPos (2) Begin position of the interval.

endPos (3) End position of the interval.

name (4) Name of the interval.

score (5) A score, could also be in scientific notation or several values in a comma/colon-separated list.

strand (6) The strand of the feature, + for forward, − for reverse, . for unknown/dont-care.

thickBegin (7) Begin position where the feature is drawn thick in the UCSC browser.

thickEnd (8) End position where the feature is drawn thick in the UCSC browser.

itemRgb (9) Comma-separated triple with RGB values (0..255 each)

blockCount (10) The number of blocks (exons) in the BED line (for the UCSC browser).

blockStarts (11) Comma-separated list with begin positions of exons (for the UCSC browser, should be consistent with blockCount).

blockSizes (12) Comma-separated list with exon lists (for the UCSC browser, should be consistent with blockCount).

Tip: 1-based and 0-based positions.

There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval [1,000; 2,000] starts at character 1,000 and ends at character 2,000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.

2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be [999; 2,000) in this case.

In text representations, such as GFF and GTF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals. BED is a text format using 0-based positions.

A First Working Example

The following example shows an example of a program that reads the file with the path example.bed and prints its contents back to the user on stdout. If you want to try out this program then create a file with the sample BED content from above and adjust the path "example.bed" in the program below to the path to your BED file (e.g. "path/to/my_example.bed").

```cpp
#include <seqan/basic.h>
#include <seqan/bed_io.h>

int main()
{
    // Open input stream.
    seqan::BedStream bedIn("example.bed");
```
The program first opens a `BedStream` for reading, then one for writing. You can read from stdin and write to stdout using "-" as the file name.

The member `sequenceNames` of your `BedStream` object `bedIn` contains the names of the reference sequences that have been seen in records so far. This `StringSet` of `CharString` thus gets new elements as you read the BED file. For the translation between reference names and numeric ids, a cache is used. The function `addSequenceName` can be used to register the sequence name with the `bedOut` stream. This will also update the cache.

The BED records are read into `BedRecord` objects which we will focus on below. In this case, we use the `Bed3Record` specialization of the `BedRecord` class.

**Tip:** BED records and additional data.

This means that the first three columns are read and interpreted and available in the class members. The remaining data is stored in the `data` member variable of the record. This means that the data stored after the first three columns could be empty or of an arbitrary format.

Note that the example above is missing error handling. This means that if the input format is ill-formed, error return codes are not handled appropriately and the program might do something unexpected in the case of an error. We will fix this in **Assignment 1**.

You can see the output of the program below when called with the input file from above.

To add error handling, we have to check return values. The `readRecord` call returns a status code different from 0, indicating an error.

In **Assignment 1**, we will add error handling to the program.

**Assignment 1**

Adding Error Handling
Type Review

Objective Add error handling using the hints below.

Hints The functions `readRecord` and `writeRecord` return a status code `int`, 0 on success, 1 on errors. The function `isGood` checks whether the state of a `BedStream` is erroneous.

Solution

```cpp
#include <seqan/basic.h>
#include <seqan/bed_io.h>

int main()
{
    // Open input stream
    seqan::BedStream bedIn("example.bed");
    if (!isGood(bedIn))
    {
        std::cerr << "ERROR: Could not open example.bed\n";
        return 1;
    }

    // Open output stream, filename "-" means stdout.
    seqan::BedStream bedOut("-", seqan::BedStream::WRITE);

    // Read the file record by record.
    seqan::BedRecord<seqan::Bed3> record;
    while (!atEnd(bedIn))
    {
        if (readRecord(record, bedIn) != 0)
        {
            std::cerr << "ERROR: Problem reading from example.bed\n";
            return 1;
        }

        // If record is on a sequence that is not known to bedOut yet then we
        // have to make it known there.
        if (record.rID >= (int)length(bedOut.sequenceNames))
            addSequenceName(bedOut, record.ref);

        if (writeRecord(bedOut, record) != 0)
        {
            std::cerr << "ERROR: Problem writing to stdout.\n";
            return 1;
        }
    }

    return 0;
}
```

The Class `BedRecord`

The class `BedRecord` stores one record in a BED file. Note that there are various specializations, each storing a different number of fields. We show the quasi-definition of `Bed12Record` below. The other specializations have less fields.

```cpp
namespace seqan {

class BedRecord {
```
The static members INVALID_POS, INVALID_REFID store sentinel values for marking positions and reference sequence ids as invalid.

The member ref stores the contig/reference name of the genomic interval. This information is somewhat redundant with the rID member that is filled automatically when reading from a BedStream such that the BedStream’s sequenceNames[record.rID] == record.ref. Translating reference names to integers is useful in many applications.

When writing and record.rID == INVALID_REFID then record.ref is written out as the reference name and sequenceNames[record.rID] is written out otherwise. The user has to take care that record.rID is a valid reference id in this case.

**Assignment 2**

**Counting Records**

**Type Review**

**Objective** Change the result of Assignment 1 by counting the number of variants for each chromosome/contig instead of writing out the records.

**Solution**

```c++
#include <seqan/basic.h>
#include <seqan/bed_io.h>

int main()
{
    // Open input stream
    seqan::BedStream bedIn("example.bed");
    if (!isGood(bedIn))
    {
        std::cerr << "ERROR: Could not open example.bed\n";
    }
```
Assignment 3

Generating BED From Scratch

Type Application

Objective Write a program that prints the following BED file. Create BedRecord<Bed6> objects and write them to a BedStream using writeRecord().

Solution

return 1;
}

// Array of counters and sequence names.
seqan::String<unsigned> counters;
seqan::StringSet<seqan::CharString> seqNames;

// Read the file record by record.
while (!atEnd(bedIn))
{
    if (readRecord(record, bedIn) != 0)
    {
        std::cerr << "ERROR: Problem reading from example.bed"
                    << std::endl;
        return 1;
    }

    // Resize counters and write seqNames if necessary.
    if ((int)length(counters) <= record.rID)
    {
        resize(counters, record.rID + 1, 0);
        resize(seqNames, record.rID + 1);
    }

    if (counters[record.rID] == 0)
        seqNames[record.rID] = record.ref;

    // Register record with counters.
    counters[record.rID] += 1;
}

// Print result.
std::cout << "RECORDS ON CONTIGS"
           << "\n";
for (unsigned i = 0; i < length(seqNames); ++i)
    if (counters[i] > 0u)
        std::cout << seqNames[i] << \t << counters[i] << \n;

return 0;

The output is
RECORDS ON CONTIGS
chr1  5
```cpp
#include <seqan/basic.h>
#include <seqan/bed_io.h>
#include <sstream>

int main()
{
    seqan::BedStream out("-", seqan::BedStream::WRITE);

    // Add sequence names.
    addSequenceName(out, "chr7");

    // Write out the records.
    seqan::BedRecord<seqan::Bed6> record;
    record.rID = 0;
    record.beginPos = 127471195;
    record.endPos = 127472363;
    record.name = "Pos1";
    record.score = "0";
    record.strand = '+';
    if (writeRecord(out, record) != 0)
        std::cerr << "ERROR: Problem writing output file."
    ;

    record.rID = 0;
    record.beginPos = 127472362;
    record.endPos = 127473530;
    record.name = "Pos2";
    record.score = "0";
    record.strand = '+';
    if (writeRecord(out, record) != 0)
        std::cerr << "ERROR: Problem writing output file."
    ;

    return 0;
}
```

**Next Steps**

- Continue with the *Tutorial*. 

**Contents**

- GFF and GTF I/O
  - GFF File Structure
  - A First Working Example
    - Assignment 1
  - The Class GffRecord
    - Assignment 2
    - Assignment 3
  - GFF and GTF
  - Next Steps
2.1.22 GFF and GTF I/O

Learning Objective In this tutorial, you will learn how to use the high-level interface GffStream class to read and write GFF and GTF files.

Difficulty Average

Duration 45 min

Prerequisites Exposure to the GFF and GTF formats is useful.

This tutorial deals with how to easily read and write GFF and GTF files using the GffStream class. It starts out with a quick reminder on the structure of GFF and GTF files and will then continue with how to read and write GFF and GTF files.

The GFF and GTF formats are used for annotating genomic intervals (an interval with begin/end position on a contig/chromosome). GFF exist in versions 2 and 3 and GTF is sometimes called “GFF 2.5”. There are specifications for GFF 2, GFF 3, and GTF available elsewhere. GFF and GTF are TSV-based formats and in general have the same structure. The main difference is the underlying system/ontology for the annotation but also smaller differences in the format.

In this tutorial, we will focus on the format GFF 3 since it is the most current one with most complete tool support. The information of this tutorial can easily be translated to the other two formats.

The SeqAn module gff_io allows the reading and writing of the GFF and GTF formats.

Tip: Format Version Support in SeqAn

GffStream allows to read GFF files in version 2 and 3 and GTF files. For writing, only GFF 3 and GTF are supported at the moment.

GFF File Structure

The following is an example of a GFF 3 file:

```
ctg123 . gene 1000 9000 . + . ID=gene00001;Name=EDEN
ctg123 . TF_binding_site 1000 1012 . + . Parent=gene00001
ctg123 . mRNA 1050 9000 . + . ID=mRNA00001;Parent=gene00001
ctg123 . mRNA 1050 9000 . + . ID=mRNA00002;Parent=gene00001
ctg123 . mRNA 1300 9000 . + . ID=mRNA00003;Parent=gene00001
ctg123 . exon 1300 1500 . + . Parent=mRNA00003
ctg123 . exon 1050 1500 . + . Parent=mRNA00001,mRNA00002
ctg123 . exon 3000 3902 . + . Parent=mRNA00001,mRNA00002
ctg123 . exon 5000 5500 . + . Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon 7000 9000 . + . Parent=mRNA00001,mRNA00002,mRNA00003
```

ctg123 . CDS 1201 1500 . + 0 ID=cds00001;Parent=mRNA00001
ctg123 . CDS 3000 3902 . + 0 ID=cds00001;Parent=mRNA00001
ctg123 . CDS 5000 5500 . + 0 ID=cds00001;Parent=mRNA00001
ctg123 . CDS 7000 7600 . + 0 ID=cds00001;Parent=mRNA00001
ctg123 . CDS 1201 1500 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 3000 3902 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 5000 5500 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 7000 7600 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 1201 1500 . + 0 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 3000 3902 . + 0 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 5000 5500 . + 1 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 7000 7600 . + 1 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 3391 3902 . + 0 ID=cds00004;Parent=mRNA00003
ctg123 . CDS 5000 5500 . + 1 ID=cds00004;Parent=mRNA00003
ctg123 . CDS 7000 7600 . + 1 ID=cds00004;Parent=mRNA00003
The meaning of the columns are as follows:

**seq id (1)** Name of the reference sequence.

**source (2)** Free text field describing the source of the annotation, such as a software (e.g. “Genescan”) or a database (e.g. “Genebank”), “.” for none.

**type (3)** The type of the annotation.

**start (4)** The 1-based begin position of the annotation.

**end (5)** The 1-based end position of the annotation.

**score (6)** The score of the annotation, “.” for none.

**strand (7)** The strand of the annotation, “+” and “−” for forward and reverse strand, “.” for features that are not stranded.

**phase (8)** Shift of the feature regarding to the reading frame, one of “0”, “1”, “2”, and “.” for missing/dont-care.

**attributes (9)** A list of key/value attributes. For GFF 3, this is a list of key=value pairs, separated by semicolons (e.g. ID=cds00003;Parent=mRNA00003). For GTF and GFF 2, this is a list of tuples, separated by semicolon. The first entry gives the key, the following entries are values. Strings are generally enclosed in quotes (e.g. Target "HBA_HUMAN" 11 55 ; E_value 0.0003)

---

**Tip:** 1-based and 0-based positions.

There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval \([1,000; 2,000]\) starts at character 1,000 and ends at character 2,000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.

2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be \([999; 2,000)\) in this case.

In text representations, such as GFF and GTF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals.

---

### A First Working Example

The following example shows an example of a program that reads the file with the path `example.gff` and prints its contents back to the user on stdout. If you want to try out this program then create a file with the sample GFF content from above and adjust the path "example.gff" in the program below to the path to your GFF file (e.g. "path/to/my_example.gff").

```cpp
#include <seqan/basic.h>
#include <seqan/gff_io.h>

int main()
{
    // Open input stream.
    seqan::GffStream gffIn("example.gff");
    // Open output stream, filename "-" means stdout.
    seqan::GffStream gffOut("-", seqan::GffStream::WRITE);

    // Read the file record by record.
    seqan::GffRecord record;
```
while (!atEnd(gffIn))
{
    readRecord(record, gffIn);

    // If record is on a sequence that is not known to gffOut yet then we
    // have to make it known there.
    if (record.rID >= (int)length(gffOut.sequenceNames))
        addSequenceName(gffOut, record.ref);

    writeRecord(gffOut, record);
}

return 0;

The program first opens a GffStream for reading, then one for writing. You can read from stdin and write to stdout using "-" as the file name.

The member sequenceNames of your GffStream object gffIn contains the names of the reference sequences that have been seen in records so far. This StringSet of CharString thus gets new elements as you read the Gff file. For the translation between reference names and numeric ids, a cache is used. The function [dox:GffStream#addSequenceName addSequenceName can be used to register the sequence name with the gffOut stream. This will also update the cache.

Note that the example above is missing error handling. This means that if the input format is ill-formed, error return codes are not handled appropriately and the program might do something unexpected in the case of an error. We will fix this in Assignment 1.

You can see the output of the program below when called with the input file from above.

c tg123 . gene 1000 9000 . + . ID=gene00001;Name=EDEN
c tg123 . TF_binding_site 1000 1012 . + . ID=mRNA00001;Parent=gene00001
c tg123 . mRNA 1050 9000 . + . ID=mRNA00002;Parent=gene00001
c tg123 . mRNA 1300 9000 . + . ID=mRNA00003;Parent=gene00001
c tg123 . exon 1300 1500 . + . Parent=mRNA00003
ctg123 . exon 1050 1500 . + . Parent=mRNA00001,mRNA00002
c tg123 . exon 3000 3902 . + . Parent=mRNA00001,mRNA00002,mRNA00003
c tg123 . exon 5000 5500 . + . Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon 7000 9000 . + . Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . CDS 1201 1500 . + 0 ID=cds00001;Parent=mRNA00001
ctg123 . CDS 3000 3902 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 5000 5500 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 7000 7600 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 3301 3902 . + 0 ID=cds00002;Parent=mRNA00002
ctg123 . CDS 5000 5500 . + 1 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 7000 7600 . + 1 ID=cds00003;Parent=mRNA00003
ctg123 . CDS 3391 3902 . + 0 ID=cds00004;Parent=mRNA00003
ctg123 . CDS 5000 5500 . + 1 ID=cds00004;Parent=mRNA00003
ctg123 . CDS 7000 7600 . + 1 ID=cds00004;Parent=mRNA00003

t o add error handling, we have to check return values. The readRecord call returns a status code different from 0, indicating an error.

In Assignment 1, we will add error handling to the program.
Assignment 1

Adding Error Handling

Type Review

Objective Add error handling using the hints below.

Hints The functions readRecord and writeRecord return a status code `int`, 0 on success, 1 on errors. The function isGood checks whether the state of a GffStream is erroneous.

Solution

```cpp
#include <seqan/basic.h>
#include <seqan/gff_io.h>

int main()
{
    // Open input stream
    seqan::GffStream gffIn("example.gff");
    if (!isGood(gffIn))
    {
        std::cerr << "ERROR: Could not open example.gff\n";
        return 1;
    }
    // Open output stream, filename "-" means stdout.
    seqan::GffStream gffOut("-", seqan::GffStream::WRITE);

    // Read the file record by record.
    seqan::GffRecord record;
    while (!atEnd(gffIn))
    {
        if (readRecord(record, gffIn) != 0)
        {
            std::cerr << "ERROR: Problem reading from example.gff\n";
            return 1;
        }
        // If record is on a sequence that is not known to gffOut yet then we
        // have to make it known there.
        if (record.rID >= (int)length(gffOut.sequenceNames))
            addSequenceName(gffOut, record.ref);

        if (writeRecord(gffOut, record) != 0)
        {
            std::cerr << "ERROR: Problem writing to stdout.\n";
            return 1;
        }
    }
    return 0;
}
```

The Class GffRecord

The class `GffRecord` stores one record in a Gff file.
namespace seqan {

class GffRecord
{
public:
    CharString ref; // reference name
    __int32 rID; // index in sequenceNames of GffStream
    CharString source; // source free text descriptor
    CharString type; // type of the feature
    __int32 beginPos; // begin position of the interval
    __int32 endPos; // end position of the interval
    float score; // score of the annotation
    char strand; // the strand
    char phase; // one of '0', '1', '2', and '.'

    // The key/value list, split into a list of keys and values.
    StringSet<CharacterString> tagName;
    StringSet<CharacterString> tagValue;

    // Returns float value for an invalid score.
    static float INVALID_SCORE();

    // Constants for marking reference id and position as invalid.
    static const __int32 INVALID_IDX = -1;
    static const __int32 INVALID_POS = -1;
};
}

// namespace seqan

The static members INVALID_POS, INVALID_REFID store sentinel values for marking positions and reference sequence ids as invalid. The static function INVALID_SCORE() returns the IEEE float “NaN” value. In C++11, there will be a std::nan() function but for now, we need this here.

The member ref stores the contig/reference name of the genomic interval. This information is somewhat redundant with the rID member that is filled automatically when reading from a GffStream such that the GffStream’s sequenceNames[record.rID] == record.ref. Translating reference names to integers is useful in many applications.

When writing and record.rID == INVALID_REFID then record.ref is written out as the reference name and sequenceNames[record.rID] is written out otherwise. The user has to take care that record.rID is a valid reference id in this case.

Assignment 2

Counting Records

Type Review

Objective Change the result of Assignment 1 by counting the number of variants for each chromosome/contig instead of writing out the records.

Solution

#include <seqan/basic.h>
#include <seqan/gff_io.h>

int main()
{

// Open input stream
seqan::GffStream gffIn("example.gff");
if (!isGood(gffIn))
{
    std::cerr << "ERROR: Could not open example.gff\n";
    return 1;
}

// Array of counters and sequence names.
seqan::String<unsigned> counters;
seqan::StringSet<seqan::CharString> seqNames;

// Read the file record by record.
seqan::GffRecord record;
while (!atEnd(gffIn))
{
    if (readRecord(record, gffIn) != 0)
    {
        std::cerr << "ERROR: Problem reading from example.gff\n";
        return 1;
    }

    // Resize counters and write seqNames if necessary.
    if ((int)length(counters) <= record.rID)
    {
        resize(counters, record.rID + 1, 0);
        resize(seqNames, record.rID + 1);
    }
    if (counters[record.rID] == 0)
        seqNames[record.rID] = record.ref;

    // Register record with counters.
    counters[record.rID] += 1;
}

// Print result.
std::cout << "RECORDS ON CONTIGS\n"
for (unsigned i = 0; i < length(seqNames); ++i)
{
    if (counters[i] > 0u)
        std::cout << seqNames[i] << '	' << counters[i] << '\n';
}
return 0;

The output is
RECORDS ON CONTIGS
ctg123 23

Assignment 3

Generating GFF From Scratch

Type Application

Objective Write a program that prints the following GFF file. Create GffRecord objects and write them to a GffStream using writeRecord().
```cpp
#include <seqan/basic.h>
#include <seqan/gff_io.h>
#include <sstream>

int main()
{
    seqan::GffStream out("-", seqan::GffStream::WRITE);

    // Add sequence names.
    addSequenceName(out, "ctg123");

    // Write out the records.
    seqan::GffRecord record;
    record.rID = 0;
    record.source = "";
    record.type = "gene";
    record.beginPos = 999;
    record.endPos = 9000;
    record.strand = '+';
    record.score = seqan::GffRecord::INVALID_SCORE();
    appendValue(record.tagName, "ID");
    appendValue(record.tagValue, "gene0001");
    appendValue(record.tagName, "Name");
    appendValue(record.tagValue, "EDEN");
    if (writeRecord(out, record) != 0)
        std::cerr << "ERROR: Problem writing output file.";

    clear(record.tagName);
    clear(record.tagValue);

    record.rID = 0;
    record.source = "";
    record.type = "TF_binding_site";
    record.beginPos = 999;
    record.endPos = 1012;
    record.strand = '+';
    record.score = seqan::GffRecord::INVALID_SCORE();
    appendValue(record.tagName, "Parent");
    appendValue(record.tagValue, "gene0001");
    if (writeRecord(out, record) != 0)
        std::cerr << "ERROR: Problem writing output file.";

    return 0;
}
```

**GFF and GTF**

The class `GffStream` transparently reads files in both GFF and GTF format. When writing, you can select the output format with the third parameter to the constructor `GffStream` or the function `open`. When using `GffStream::GFF`, GFF 3 is used, when using `GffStream::GTF`, the GTF format. The default is to use `GffStream::GFF`. 
The following program converts a GFF file into a GTF file.

```cpp
#include <seqan/basic.h>
#include <seqan/gff_io.h>

int main()
{
    // Open input stream.
    seqan::GffStream gffIn("example.gff");
    // Open output stream, filename "-" means stdout.
    seqan::GffStream gffOut("-", seqan::GffStream::WRITE, seqan::GffStream::GTF);

    // Read the file record by record.
    seqan::GffRecord record;
    while (!atEnd(gffIn))
    {
        readRecord(record, gffIn);
        // If record is on a sequence that is not known to gffOut yet then we
        // have to make it known there.
        if (record.rID >= (int)length(gffOut.sequenceNames))
            addSequenceName(gffOut, record.ref);

        writeRecord(gffOut, record);
    }

    return 0;
}
```

Given the GFF file at the top, the output of the example above will look as follows:

ctg123 . gene 1000 9000 . + . ID "gene00001"; Name "EDEN";
ctg123 . TF_binding_site 1000 1012 . + . Parent "gene00001";
ctg123 . mRNA 1050 9000 . + . ID "mRNA00001"; Parent "gene00001";
ctg123 . mRNA 1300 9000 . + . ID "mRNA00003"; Parent "gene00001";
ctg123 . exon 1300 1500 . + . Parent "mRNA00003";
ctg123 . exon 1050 1500 . + . Parent "mRNA00001,mRNA00002";
ctg123 . exon 3000 3902 . + . Parent "mRNA00001,mRNA00003";
ctg123 . exon 5000 5500 . + . Parent "mRNA00001,mRNA00002,mRNA00003";
ctg123 . exon 7000 9000 . + . Parent "mRNA00001,mRNA00002,mRNA00003";
ctg123 . CDS 1201 1500 . + 0 ID "cds00001"; Parent "mRNA00001";
ctg123 . CDS 3000 3902 . + 0 ID "cds00001"; Parent "mRNA00001";
ctg123 . CDS 5000 5500 . + 0 ID "cds00001"; Parent "mRNA00001";
ctg123 . CDS 7000 7600 . + 0 ID "cds00001"; Parent "mRNA00001";
ctg123 . CDS 1201 1500 . + 0 ID "cds00002"; Parent "mRNA00002";
ctg123 . CDS 3000 3902 . + 0 ID "cds00002"; Parent "mRNA00002";
ctg123 . CDS 5000 5500 . + 0 ID "cds00002"; Parent "mRNA00002";
ctg123 . CDS 3301 3902 . + 0 ID "cds00003"; Parent "mRNA00003";
ctg123 . CDS 5000 5500 . + 1 ID "cds00003"; Parent "mRNA00003";
ctg123 . CDS 7000 7600 . + 1 ID "cds00003"; Parent "mRNA00003";
ctg123 . CDS 3391 3902 . + 0 ID "cds00004"; Parent "mRNA00003";
ctg123 . CDS 5000 5500 . + 1 ID "cds00004"; Parent "mRNA00003";
ctg123 . CDS 7000 7600 . + 1 ID "cds00004"; Parent "mRNA00003";

Next Steps

• Continue with the Tutorial.
2.1.23 Modifiers

Learning Objective  In this tutorial you will learn how to modify the elements of a container without copying them using SeqAn modifiers. You will learn about the different specializations and how to work with them.

Difficulty  Basic

Duration  20 min

Prerequisites  A First Example, Sequences

Overview

Modifiers give a different view to other classes. They can be used to change the elements of a container without touching them. For example, someone gave you an algorithm that works on two arbitrary String objects, but you want to use it for the special pair of a string and its reverse (left-to-right mirror). The classical approach would be to make a copy of the one string, where all elements are mirrored from left to right and call the algorithm with both strings. With modifiers, e.g. a ModifiedString, you can create the reverse in $O(1)$ extra memory without copying the original string. This can be handy if the original sequence is large.

Modifiers implement a certain concept (e.g. ContainerConcept, Iterator, ...) or class interface (String, ...) and thus can be used as such. The mirror modifier is already part of SeqAn and implements the class interface of String and can be used in every algorithm that works on strings.

The Modified String

The ModifiedString is a modifier that implements the String interface and thus can be used like a String. It has two template parameters. The first one specifies a sequence type (e.g. String, Segment, ...) and the second one specifies the modifiers behavior. That can be ModReverseString for mirroring a string left to right or ModViewModifiedString for applying a function to every single character (like ‘C’->’G’, ‘A’->’T’, ...).

ModReverse

We begin with the specialization ModReverseString from the example above. Now we have a given string:
#include <iostream>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace seqan;

int main ()
{
    String<char> myString = "A man, a plan, a canal-Panama";

    and want to get the reverse. So we need a ModifiedString specialized with String<char> and ModReverseString.
    We create the modifier and link it with myString:
    
    ModifiedString< String<char>, ModReverse > myModifier(myString);

    The result is:
    
    std::cout << myString << std::endl;
    std::cout << myModifier << std::endl;

    A man, a plan, a canal-Panama
    amanaP-lanac a ,nalp a ,nam A

    To verify that we didn’t copy myString, we replace an infix of the original string and see that, as a side effect, the
    modified string has also changed:

    replace(myString, 9, 9, "master ");
    std::cout << myString << std::endl;
    std::cout << myModifier << std::endl;
    return 0;
}

A man, a master plan, a canal-Panama
amanaP-lanac a ,nalp retsam a ,nam A

ModView

Another specialization of the ModifiedString is the ModViewModifiedString modifier. Assume we need all characters
of myString to be in upper case without copying myString. In SeqAn you first create a functor (a STL unary
function) which converts a character to its upper-case character.

struct MyFunctor : public std::unary_function<char,char>
{
    inline char operator()(char x) const
    {
        if (('a' <= x) && (x <= 'z')) return (x + ('A' - 'a'));
        return x;
    }
};

and then create a ModifiedString specialized with ModView<MyFunctor>:

ModifiedString< String<char>, ModView<MyFunctor> > myModifier(myString);

The result is:
A man, a plan, a canal—Panama
A MAN, A PLAN, A CANAL—PANAMA

The upper-case functor and some other predefined functors are part of SeqAn (in `seqan/modifier/modifier_functors.h`) already. The following functors can be used as an argument of `ModViewModifiedString`:

- **FunctorUpcase<TValue>** Converts each character of type `TValue` to its upper-case character
- **FunctorLowcase<TValue>** Converts each character to type `TValue` to its lower-case character
- **FunctorComplement<Dna>** Converts each nucleotide to its complementary nucleotide
- **FunctorComplement<Dna5>** The same for the `Dna5` alphabet
- **FunctorConvert<TInValue,TOutValue>** Converts the type of each character from `TInValue` to `TOutValue`

So instead of defining your own functor we could have used a predefined one:

```cpp
ModifiedString< String<char>, ModView<FunctorUpcase<char> > > myModifier(myString);
```

### Assignment 1

**Type Review**

**Objective**

In this assignment you will create a modifier using your own functor. Assume you have given two Dna sequences as strings as given in the code example below. Let’s assume you know that in one of your Dna sequences a few ‘C’ nucleotides are converted into ‘T’ nucleotides, but you still want to compare the sequence. Extend the code example as follows:

1. Write a functor which converts all ‘C’ nucleotides to ‘T’ nucleotides.
2. Define a `ModifiedString` with the specialization `ModViewModifiedString` using this functor.
3. Now you can modify both sequences to compare them, treating all ‘Cs’ as ‘Ts’. Print the results.

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace std;
using namespace seqan;

int main()
{
    typedef String<Dna> TSequence;

    TSequence seq1 = "CCCGGCATCATCC";
    TSequence seq2 = "CTTGGCATTATTC";

    std::cout << seq1 << std::endl;
    std::cout << seq2 << std::endl;
    std::cout << std::endl;
    ```
Solution

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace std;
using namespace seqan;

struct ConvertCT : public ::std::unary_function<Dna, Dna>
{
    inline Dna operator()(Dna x) const
    {
        if (x == 'C') return 'T';
        return x;
    }
};

int main()
{
    typedef String<Dna> TSequence;
    TSequence seq1 = "CCCGGCATCATCC";
    TSequence seq2 = "CTTGGCATTATTC";
    std::cout << seq1 << std::endl;
    std::cout << seq2 << std::endl;

    typedef ModifiedString<TSequence, ModView<ConvertCT>> TModCT;
    TModCT modCT1(seq1);
    TModCT modCT2(seq2);
    std::cout << modCT1 << std::endl;
    std::cout << modCT2 << std::endl;

    return 0;
}
```

```cpp
CCCGGCATCATCC
CTTGGCATTATTC
TTTGGTATTATTT
TTTGGTATTATTT
```

For some commonly used modifiers you can use the following shortcuts:
The Modified Iterator

We have seen how a ModifiedString can be used to modify strings without touching or copying original data. The same can be done with iterators. The ModifiedIterator implements the Iterator concept and thus can be used in every algorithm or data structure that expects an iterator. In fact, we have already used the ModifiedIterator unknowingly in the examples above, as in our cases the ModifiedString returns a corresponding ModifiedIterator via the Iterator meta-function. The main work is done in the ModifiedIterator, whereas the ModifiedString only overloads the begin and end. Normally, you are going to use the ModifiedString and maybe the result of its Iterator meta-function instead of a ModifiedIterator directly.

Nested Modifiers

As modifiers implement a certain concept and depend on classes of this concept, two modifiers can be chained to create a new modifier. We have seen how the ModifiedString specialized with ModReverseString and ModViewModifiedString can be used. Now we want to combine them to create a modifier for the reverse complement of a DnaString. We begin with the original string:

String<Dna> myString = "attacgg";

Then we define the modifier that complements a DnaString:

typedef ModifiedString<String<Dna>, ModComplementDna> TMyComplement;

This modifier now should be reversed from left to right:

typedef ModifiedString<TMyComplement, ModReverse> TMyReverseComplement;

The original string can be given to the constructor.

TMyReverseComplement myReverseComplement (myString);

The result is:

std::cout << myString << 'n';
std::cout << myReverseComplement << 'n';

infix(myString, 1, 1) = "cgt";

std::cout << myString << 'n';
std::cout << myReverseComplement << 'n';

ATTACGG
CCGTAAT
Using a predefined shortcut, the whole example could be reduced to:

```cpp
String<Dna> myString = "attacgg";
std::cout << myString << std::endl;
std::cout << DnaStringReverseComplement(myString) << std::endl;
```

### 2.1.24 Randomness

**Learning Objective** In this tutorial, you will get a short walk-through of SeqAn’s module random. The module contains code for generating random numbers in various distributions.

**Difficulty** Basic

**Duration** 10 min

**Prerequisites** A First Example, Sequences

This tutorial explains how to use the module random. random primarily provides the two classes Rng (Random Number Generator) and Pdf (Probability Density Functions).

**Random Number Generation**

Instances of the class Rng are responsible for random number generation. Currently, SeqAn provides only one specialization, the MersenneTwisterRng. This class is used for generating random 32-bit numbers. These 32-bit numbers are then used by the Pdf specializations to generate random numbers in certain distributions as explained further below.

The following program gives an example. First, we include the header <seqan/random.h> to get access to the module’s functionality.

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h>
#include <seqan/random.h>

using namespace seqan;

const int SEED = 42;

int main()
{
```
During the initialization of the `Rng` object you have to pass a seed used as a start point for the randomization. Finally, function `pickRandomNumber` picks a random number from a `Rng`.

```cpp
Rng<MersenneTwister> rng(SEED);
std::cout << "pickRandomNumber(rng) == " << pickRandomNumber(rng) << std::endl;
```

The output of this fragment is:
```
pickRandomNumber(rng) == 1608637542
```

**Mersenne Twister Rng** generates 32-bit unsigned numbers. However, you should not rely on any specific type and use `Value` metafunction instead.

```cpp
typedef Value<Rng<MersenneTwister> >::Type TMTValue;
TMTValue value = pickRandomNumber(rng);
```

If you prefer a special distribution of the randomly generated numbers you can use the above mentioned specializations of `Pdf`. SeqAn currently provides normal, log-normal and uniform probability density functions. Note, for uniform distributions the range of values is given as a closed interval, i.e. the last value is enclosed in the range.

```cpp
Pdf<Uniform<double> > uniformDouble(0, 1);
std::cout << "pickRandomNumber(rng, uniformDouble) == " << pickRandomNumber(rng, uniformDouble) << std::endl;
Pdf<Uniform<int> > uniformInt(0, 42);
std::cout << "pickRandomNumber(rng, uniformInt) == " << pickRandomNumber(rng, uniformInt) << std::endl;
Pdf<Normal> normal(0, 1);
std::cout << "pickRandomNumber(rng, normal) == " << pickRandomNumber(rng, normal) << std::endl;
Pdf<LogNormal> logNormal(0, 1);
std::cout << "pickRandomNumber(rng, logNormal) == " << pickRandomNumber(rng, logNormal) << std::endl;
```

The output of this fragment is:
```
pickRandomNumber(rng, uniformDouble) == 0.950714
pickRandomNumber(rng, uniformInt) == 27
pickRandomNumber(rng, normal) == 0.419823
pickRandomNumber(rng, logNormal) == 1.22431
```

Also note that you can initialize the **Log-Normal Pdf** either with mean and standard deviation of the log-normal distribution or the underlying normal distribution. By default, you initialize it with the mean and standard deviation (mu and sigma) of the underlying normal distribution. Use the tags `MuSigma` and `MeanStdDev` in the constructor to select a mode.

```cpp
Pdf<LogNormal> logNormal2(0, 1, MuSigma());
std::cout << "pickRandomNumber(rng, logNormal2) == " << pickRandomNumber(rng, logNormal2) << std::endl;
Pdf<LogNormal> logNormal3(0.1, 1, MeanStdDev());
std::cout << "pickRandomNumber(rng, logNormal3) == " << pickRandomNumber(rng, logNormal3) << std::endl;
```

```
pickRandomNumber(rng, logNormal) == 1.22431
pickRandomNumber(rng, logNormal2) == 2.78004
pickRandomNumber(rng, logNormal3) == 0.00155248
```

**Shuffling**

The function `shuffle` allows to shuffle a container, given a random number generator:

```cpp
CharString container = "Hello World!";
shuffle(container, rng);
std::cout << "shuffle("Hello World!") == " << container << std::endl;
```

return 0;
```
The output of this fragment is:

`shuffle("Hello World!") == oreWlloHld`

### 2.1.25 Seed-and-Extend

**Learning Objective** In this tutorial, you will learn about the seeds-related SeqAn functionality. You will learn how to do seed-and-extend with SeqAn, how to do local and global chaining of seeds. Finally, we will show how to create a banded alignment around a seed chain.

**Difficulty** Average

**Duration** 2 h

**Prerequisites** *Sequences*

Many efficient heuristics to find high scoring, but inexact, local alignments between two sequences start with small exact (or at least highly similar) segments, so called seeds, and extend or combine them to get larger highly similar regions. Probably the most prominent tool of this kind is BLAST [AGM+90], but there are many other examples like FASTA [Pee90] or LAGAN [BDC+03].

SeqAn’s header file for all data structures and functions related to two-dimensional seeds is `<seqan/seeds.h>`. 
The Seed Class

The Seed class allows to store seeds. Seeds have a begin and end position in each sequence. Often, two or more close seeds are combined into a larger seed, possibly causing a shift in horizontal or vertical direction between the begin position of the upper left seed and the end position of the lower right seed. For this reason, the Seed class also stores an upper and a lower diagonal to reflect the expansion between those shifted seeds.

The image to the right shows an example where three smaller seeds (black diagonals) were combined (or “chained locally”) into one larger seed (green nine-sided area). The first seed lies on the begin diagonal, the lowermost seed on the lower diagonal and the uppermost seed on the upper diagonal.

The Simple Seed specialization only stores the begin and end positions of the seed (left-uppermost and right-lowermost corners of green surface) in both sequences and the upper and lower diagonal. The initial diagonals are not stored. The ChainedSeed specialization additionally stores these information. In most cases, the Simple Seed class is sufficient since the best alignment around the seeds has to be determined using a banded alignment algorithm of the seed infixes anyway.

You can get/set the begin and end position in the horizontal/vertical sequences using the functions beginPositionH, beginPositionV, setBeginPositionH, and setBeginPositionV. The band information can be queried and updated using lowerDiagonal, upperDiagonal, setLowerDiagonal, and setUpperDiagonal. Note, we use the capital letters 'H' and 'V' to indicate horizontal direction and vertical direction, respectively, while the database is always considered as the horizontal sequence and the query as the vertical sequence in the context of sequence alignments.

The following program gives an example of creating seeds as well as setting and reading properties.

```cpp
// Default-construct seed.
seqan::Seed<seqan::Simple> seed1;
std::cout << "seed1\n"
<< "beginPositionH == " << beginPositionH(seed1) << "\n"
<< "endPositionH == " << endPositionH(seed1) << "\n"
<< "beginPositionV == " << beginPositionV(seed1) << "\n"
<< "endPositionV == " << endPositionV(seed1) << "\n"
<< "lowerDiagonal == " << lowerDiagonal(seed1) << "\n"
<< "upperDiagonal == " << upperDiagonal(seed1) << "\n"
;

// Construct seed with begin and end position in both sequences.
seqan::Seed<seqan::Simple> seed2(3, 10, 7, 14);
setUpperDiagonal(seed2, -7);
setLowerDiagonal(seed2, -9);
std::cout << "seed2\n"
<< "beginPositionH == " << beginPositionH(seed2) << "\n"
```
The output to the console is as follows.

```cpp
seed1
beginPositionH == 0
endPositionH == 0
beginPositionV == 0
endPositionV == 0
lowerDiagonal == 0
upperDiagonal == 0

seed2
beginPositionH == 3
endPositionH == 7
beginPositionV == 10
endPositionV == 14
lowerDiagonal == -9
upperDiagonal == -7
```

**Assignment 1**

**Type** Review

**Objective** Extend the program above such that `seed1` is updated from `seed2` and all members (begin positions, end positions, diagonals) are equal to the corresponding member of `seed` times two. For example, the lower diagonal of `seed2` should be $2 \times \text{lowerDiagonal(seed1)}$.

**Solution**

**Seed Extension**

Seeds are often created quickly using a $k$-mer index: When a $k$-mer of a given length is found in both sequences, we can use it as a seed. However, the match can be longer than just $k$ characters. To get longer matches, we use seed extension.

In the most simple case we simply look for matching characters in both sequences to the left and right end of the seed. This is called match extension and available through the `extendSeed` function using the `MatchExtend` tag.
```cpp
extendSeed(seed, seqH, seqV, seqan::EXTEND_BOTH, scoringScheme, 3,
    seqan::UnGappedXDrop());
// Print the resulting seed.
std::cout << "result:\n"
        << "seedH: " << infix(seqH, beginPositionH(seed),
                         endPositionH(seed)) << "\n"
        << "seedV: " << infix(seqV, beginPositionV(seed),
                         endPositionV(seed)) << "\n";

original
seedH: ROW
seedV: ROW
result
seedH: ick BROW
seedV: ick BROW

Assignment 2

Type Review
Objective Change the example from above to extend the seed to both sides.
Solution

```cpp
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/seeds.h>

int main()
{
    // The horizontal and vertical sequence (database and query).
    seqan::CharString seqH = "The quick BROWN fox jumped again!";
    seqan::CharString seqV = "thick BROWNIES for me!";
    // ^^^
    // Create seed and print the seed sequence.
    seqan::Seed<seqan::Simple> seed(11, 7, 14, 10);
    std::cout << "original:\n"
        << "seedH: " << infix(seqH, beginPositionH(seed),
                         endPositionH(seed)) << "\n"
        << "seedV: " << infix(seqV, beginPositionV(seed),
                         endPositionV(seed)) << "\n";
    // Perform match extension.
    extendSeed(seed, seqH, seqV, seqan::EXTEND_BOTH, seqan::MatchExtend());
    // Print the resulting seed.
    std::cout << "result:\n"
        << "seedH: " << infix(seqH, beginPositionH(seed),
                         endPositionH(seed)) << "\n"
        << "seedV: " << infix(seqV, beginPositionV(seed),
                         endPositionV(seed)) << "\n";

    return 0;
}
```

A more complex case is the standard bioinformatics approach of x-drop extension:

In the ungapped case, we extend the seed by comparing the i-th character to the left/right of the seed of the horizontal sequence with the j-th character to the left/right of the seed in the vertical sequence. Matches and mismatches are
assigned with scores (usually matches are assigned with positive scores and mismatches are assigned with negative scores). The scores are summed up. When one or more mismatches occur, the running total will drop. When the sum drops more strongly than a value :math:`x`, the extension is stopped.

This approach is also available in the gapped case in the SeqAn library. Here, creating gaps is also possible but also assigned negative scores.

```cpp
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/score.h>
#include <seqan/seeds.h>

int main()
{
    // The horizontal and vertical sequence (database and query).
    seqan::CharString seqH = "The quick BROWN fox jumped again!";
    seqan::CharString seqV = "thick BROWNIES for me!";
    // ^^^
    // Create seed and print the seed sequence.
    seqan::Seed<seqan::Simple> seed(11, 7, 14, 10);
    std::cout << "original

seedH: " << infix(seqH, beginPositionH(seed),
    endPositionH(seed)) << "

seedV: " << infix(seqV, beginPositionV(seed),
    endPositionV(seed)) << "

// Perform match extension.
extendSeed(seed, seqH, seqV, seqan::EXTEND_LEFT, seqan::MatchExtend());
    // Print the resulting seed.
    std::cout << "result

seedH: " << infix(seqH, beginPositionH(seed),
    endPositionH(seed)) << "

seedV: " << infix(seqV, beginPositionV(seed),
    endPositionV(seed)) << "

original
seedH: ROW
seedV: ROW
result
seedH: ick BROWN fox
seedV: ick BROWN box
```

Assignment 3

Type Review

Objective Change the example from above to use gapped instead of ungapped x-drop extension.

Solution

```cpp
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/score.h>
#include <seqan/seeds.h>

int main()
{
    // The horizontal and vertical sequence (database and query).
    seqan::CharString seqH = "The quick BROWN fox jumped again!";
    seqan::CharString seqV = "thick BROWN boxes of brownies!";
    // ^^^
    // Create seed and print the seed sequence.
    seqan::Seed<seqan::Simple> seed(11, 7, 14, 10);
    std::cout << "original

" << "seedH: " << infix(seqH, beginPositionH(seed),
    endPositionH(seed)) << "

" << "seedV: " << infix(seqV, beginPositionV(seed),
    endPositionV(seed)) << "

    // Perform match extension.
    extendSeed(seed, seqH, seqV, seqan::EXTEND_LEFT, seqan::MatchExtend());
    // Print the resulting seed.
    std::cout << "result

" << "seedH: " << infix(seqH, beginPositionH(seed),
    endPositionH(seed)) << "

" << "seedV: " << infix(seqV, beginPositionV(seed),
    endPositionV(seed)) << "

original
seedH: ROW
seedV: ROW
result
seedH: ick BROWN fox
seedV: ick BROWN box
```
endPositionH(seed)) << "\n"
  "seedV: " << infix(seqV, beginPositionV(seed), endPositionV(seed)) << "\n";

// Perform match extension.
  seqan::Score<int, seqan::Simple> scoringScheme(1, -1, -1);
  extendSeed(seed, seqH, seqV, seqan::EXTEND_BOTH, scoringScheme, 3,
             seqan::GappedXDrop());
// Print the resulting seed.
  std::cout << "result\n"
           << "seedH: " << infix(seqH, beginPositionH(seed),
                          endPositionH(seed)) << "\n"
           << "seedV: " << infix(seqV, beginPositionV(seed),
                          endPositionV(seed)) << "\n";

  return 0;
}

After extending a seed, we might wish to actually get the resulting alignment. When using gapped x-drop extension, we need to perform a banded global alignment of the two sequence infixes that correspond to the seed. This is shown in the following example:

// The horizontal and vertical sequence (database and query).
  seqan::CharString seqH = "The quick BROWN fox jumped again!";
  seqan::CharString seqV = "thick BROWN boxes of brownies!";

// Create seed and print the seed sequence.
  seqan::Seed<seqan::Simple> seed(11, 7, 14, 10);

// Perform match extension.
  seqan::Score<int, seqan::Simple> scoringScheme(1, -1, -1);
  extendSeed(seed, seqH, seqV, seqan::EXTEND_BOTH, scoringScheme, 3,
             seqan::UnGappedXDrop());

// Perform a banded alignment.
  seqan::Align<seqan::CharString> align;
  resize(rows(align), 2);
  assignSource(row(align, 0), infix(seqH, beginPositionH(seed),
                         endPositionH(seed)));
  assignSource(row(align, 1), infix(seqV, beginPositionV(seed),
                         endPositionV(seed)));

// TODO(holtgrew): Use seed diagonals as bands.
  globalAlignment(align, scoringScheme);
  std::cerr << "Resulting alignment\n" << align << "\n";

Resulting alignment
  0 . :
  ick BROWN fox
     ! ! ! ! ! ! !
  ick BROWN box

Assignment 4

Type Review

Objective Change the example from above to a gap open score of −2 and a gap extension score of −2. Use this
scoring scheme for the global alignment as well and thus Gotoh’s algorithm.

**Solution**  Note that we do not have to explicitly call Gotoh’s algorithm in `globalAlignment()`. The fact that the gap extension score is different from the gap opening score is enough.

```c
#include <seqan/align.h>
#include <seqan/file.h>
#include <seqan/score.h>
#include <seqan/seeds.h>
#include <seqan/sequence.h>

int main()
{
    // The horizontal and vertical sequence (database and query).
    seqan::CharString seqH = "The quick BROWN fox jumped again!";
    seqan::CharString seqV = "thick BROWN boxes of brownies!";
    // ^^^
    // Create seed and print the seed sequence.
    seqan::Seed<seqan::Simple> seed(11, 7, 14, 10);

    // Perform match extension.
    seqan::Score<int, seqan::Simple> scoringScheme(1, -1, -1, -2);
    extendSeed(seed, seqH, seqV, seqan::EXTEND_BOTH, scoringScheme, 3,
               seqan::GappedXDrop());

    // Perform a banded alignment.
    seqan::Align<seqan::CharString> align;
    resize(rows(align), 2);
    assignSource(row(align, 0), infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)));
    assignSource(row(align, 1), infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)));

    // TODO(holtgrew): Use seed diagonals as bands.
    globalAlignment(align, scoringScheme);
    std::cerr << "Resulting alignment";
    // FRAGMENT(footer)
    return 0;
}
```

### Local Chaining using Seed Sets

Usually, we quickly determine a large number of seeds. When a seed is found, we want to find a “close” seed that we found previously and combine it to form a longer seed. This combination is called local chaining. This approach has been pioneered in the CHAOS and BLAT programs.

SeqAn provides the `SeedSet` class as a data structure to efficiently store seeds and combine new seeds with existing ones. The following example creates a `SeedSet` object `seeds`, adds four seeds to it and then prints its contents.

```c
typedef seqan::Seed<seqan::Simple> TSeed;
typedef seqan::SeedSet<seqan::Simple> TSeedSet;

TSeedSet seeds;
addSeed(seeds, TSeed(0, 0, 2), seqan::Single());
addSeed(seeds, TSeed(3, 5, 2), seqan::Single());
addSeed(seeds, TSeed(4, 2, 3), seqan::Single());
addSeed(seeds, TSeed(9, 9, 2), seqan::Single());
```
std::cout << "Resulting seeds.\n";

typedef seqan::Iterator<TSeedSet>::Type TIter;
for (TIter it = begin(seedSet, seqan::Standard()); it != end(seedSet, seqan::Standard()); ++it)
    std::cout << "(" << beginPositionH(*it) << ", " << endPositionH(*it) << ", " << beginPositionV(*it) << ", " << endPositionV(*it) << ", " << lowerDiagonal(*it) << ", " << upperDiagonal(*it) << ")\n";

The output of the program above can be seen below.

Resulting seeds.
(3, 5, 5, 7, -2, -2)
(0, 2, 0, 2, 0, 0)
(9, 11, 9, 11, 0, 0)
(4, 7, 2, 5, 2, 2)

Note that we have used the Single() tag for adding the seeds. This forces the seeds to be added independent of the current seed set contents.

By using different overloads of the addSeed, we can use various local chaining strategies when adding seed A.

**Merge** If there is a seed B that overlaps with A and the difference in diagonals is smaller than a given threshold then A can be merged with B.

**SimpleChain** If there is a seed B whose distance in both sequences is smaller than a given threshold then A can be chained to B.

**Chaos** Following the strategy of Chaos [BCGottgens+03], if A is within a certain distance to B in both sequences and the distance in diagonals is smaller than a given threshold then A can be chained to B.

The addSeed function returns a boolean value indicating success in finding a suitable partner for chaining. A call using the Single strategy always yields true.

The following example shows how to use the SimpleChain strategy.

```cpp
typedef seqan::Seed<seqan::Simple> TSeed;
typedef seqan::SeedSet<seqan::Simple> TSeedSet;

seqan::Dna5String seqH;
seqan::Dna5String seqV;
seqan::Score<int, seqan::Simple> scoringScheme(1, -1, -1);

seqan::String<TSeed> seeds;
appendValue(seeds, TSeed(0, 0, 2));
appendValue(seeds, TSeed(3, 5, 2));
appendValue(seeds, TSeed(4, 2, 3));
appendValue(seeds, TSeed(9, 9, 2));

TSeedSet seedSet;
for (unsigned i = 0; i < length(seeds); ++i)
{
    if (!addSeed(seedSet, seeds[i], 2, 2, scoringScheme, seqH, seqV, seqan::SimpleChain()))
        addSeed(seedSet, seeds[i], seqan::Single());
}

std::cout << "Resulting seeds.\n";
```
As we can see, the seed $\text{TSeed}(4, 2, 3)$ has been chained to $\text{TSeed}(0, 0, 2)$.

Resulting seeds.
- $(3, 5, 5, 7, -2, -2)$
- $(0, 7, 0, 5, 0, 2)$
- $(9, 11, 9, 11, 0, 0)$

Assignment 5

Type Review

Objective Change the example above to use the Chaos strategy instead of the SimpleChain.

Solution

```cpp
#include <seqan/file.h>
#include <seqan/score.h>
#include <seqan/seeds.h>
#include <seqan/sequence.h>

int main()
{
    typedef seqan::Seed<seqan::Simple> TSeed;
    typedef seqan::SeedSet<seqan::Simple> TSeedSet;

    seqan::Dna5String seqH;
    seqan::Dna5String seqV;
    seqan::Score<int, seqan::Simple> scoringScheme(1, -1, -1);

    seqan::String<TSeed> seeds;
    appendValue(seeds, TSeed(0, 0, 2));
    appendValue(seeds, TSeed(3, 5, 2));
    appendValue(seeds, TSeed(4, 2, 3));
    appendValue(seeds, TSeed(9, 9, 2));

    TSeedSet seedSet;
    for (unsigned i = 0; i < length(seeds); ++i)
    {
        if (!addSeed(seedSet, seeds[i], 2, 2, scoringScheme, seqH, seqV, seqan::Chaos()))
            addSeed(seedSet, seeds[i], seqan::Single());
    }

    std::cout << "Resulting seeds.\n";
    typedef seqan::Iterator<TSeedSet>::Type TIter;
    for (TIter it = begin(seedSet, seqan::Standard());
        it != end(seedSet, seqan::Standard()); ++it)
    {
        std::cout << "(" << beginPositionH(*it) << ", " << endPositionH(*it)
                  << ", " << beginPositionV(*it) << ", " << endPositionV(*it)
                  << ", " << lowerDiagonal(*it) << ", " << upperDiagonal(*it)
                  << ")\n";
    }
}
```

2.1. Tutorial
Global Chaining

After one has determined a set of candidate seeds, a lot of these seeds will conflict. The image to the right shows an example. Some conflicting seeds might be spurious matches or come from duplication events.

Often, we need to find a linear ordering of the seeds such that each seed starts after all of its predecessor end in both sequences. This can be done efficiently using dynamic sparse programming (in time $O(n \log n)$ where $n$ is the number of seeds) as described in [Gus97]. The red seeds in the image to the right show such a valid chain.

This functionality is available in SeqAn using the `chainSeedsGlobally` function. The function gets a sequence container of `Seed` objects for the result as its first parameter and a `SeedSet` as its second parameter. A subset of the seeds from the `SeedSet` are then selected and stored in the result sequence.

The following shows a simple example.

```cpp
typedef seqan::Seed<seqan::Simple> TSeed;
typedef seqan::SeedSet<seqan::Simple> TSeedSet;
TSeedSet seedSet;
addSeed(seedSet, TSeed(0, 0, 2), seqan::Single());
addSeed(seedSet, TSeed(3, 5, 2), seqan::Single());
addSeed(seedSet, TSeed(4, 2, 3), seqan::Single());
addSeed(seedSet, TSeed(9, 9, 2), seqan::Single());

seqan::String<TSeed> result;
chainSeedsGlobally(result, seedSet, seqan::SparseChaining());
```

Assignment 6

Type Review

Objective Change the example from above to use a different chain of seeds. The seeds should be TSeed(1, 1, 3), TSeed(6, 9, 2), TSeed(10, 13, 3), and TSeed(20, 22, 5).

Solution
Banded Chain Alignment

After obtaining such a valid seed chain, we would like to obtain an alignment along the chain. For this, we can use the so-called banded chain alignment algorithm (introduced by Brudno's LAGAN). Around seeds, we can use banded DP alignment and the spaces between seeds can be aligned using standard DP programming alignment.

In SeqAn you can compute the banded chain alignment by calling the function `bandedChainAlignment`. This function gets the structure in which the alignment should be stored as the first parameter. This corresponds to the interface of the `globalAlignment` and allows the same input types. Additionally, this function requires a non-empty, non-decreasing monotonic chain of seeds which is used as the rough global map for computing the global alignment. The third required parameter is the Score.

Note, that there are a number of optional parameters that can be specified. These include a second Score which, if specified, is used to evaluate the regions between two consecutive seeds differently than the regions around the seeds itself (for which then the first specified score is taken.). As for the global alignment you can use the `AlignConfig` to specify the behavior for initial and end gaps. The last optional parameter is the band extension. This parameter specifies to which size the bands around the anchors should be extended. The default value is 15 and conforms the default value in the LAGAN-algorithm [BDC+03].

Important: At the moment the specified value for the band extension must be at least one.

defined $TSeed$;

Dna5String sequenceH = "CGAACACTCCACACA";
Dna5String sequenceV = "GGCGATNNACATGCACA";

String<TSeed> seedChain;
appendValue(seedChain, TSeed(0, 2, 5, 6));
appendValue(seedChain, TSeed(6, 9, 9, 12));
appendValue(seedChain, TSeed(11, 14, 17, 16));

Align<Dna5String, ArrayGaps> alignment;
resize(rows(alignment), 2);
assignSource(row(alignment, 0), sequenceH);
assignSource(row(alignment, 1), sequenceV);
SeqAn Manual, Release 1.4.2

Score<int, Simple> scoringScheme(2, -1, -2);

int result = bandedChainAlignment(alignment, seedChain, scoringScheme, 2);

std::cout << "Score: " << result << std::endl;
std::cout << alignment << std::endl;

The output of the example above.

Score: 5
  0 . : . :
  --CGAAT--CCATCCCACACA
  || || |||| | |||
  GGCG-ATNNNCATGG--CACA

Assignment 7

Type Review

Objective Change the example form above to use two different scoring schemes. The scoring scheme for the seeds should use the Levenshtein distance and the score for the gap regions should be an affine score with the following values: match = 2, mismatch = -1, gap open = -2, gap extend = -1.

Furthermore, we are looking for a semi-global alignment here the initial and end gaps in the query sequence are free.

Solution

#include <seqan/sequence.h>
#include <seqan/align.h>
#include <seqan/score.h>
#include <seqan/seeds.h>

int main()
{

typedef seqan::Seed<seqan::Simple> TSeed;

seqan::Dna5String sequenceH = "CGAATCCATCCCACACA";
seqan::Dna5String sequenceV = "GGCGATNNNCATGGCACA";
seqan::Score<int, seqan::Simple> scoringSchemeAnchor(0, -1, -1);
seqan::Score<int, seqan::Simple> scoringSchemeGap(2, -1, -1, -2);

seqan::String<TSeed> seedChain;
seqan::appendValue(seedChain, TSeed(0, 2, 5, 6));
seqan::appendValue(seedChain, TSeed(6, 9, 9, 12));
seqan::appendValue(seedChain, TSeed(11, 14, 17, 16));

seqan::Align<seqan::Dna5String, seqan::ArrayGaps> alignment;
seqan::resize(seqan::rows(alignment), 2);
seqan::assignSource(seqan::row(alignment, 0), sequenceH);
seqan::assignSource(seqan::row(alignment, 1), sequenceV);
seqan::AlignConfig<true, false, false, true> alignConfig;

int result = seqan::bandedChainAlignment(alignment, seedChain, scoringSchemeAnchor, scoringSchemeGap, alignConfig, 2);

std::cout << "Score: " << result << std::endl;
std::cout << alignment << std::endl;
2.1.26 Parsing Command Line Arguments

**Learning Objective** You will learn how to use the `ArgumentParser` class to parse command line arguments. This tutorial is a walk-through with links into the API documentation and also meant as a source for copy-and-paste code.

**Difficulty** Easy

**Duration** 30-60 min

**Prerequisites** *A First Example, Sequences*, familiarity with building SeqAn apps

The simplest possible and also most flexible interface to a program is through the command line. This tutorial explains how to parse the command line using the SeqAn library’s `ArgumentParser` class.

Using this class will give you the following functionality:

- Robust parsing of command line arguments.
- Simple verification of arguments (e.g. within a range, one of a list of allowed values).
- Automatically generated and nicely formatted help screens when called with `--help`. You can also export this help to HTML and man pages.
- In the future, you will be able to automatically generate nodes for workflow engines such as KNIME or Galaxy.
As a continuous example, we will write a little program that is given strings on the command line and applies an operation to every i-th character:

```bash
# modify_string --uppercase -i 2 "This is some text!"
This iS SoMe TeXt!
# modify_string "This is some text!" --lowercase -i 1
this is some text!
```

The program has three types of command line options/arguments:

- **Two flag options** --uppercase and --lowercase that select the operation.
- **One (value) option** -i that selects the period of the characters that the operation is to be applied to and is given a value (2 in the first call, 1 in the second).
- **One (positional) argument** with the text to modify (above "This is some text!"). In contrast to options, arguments are not identified by their names but by their position.

Command line options can have a **long name** (e.g. --lowercase) and/or a **short name** (e.g. -i).

### A First Working Example

The following small program will (1) setup an ArgumentParser object named parser,(2) parse the command line, (3) exit the program if there were errors or the user requested a functionality that is already built into the command line parser, and (4) printing the settings given from the command line. Such functionality is printing the help, for example.

```c++
#include <iostream>
#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    addArgument(parser, seqan::ArgParseArgument(seqan::ArgParseArgument::STRING, "TEXT"));
    addOption(parser, seqan::ArgParseOption("i", "period", "Period to use for the index.",
                                            seqan::ArgParseArgument::INTEGER, "INT"));
    addOption(parser, seqan::ArgParseOption("U", "uppercase", "Select to-uppercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    unsigned period = 0;
    getOptionValue(period, parser, "period");
    bool toUppercase = isSet(parser, "uppercase");
    seqan::CharString text;
    getArgumentValue(text, parser, 0);
}
```
Let us first play a bit around with the program before looking at it in detail.

For example, we can already let the program generate an online help:

```bash
# modify_string -h
modify_string
============
SYNOPSIS
DESCRIPTION
- h, --help
  Displays this help message.
- i, --period INT
  Period to use for the index.
- U, --uppercase
  Select to-uppercase as operation.

VERSION
modify_string version:
Last update
```

While already informative, the help screen looks like there is something missing. For example, there is no synopsis, no version and no date of the last update given. We will fill this in later.

When we pass some parameters, the settings are printed:

```bash
# modify_string "This is a test." -i 1 -U
period 1
uppercase 1
text This is a test.
```

When we try to use the `--lowercase/-L` option, we get an error. This is not surprising since we did not tell the argument parser about this option yet.

```bash
# modify_string "This is a test." -i 1 -L
modify_string: illegal option -- L
```

### A Detailed Look

Let us look at this program in detail now. The required SeqAn module is `seqan/arg_parse.h`. After inclusion, we can create an `ArgumentParser` object:

```cpp
seqan::ArgumentParser parser("modify_string");
```

Then, we define a positional argument using the function `addArgument`. The function accepts the parser and an `ArgParseArgument` object. We call the `ArgParseArgument` constructor with three parameters: the type of the argument (a string), and a label for the documentation.

```cpp
addArgument(parser, seqan::ArgParseArgument::STRING, "TEXT");
```
Then, we add options to the parser using `addOption`. We pass the parser and an ArgParseOption object.

```cpp
addOption(parser, seqan::ArgParseOption(
    "i", "period", "Period to use for the index.",
    seqan::ArgParseArgument::INTEGER, "INT"));
addOption(parser, seqan::ArgParseOption(
    "U", "uppercase", "Select to-uppercase as operation.");
```

The ArgParseOption constructor is called in two different variants. Within the first addOption call, we construct an integer option with a short and long name, a documentation string, and give it the label “INT”. The second option is a flag (indicated by not giving a type) with a short and a long name and a description.

**Tip:** List Arguments and Options.

You have to mark an option to be a list if you want to be able to collect multiple values for it from the command line. Consider the following program call:

```
# program -a 1 -a 2 -a 3
```

If the option `a` is not a list then the occurrence `-a 3` overwrites all previous settings.

However, if `a` is marked to be a list, then all values (1, 2, and 3) are stored as its values. We can get the number of elements using the function `getOptionValueCount` and then access the individual arguments using the function `getOptionValue`. You can mark an option and arguments to be lists by using the `isList` parameter to the ArgParseArgument and ArgParseOption constructors.

For arguments, only the first or the last argument or none can be a list but not both. Consider this program call:

```
# program arg0 arg1 arg2 arg3
```

For example, if the program has three arguments and the first one is a list then `arg0` and `arg1` would be the content of the first argument. If it has two arguments and the last one is a list then `arg1`, `arg2`, and `arg3` would be the content of the last argument.

Next, we parse the command line using `parse`.

```cpp
seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);
```

We then check the result of the parsing operation. The result is `seqan::ArgumentParser::PARSE_ERROR` if there was a problem with the parsing. Otherwise, it is `seqan::ArgumentParser::PARSE_OK` if there was no problem and no special functionality of the argument parser was triggered. The command line parser automatically adds some arguments, such as `--help`. If such built-in functionality is triggered, it will return a value that is neither `PARSE_ERROR` nor `PARSE_OK`.

The following two lines have the following behaviour. If the parsing went through and no special functionality was triggered then the branch is not taken. Otherwise, the method `main()` is left with 1 in case of errors and with 0 in case special behaviour was triggered (e.g. the help was printed).

```cpp
if (res != seqan::ArgumentParser::PARSE_OK)
    return res == seqan::ArgumentParser::PARSE_ERROR;
```

Finally, we access the values from the command line using the ArgumentParser. The function `getOptionValue` allows us to access the values from the command line after casting into C++ types. The function `isSet` allows us to query whether a given argument was set on the command line.

```cpp
unsigned period = 0;
getOptionValue(period, parser, "period");
bool toUppercase = isSet(parser, "uppercase");
seqan::CharString text;
getOptionValue(text, parser, 0);
```
Assignment 1

Type  Reproduction

Objective  Copy the source code of the full First Working Example above into a demo or an app in your sandbox. Compile it and test printing the help screen and calling it with the two command lines above.

Solution  You can do it!

Assignment 2

Type  Reproduction

Objective  Adjust the program from above to also accept an option to convert characters to lower case, just as it accepts options to convert characters to upper case. The long name should be `--lowercase`, the short name should be `-L`. As for the `--uppercase` option, the program should print whether the flag was set or not.

Hint  Copy the two lines for defining the `<tt>--uppercase</tt>` option and replace the strings appropriately.

Solution

```cpp
#include <iostream>
#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation.");
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation.");

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Hit if the flag was set.
    unsigned period = 0;
    getOptionValue(period, parser, "period");

    std::cout << "period	" << period << '
' << "uppercase	" << toUppercase << '
' << "text	" << text << '
';
```

2.1. Tutorial
Using Default Values

Would it not be nice if we could specify a default value for --period, so it is 1 if not specified and simply each character is modified? We can do this by using the function setDefaultValue:

```cpp
defaultValue(parser, "period", "1");
```

Note that we are giving the default value as a string. The ArgumentParser object will simply interpret it as if it was given on the command line. There, of course, each argument is a string.

Assignment 3

Setting a default value

Type Reproduction

Objective Adjust the previous program to accept default values by adding the setDefaultValue() line from above into your program.

Solution

```cpp
#include <iostream>

#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    addArgument(parser, seqan::ArgParseArgument(
            seqan::ArgParseArgument::STRING, "TEXT");
    addOption(parser, seqan::ArgParseOption(
            "i", "period", "Period to use for the index.",
            seqan::ArgParseArgument::INTEGER, "INT");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
            "U", "uppercase", "Select to-uppercase as operation.");
    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
```
// Otherwise, exit with code 0 (e.g. help was printed).
if (res != seqan::ArgumentParser::PARSE_OK)
    return res == seqan::ArgumentParser::PARSE_ERROR;

// Extract option values and print them.
unsigned period = 0;
getOptionValue(period, parser, "period");
bool toUppercase = isSet(parser, "uppercase");
bool toLowercase = isSet(parser, "lowercase");
seqan::CharString text;
getOptionValue(text, parser, 0);

std::cout << "period \t" << period << '\n'
    << "uppercase\t" << toUppercase << '\n'
    << "lowercase\t" << toLowercase << '\n'
    << "text \t" << text << '\n';

return 0;

Best Practice: Using Option Structs

Instead of just printing the options back to the user, we should actually store them. To follow best practice, we should
not use global variables for this but instead pass them as parameters.

We will thus create a ModifyStringOptions struct that encapsulates the settings the user can give to the
modify_string program. Note that we initialize the variables of the struct with initializer lists, as it is best practice
in modern C++.

The <tt>ModifyStringOptions</tt> struct’s definition is shown below. Click “more...” to see the whole updated
program.

```cpp
struct ModifyStringOptions {
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), toUppercase(false), toLowercase(false)
    {};
};
```

#include <iostream>
#include <seqan/arg_parse.h>

struct ModifyStringOptions {
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), toUppercase(false), toLowercase(false)
int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    addArgument(parser, seqan::ArgParseArgument{
        seqan::ArgParseArgument::STRING, "TEXT"});
    addOption(parser, seqan::ArgParseOption{
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"});
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption{
        "U", "uppercase", "Select to-uppercase as operation."});
    addOption(parser, seqan::ArgParseOption{
        "L", "lowercase", "Select to-lowercase as operation."});

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    ModifyStringOptions options;
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    getArgumentValue(options.text, parser, 0);

    std::cout << "period \t \t" << options.period << ‘\n’
              << "uppercase\t" << options.toUppercase << ‘\n’
              << "lowercase\t" << options.toLowercase << ‘\n’
              << "text \t\t" << options.text << ‘\n’;

    return 0;
}

**Best Practice: Wrapping Parsing In Its Own Function**

As a next step towards a cleaner program, we should extract the argument parsing into its own function, e.g. call it `parseCommandLine()`. Following the style guide (*SeqAn C++ Code Style*), we first pass the output parameter, then the input parameters. The return value of our function is a `seqan::ArgumentParser::ParseResult` such that we can differentiate whether the program can go on, the help was printed and the program is to exit with success, or there was a problem with the passed argument and the program is to exit with an error code.

Also, note that we should check that the user cannot specify both to-lowercase and to-uppercase. This check cannot be performed by the `ArgumentParser` by itself but we can easily add this check. We add this functionality to the `parseCommandLine()` function.

Click **more...** to see the updated program.
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions()
    {
        period(1), toUppercase(false), toLowercase(false)
    }
};

seqan::ArgumentParser::ParseResult parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument{
        seqan::ArgParseArgument::STRING, "TEXT"});

    // Define Options
    addOption(parser, seqan::ArgParseOption{
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"});
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption{
        "U", "uppercase", "Select to-uppercase as operation.");
    addOption(parser, seqan::ArgParseOption{
        "L", "lowercase", "Select to-lowercase as operation.");

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;

    // Extract option values.
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    getValueValue(options.text, parser, 0);

    // If both to-uppercase and to-lowercase were selected then this is an error.
    if (options.toUppercase && options.toLowercase)
    {
        std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n";
        return seqan::ArgumentParser::PARSE_ERROR;
    }

    return seqan::ArgumentParser::PARSE_OK;
}
```c
int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    std::cout << "period \t" << options.period << ‘\n’
               << "uppercase\t" << options.toUppercase << ‘\n’
               << "lowercase\t" << options.toLowercase << ‘\n’
               << "text \t" << options.text << ‘\n’;

    return 0;
}
```

### Feature-Complete Example Program

The command line parsing part of our program is done now. Let us now add a function `modifyText()` that is given a `ModifyStringOptions` object and text and modifies the text. We simply use the C standard library functions `toupper()` and `tolower()` from the header `<cctype>` for converting to upper and lower case.

```c
#include <iostream>
#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions():
        period(1), toUppercase(false), toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT")
    );

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INT));
    setDefaultValue(parser, "period", "1");
```
addOption(parser, seqan::ArgParseOption("U", "uppercase", "Select to-uppercase as operation."));  
addOption(parser, seqan::ArgParseOption("L", "lowercase", "Select to-lowercase as operation."));  

// Parse command line.  
seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);  

// Only extract options if the program will continue after parseCommandLine()  
if (res != seqan::ArgumentParser::PARSE_OK)  
    return res;  

// Extract option values.  
getOptionValue(options.period, parser, "period");  
options.toUppercase = isSet(parser, "uppercase");  
options.toLowercase = isSet(parser, "lowercase");  
seqan::getArgumentValue(options.text, parser, 0);  

// If both to-uppercase and to-lowercase were selected then this is an error.  
if (options.toUppercase && options.toLowercase)  
    {  
        std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n";  
        return seqan::ArgumentParser::PARSE_ERROR;  
    }  

return seqan::ArgumentParser::PARSE_OK;  

seqan::CharString modifyString(seqan::CharString const & text,  
                ModifyStringOptions const & options)  
{  
    seqan::CharString result;  
    
    if (options.toLowercase)  
    {  
        for (unsigned i = 0; i < length(text); ++i)  
            if (i % options.period == 0u)  
                appendValue(result, tolower(text[i]));  
            else  
                appendValue(result, text[i]);  
    }  
    else  
    {  
        for (unsigned i = 0; i < length(text); ++i)  
            if (i % options.period == 0u)  
                appendValue(result, toupper(text[i]));  
            else  
                appendValue(result, text[i]);  
    }  

    return result;  
}  

int main(int argc, char const ** argv)
// Parse the command line.
ModifyStringOptions options;
seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

// If parsing was not successful then exit with code 1 if there were errors.
// Otherwise, exit with code 0 (e.g. help was printed).
if (res != seqan::ArgumentParser::PARSE_OK)
    return res == seqan::ArgumentParser::PARSE_ERROR;

std::cout << modifyString(options.text, options) << '\n';

return 0;

Setting Restrictions

One nice feature of the ArgumentParser is that it is able to perform some simple checks on the parameters. We can:

- check numbers for whether they are greater/smaller than some limits,
- mark options as being required, and
- setting lists of valid values for each option.

In this section, we will give some examples.

Setting Minimum and Maximum Values

The functions setMinValue and setMaxValue allow to give a smallest and/or largest value for a given option. Of course, this only works with integer- and double-typed command line options.

We can pass both the short and the long option name to these functions. The value is given as a string and parsed the same as parameters on the command line.

seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption{
    "i", "integer-value", "An integer option",
    seqan::ArgParseArgument::INTEGER, "INT"});

setMinValue(parser, "i", "10");
setMaxValue(parser, "integer-value", "20");

Assignment 4

Setting min-value on --period

Type Reproduction

Objective Use the function setMinValue to set a minimal value of 1 for the parameter --period.

Solution

#include <iostream>
#include <seqan/arg_parse.h>
struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), toUppercase(false), toLowercase(false)
    {};
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setMinValue(parser, "period", "1");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "r", "range", "Range of the text to modify.",
        seqan::ArgParseArgument::INTEGER, "INT", false, 2));
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;

    // Extract option values.
    getOptionValue(options.period, parser, "period");
    getOptionValue(options.rangeBegin, parser, "range", 0);
    getOptionValue(options.rangeEnd, parser, "range", 1);
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    seqan::getArgumentValue(options.text, parser, 0);

    // If both to-uppercase and to-lowercase were selected then this is an error.
    if (options.toUppercase && options.toLowercase)
    {
        std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n";
        return seqan::ArgumentParser::PARSE_ERROR;
    }
}
return seqan::ArgumentParser::PARSE_OK;
}

seqan::CharString modifyString(seqan::CharString const & text, 
    ModifyStringOptions const & options)
{
    seqan::CharString result;
    if (options.toLowercase)
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i >= options.rangeBegin && i < options.rangeEnd &&
            (i % options.period == 0u))
                appendValue(result, tolower(text[i]));
            else
                appendValue(result, text[i]);
        }
    }
    else
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i >= options.rangeBegin && i < options.rangeEnd &&
            (i % options.period == 0u))
                appendValue(result, toupper(text[i]));
            else
                appendValue(result, text[i]);
        }
    }
    return result;
}

int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    std::cout << modifyString(options.text, options) << '\n';

    return 0;
}

Marking Options as Required

We can mark options as being required using the function setRequired:

seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption{
  "i", "integer-value", "An integer option",
Setting List of Valid Values

Sometimes, it is useful to give a list of valid values for a command line option. You can give it as a space-separated list in a string to `setValidValues`. The check whether the value from the command line is valid is case sensitive.

```
seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption("", "distance-model", "Distance model, either HAMMING or EDIT.",
    seqan::ArgParseArgument::STRING, "STR"));
setValidValues(parser, "distance-model", "HAMMING EDIT");
```

More Option and Argument Types

There are two slightly more special option and argument types: paths to input/output files and tuple values.

Input/Output File Names

We could use `seqan::ArgParseArgument::STRING` to specify input and output files. However, there are two special argument/option types `seqan::ArgParseArgument::INPUTFILE` and `seqan::ArgParseArgument::OUTPUTFILE` that are more suitable:

1. In the near future, we plan to add basic checks for whether input files exist and are readable by the user. You will still have to check whether opening was successful when actually doing this but the program will fail earlier if the source file or target location are not accessible. The user will not have to wait for the program to run through to see that he mistyped the output directory name, for example, and you do not have to write this check.

2. For workflow engine integration, the input and output file options and arguments will be converted into appropriate input and output ports of the nodes.

3. You can use the previously introduced restrictions to specify what kind of files you expect and the ArgumentParser will check while parsing if the correct file type was provided.

Here is an example for defining input and output file arguments:

```
addOption(parser, seqan::ArgParseOption("I", "input-file", "Path to the input file",
    seqan::ArgParseArgument::INPUTFILE, "IN"));
addOption(parser, seqan::ArgParseOption("O", "output-file", "Path to the output file",
    seqan::ArgParseArgument::OUTPUTFILE, "OUT"));
```

The restrictions are added by defining the expected file extension.

```
setValidValues(parser, "input-file", "txt");
setValidValues(parser, "output-file", "txt");
```

Again multiple values are provided as space-separated list. Note that the file ending check is case insensitive, so you do not need to provide `txt` and `TXT`.

You can simply read the values of these options as you would read string options:
Assignment 5

Using File Command Line Options

**Type** Reproduction

**Objective** Replace the argument TEXT by a a command line option \(-I\)/--input-file in the program above. The program should then read in the text instead of using the command line argument.

**Hint** We will also replace the text member of ModifyStringOptions, you might wish to do the same.

**Solution**

```cpp
#include <iostream>
#include <seqan/arg_parse.h>

struct ModifyStringOptions {
    unsigned period;
    unsigned rangeBegin, rangeEnd;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString inputFileName;

    ModifyStringOptions() :
        period(1), rangeBegin(0), rangeEnd(0), toUppercase(false),
        toLowercase(false) {}
};

seqan::ArgumentParser::ParseResult parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv) {
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // Define Options
    addOption(parser, seqan::ArgParseOption("I", "input-file",
        "A text file that will printed with the modifications applied.",
        seqan::ArgParseArgument::INPUTFILE));
    setValidValues(parser, "input-file", "txt");
    setRequired(parser, "input-file");

    addOption(parser, seqan::ArgParseOption("i", "period",
        "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setMinValue(parser, "period", "1");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption("U", "uppercase",
        "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption("L", "lowercase",
        "Select to-lowercase as operation."));

    // Parse Command Line Arguments
    parseCommandLine(parser, argc, argv, options);
    return parser => parser;
}
```
// Parse command line.
seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

// Only extract options if the program will continue after parseCommandLine()
if (res != seqan::ArgumentParser::PARSE_OK)
    return res;

// Extract option values.
getOptionValue(options.period, parser, "period");
options.toUppercase = isSet(parser, "uppercase");
options.toLowercase = isSet(parser, "lowercase");
getOptionValue(options.inputFileName, parser, "input-file");

// If both to-uppercase and to-lowercase were selected then this is an error.
if (options.toUppercase && options.toLowercase)
{
    std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n";
    return seqan::ArgumentParser::PARSE_ERROR;
}

return seqan::ArgumentParser::PARSE_OK;

seqan::CharString modifyString(seqan::CharString const & text,
                               ModifyStringOptions const & options)
{
    seqan::CharString result;

    if (options.toLowercase)
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendValue(result, tolower(text[i]));
            else
                appendValue(result, text[i]);
        }
    }
    else
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendValue(result, toupper(text[i]));
            else
                appendValue(result, text[i]);
        }
    }

    return result;
}

int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);
// If parsing was not successful then exit with code 1 if there were errors.
// Otherwise, exit with code 0 (e.g. help was printed).
if (res != seqan::ArgumentParser::PARSE_OK)
    return res == seqan::ArgumentParser::PARSE_ERROR;

std::fstream inFile(toCString(options.inputFileName), std::ios::binary | std::ios::in);
if (inFile.good())
{
    std::cerr << "ERROR: Could not open input file " << options.inputFileName << '\n';
    return 1;
}
seqan::CharString text;
while (inFile.good())
{
    char c = inFile.get();
    if (inFile.good())
        appendValue(text, c);
}
std::cout << modifyString(text, options);
return 0;

Tuples

We can define an ArgParseArgument and ArgParseOption to be a tuple with a fixed number of arguments. For example, an integer pair (tuple with two entries) could describe a range:

addOption(parser, seqan::ArgParseOption(
    "r", "range", "The range to modify.",
    seqan::ArgParseArgument::INTEGER, "BEGIN END",
    false, 2));

We add two parameters after the label "BEGIN END" for the documentation. First, we specify that the option is not a list option (false) and second, that we need exactly two numbers for it.

The user can now use the parameter as follows:

# modify_string -r 5 10 ...

We use the four-parameter variant with an integer index of getOptionValue to access the entries in the tuple given on the command line.

unsigned rangeBegin = 0, rangeEnd = 0;
getOptionValue(rangeBegin, parser, "range", 0);
getOptionValue(rangeEnd, parser, "range", 1);

Assignment 6

Using Tuple Command Line Options

Type Reproduction

Objective Add a command line option --range to the ArgumentParser in the program above. Modify the function modifyString() such that only parameters in the given range are changed.
Hint We will add two unsigned members rangeBegin and rangeEnd to the ModifyStringOptions struct, you might wish to do the same.

Solution

```cpp
#include <iostream>
#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    unsigned rangeBegin, rangeEnd;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), rangeBegin(0), rangeEnd(0), toUppercase(false),
        toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setMinValue(parser, "period", "1");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation.");
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation.");

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract  options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;

    // Extract option values.
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    seqan::getArgumentValue(options.text, parser, 0);

    // If both to-uppercase and to-lowercase were selected then this is an error.
    if (options.toUppercase && options.toLowercase)
```
{ std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n"; return seqan::ArgumentParser::PARSE_ERROR; }

return seqan::ArgumentParser::PARSE_OK;
}

seqan::CharString modifyString(seqan::CharString const & text, ModifyStringOptions const & options)
{
    seqan::CharString result;
    if (options.toLowercase)
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendString(result, tolower(text[i]));
            else
                appendString(result, text[i]);
        }
    }
    else
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendString(result, toupper(text[i]));
            else
                appendString(result, text[i]);
        }
    }
    return result;
}

int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;
    std::cout << modifyString(options.text, options) << '\n';
    return 0;
}

Embedding Rich Documentation

Another very useful feature of ArgumentParser is that you can embed rich documentation into your programs. You can set the short description, the version string, date, synopsis and add text documentation settings.
Let us first set the **short description**, **version string**, and **date** in our program from above. We insert the following lines just after the declaration of the variable `parser`.

```cpp
setShortDescription(parser, "String Modifier");
setVersion(parser, "1.0");
setDate(parser, "July 2012");
```

After the line with `setDate()`, we give a usage line and add to the description. This information will go to the Synopsis section of the program help.

```cpp
addUsageLine(parser,
             "[\fIOPTIONS\fP] "\fITEXT\fP");
addDescription(parser,
               "This program allows simple character modifications to "
               "each i-th character.");
```

**Tip:** Formatting Command Line Documentation

The formatting of command line parameters might seem strange, at first: **Font operators** start with `\f` (which means that they start with "\f" in in C++ string literals). The `\f` is followed by the **format specifier**. The format specifier can be one of `I`, `B`, and `P`. `I` selects italic text (underlined on the shell), `B` selects bold and `P` resets the formatting to normal text. These font operators are legacies of man pages from Unix and offered a simple-to-implement solution to text formatting.

For example, "Words \fBwere\fP made for \fIbeing\fP written!" would result in the formatted string “Words *were* made for *being* written!”

Note that formatting the command line relies on **ANSI escape codes** which is not supported by modern Windows versions. If you are using Windows, you will not see bold or underlined text.

The argument parser will add some options of its own, for example for printing the help and displaying version information. To separate our arguments from the autogenerated ones, we add the following line. This line will introduce the section “Modification Options” in the Description section of the output.

```cpp
addSection(parser, "Modification Options");
```

Finally, we will add a section with examples. Add the following lines just before the line with the `parse()` function call.

```cpp
addTextSection(parser, "Examples");
addListItem(parser,
            "\fBmodify_string\fP \fB-U\fP \fIveryverylongword\fP",
            "Print upper case version of "
            "veryverylongword.");
addListItem(parser,
            "\fBmodify_string\fP \fB-L\fP \fB-i\fP \fI3\fP \fIveryverylongword\fP",
            "Print "
            "veryverylongword" with every third character "
            "converted to upper case.");
```

That were a lot of changes! Click *more...* to see the complete program.

```cpp
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
};
```

---

**2.1. Tutorial**

259
bool toLowercase;
seqan::CharString text;

ModifyStringOptions() :
    period(1), toUppercase(false), toLowercase(false)
{};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    // Set short description, version, and date.
    setShortDescription(parser, "String Modifier");
    setVersion(parser, "1.0");
    setDate(parser, "July 2012");

    // Define usage line and long description.
    addUsageLine(parser,
        "\[\[\{\{OPTS\}\} \"\{\ITEXT\}\]\]");
    addDescription(parser,
        "This program allows simple character modifications to 
        each i-th character.");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument{
        seqan::ArgParseArgument::STRING, "TEXT"});

    // Define Options -- Section Modification Options
    addSection(parser, "Modification Options");
    addOption(parser, seqan::ArgParseOption{
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"});
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption{
        "U", "uppercase", "Select to-uppercase as operation.");
    addOption(parser, seqan::ArgParseOption{
        "L", "lowercase", "Select to-lowercase as operation.");

    // Add Examples Section.
    addTextSection(parser, "Examples");
    addListItem(parser,
        "\"\"modify_string\"-U\"\"veryverylongword\"\"",
        "Print upper case version of \"veryverylongword\"."");
    addListItem(parser,
        "\"\"modify_string\"-L\"\"i\"\"3\"\"veryverylongword\"",
        "Print \"veryverylongword\" with every third character 
        converted to upper case.");

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;
}
// Extract option values.
getOptionValue(options.period, parser, "period");
options.toUppercase = isSet(parser, "uppercase");
options.toLowercase = isSet(parser, "lowercase");
seqan::getArgumentValue(options.text, parser, 0);

// If both to-uppercase and to-lowercase were selected then this is an error.
if (options.toUppercase && options.toLowercase)
{
    std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n";
    return seqan::ArgumentParser::PARSE_ERROR;
}

return seqan::ArgumentParser::PARSE_OK;

seqan::CharString modifyString(seqan::CharString const & text,
                                ModifyStringOptions const & options)
{
    seqan::CharString result;
    if (options.toLowercase)
    {
        for (unsigned i = 0; i < length(text); ++i)
            appendValue(result, tolower(text[i]));
    }
    else
    {
        for (unsigned i = 0; i < length(text); ++i)
            appendValue(result, toupper(text[i]));
    }

    return result;
}

int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    std::cout << modifyString(options.text, options) << '\n';

    return 0;
}

Let us look at the resulting documentation. Simply call the new program with the --help option.

# modify_string --help
modify_string - String Modifier
================================

SYNOPSIS
modify_string [OPTIONS] "TEXT"

DESCRIPTION
This program allows simple character modifications to each i-th character.

-h, --help
Displays this help message.
--version
Display version information

Modification Options:
-i, --period INT
   Period to use for the index.
-U, --uppercase
   Select to-uppercase as operation.
-L, --lowercase
   Select to-lowercase as operation.

EXAMPLES
modify_string -U veryverylongword
   Print upper case version of "veryverylongword"
modify_string -L -i 3 veryverylongword
   Print "veryverylongword" with every third character converted to upper case.

VERSION
modify_string version: 1.0
Last update July 2012

Also, there is an undocumented option called --export-help that is automatically added by ArgumentParser. You can call it with the values html and man. If the option is set then the argument parser will print the documentation as HTML or man format (man pages are a widely used format for Unix documentation).

You can pipe the output to a file:

# modify_string --export-help html > modify_string.html
# modify_string --export-help man > modify_string.man

HTML can be displayed by any web browser, man pages can be displayed using the program man. Note that when opening a file using man, you have to give the file name either as an absolute or a relative path. Otherwise, it would try to look up the topic modify_string.man. To view the generated man page use:

# man ./modify_string.man

Below, you can see a part of the rendered HTML and man pages generated by the commands above.
modify_string

String Modifier

Synopsis

modify_string [OPTIONS] "TEXT"

Description

This program allows simple character modifications to each i-th character.

-h, --help
Displays this help message.

--version
Display version information

Modification Options:

-l, --period INT
Period to use for the index.

For further reading, have a look at the ArgumentParser class.

2.1.27 Genome Annotations

Learning Objective  You will learn how to work with annotations in SeqAn. After this tutorial, you will be able to write your own programs using annotations and analyzing them. You will be ready to continue with the Fragment Store Tutorial, e.g. if you want to combine your annotations with information from alignments.

Difficulty  Average

Duration  1 h
Prerequisites *Sequences, Iterators*

This tutorial will present SeqAn’s efficient and easy-to-use data structures to work with annotations. They allow to annotate genome regions with features like ‘gene’, ‘mRNA’, ‘exon’, ‘intron’ and if required with custom features. We will give you an understanding of how to load annotations from a GFF or GTF file, store them in efficient data structures, as well as how to traverse and access these information.

**AnnotationStore as Part of the FragmentStore**

This section will give you a short introduction to data structures relevant for working with annotations.

In SeqAn, annotations are stored in the so-called *annotationStore*, which is part of the *FragmentStore*. The annotationStore can only be used together with the FragmentStore, because the latter stores additional information, e.g. the contig names or sequences. The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation.

The FragmentStore can be seen as a database, where each table (called “store”) is implemented as a *String*. Each row of the table corresponds to an element in the string. The position of each element in the string implicitly represents the Id of such element in the table. All such strings are members of the class FragmentStore, are always present and empty if unused. For example, the member *contigStore* is a string of elements, each one containing among others a contig sequence.

For detailed information about the FragmentStore read the *Fragment Store* Tutorial.

Accordingly, the *annotationStore* is a *String*, where each element represents one annotation. Each element holds the necessary information, e.g. beginPos, endPos, parentId etc., as data members.

**AnnotationStore**

In this section you will learn how to work with the *annotationStore* itself.

Annotations are represented hierarchically by a tree having at least a root node.

A typical annotation tree looks as follows.

![Annotation tree example](image)

**Figure 2.3: Annotation tree example**

The following entity-relationship diagram shows the tables holding store annotations, their relationships and cardinalities.
The instantiation of an `annotationStore` happens implicitly with the instantiation of a `FragmentStore`. Therefore we simply type:

```cpp
FragmentStore<> store;
```

### Loading an Annotation File

Before we deal with the actual annotation tree, we will first describe how you can easily load annotations from a GFF or GTF file into the `FragmentStore`.

An annotation file can be read from an open input stream with the function `read`. A tag specifies if we want to read a GFF, GTF or UCSC file. The following example shows how to read an GTF file:

```cpp
// Open input stream from the current directory
std::ifstream file("example.gtf", std::ios_base::in | std::ios_base::binary);
// Read annotations from the GTF file
read(file, store, Gtf());
```

The GFF-reader is also able to detect and read GTF files. The UCSC Genome Browser uses two separate files, the `knownGene.txt` and `knownIsoforms.txt`. They must be read by two consecutive calls of `read` (first `knownGene.txt` then `knownIsoforms.txt`).

**Tip:** An annotation can be loaded without loading the corresponding contigs.

In that case empty contigs are created in the `contigStore` with names given in the annotation. A subsequent call of `loadContigs` would load the sequences of these contigs, if they have the same identifier in the contig file.
Traversing the Annotation Tree

This section will illustrate how to use iterators to traverse the annotation tree.

The annotation tree can be traversed and accessed with the AnnotationTree Iterator. Again we use the metafunction dox::ContainerConcept#Iterator Iterator to determine the appropriate iterator type for our container. A new AnnotationTree iterator can be obtained by calling begin with a reference to the FragmentStore and the AnnotationTree tag:

```cpp
Iterator<FragmentStore<>, AnnotationTree<>>::Type it;
it = begin(store, AnnotationTree<>());
```

The AnnotationTree iterator starts at the root node and can be moved to adjacent tree nodes with the functions goDown, goUp, and goRight. These functions return a boolean value that indicates whether the iterator could be moved. The functions isLeaf, isRoot, isLastChild return the same boolean without moving the iterator. With goRoot or goTo the iterator can be moved to the root node or an arbitrary node given its annotationId. If the iterator should not be moved but a new iterator at an adjacent node is required, the functions nodeDown, nodeUp, nodeRight can be used.

```cpp
// Move the iterator down to a leaf
while (goDown(it));
// Create a new iterator and if possible move it to the right sibling of the first iterator
Iterator<FragmentStore<>, AnnotationTree<>>::Type it2;
if (isLastChild(it))
    it2 = nodeRight(it);
```

The AnnotationTree iterator supports a preorder DFS traversal and therefore can also be used in typical begin-end loops with the functions goBegin (= goRoot), goEnd, goNext, atBegin, atEnd. During a preorder DFS, the descent into subtree can be skipped by goNextRight, or goNextUp which proceeds with the next sibling or returns to the parent node and proceeds with the next node in preorder DFS.

```cpp
// Move the iterator back to the beginning
goBegin(it);
// Iterate over the nodes in preorder DFS while the end is not reached and
// output if the current node is a leaf
for (goBegin(it); atEnd(it); goNext(it))
{
    if (isLeaf(it))
        std::cout << " current node is leaf" << std::endl;
}
```

Assignment 1

Type Review

Objective Copy the code below, which loads the annotations from a given GTF file into the FragmentStore and initializes an iterator on the AnnotationTree. Download the GTF file assignment_annotations.gtf, whose annotations build an AnnotationTree of the typical structure with gene, mRNA and exon level. Adjust the code to go down to the exon level and iterate over all children of the first mRNA and count them. Print the result.

Click more... to see the code.

```cpp
#include <fstream>
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/store.h>
```
using namespace seqan;

int main()
{
  FragmentStore<> store;
  std::ifstream file("assignment_annotations.gtf", std::ios_base::in | std::ios_base::binary);
  read(file, store, Gtf());
  // Create AnnotationTree iterator
  Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
  it = begin(store, AnnotationTree<>());
  // Move iterator one node down
  goDown(it);

  return 0;
}

Hints  In the given data the left-most leaf is a child of mRNA and has siblings. You can use the function goRight
to traverse over all siblings.

Solution  Click more... to see one possible solution.

Assignment 2

Type  Review

Objective  Reuse the code and the GTF file from above. Instead of counting only the children of the first mRNA
adjust the code to count the children for each given mRNA. Print the results.

Hints  After you reached the last child of the first mRNA you can use the functions goNext and goDown to traverse to
the next leaf.
Solution  
Click more... to see one possible solution.

```cpp
#include <fstream>
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    FragmentStore<> store;
    std::ifstream file("assignment_annotations.gtf", std::ios_base::in | std::ios_base::binary);
    read(file, store, Gtf());
    // Iterate over all leafs, count and print the result
    Iterator<FragmentStore<>, AnnotationTree<>>::Type it = begin(store, AnnotationTree<>());
    unsigned count = 0;
    std::cout << "Number of children for each mRNA: " << std::endl;
    // Go down to the first leaf (first child of the first mRNA)
    while (!atEnd(it))
    {
        ++count;
        // Iterate over all siblings and count
        while (goRight(it))
        {
            ++count;
            std::cout << count << std::endl;
            count = 0;
        }
        // Jump to the next mRNA or gene, go down to its first leaf and count it
        if (!atEnd(it))
        {
            goNext(it);
            if (!atEnd(it))
                while (goDown(it))
            {
                ++count;
                std::cout << count << std::endl;
                count = 0;
            }
        }
    }
    return 0;
}
```

Accessing the Annotation Tree

Let us now have a closer look how to access the information stored in the different stores representing the annotation tree.

To access or modify the node an iterator points at, the iterator returns the node’s annotationId by the `value` function (`== operator*`). With the annotationId the corresponding entry in the annotationStore could be modified manually or by using convenience functions. The function `getAnnotation` returns a reference to the corresponding entry in the annotationStore. `getName` and `setName` can be used to retrieve or change the identifier of the annotation element. As some annotation file formats don’t give every annotation a name, the function `getUniqueName` returns the name if non-empty or generates one using the type and id. The name of the parent node in the tree can be determined with `getParentName`. The name of the annotation type, e.g. ‘mRNA’ or ‘exon’, can be determined and modified with `getType` and `setType`.

Assume we have loaded the file `example.gtf` with the following content to the `FragmentStore` `store` and instantiated the iterator `it` of the corresponding annotation tree.
We now want to iterate to the first exon and output a few information:

```cpp
// Move the iterator to the begin of the annotation tree
it = begin(store, AnnotationTree<>());
// Go down to exon level
while (goDown(it)) {
    std::cout << "type: " << getType(it) << std::endl;
    std::cout << "id: " << value(it) << std::endl;
    std::cout << "begin position: " << getAnnotation(it).beginPos << std::endl;
}
```

For our example the output would be:

type: exon
id: 3
begin position: 149

An annotation can not only refer to a region of a contig but also contain additional information given as key-value pairs. The value of a key can be retrieved or set by `getValueByKey` and `assignValueByKey`. The values of a node can be cleared with `clearValues`.

A new node can be created as first child, last child, or right sibling of the current node with `createLeftChild`, `createRightChild`, or `createSibling`. All three functions return an iterator to the newly created node.

```cpp
Iterator<FragmentStore<>, AnnotationTree<> >::Type it2;
// Create a right sibling of the current node and return an iterator to this new node
it2 = createSibling(it);
```

The following list summarizes the functions provided by the `AnnotationTree` iterator.

- `getAnnotation`, `value` Return annotation object/id of current node
- `getName`, `setName`, `getType`, `setType` Access name or type of current annotation object
- `getParentName` Access parent name of current annotation object
- `clearValue`, `getValueByKey`, `assignValueByKey` Access associated values
- `goBegin`, `goEnd`, `atBegin`, `atEnd` Go to or test for begin/end of DFS traversal
- `goNext`, `goNextRight`, `goNextUp` Go next, skip subtree or siblings during DFS traversal
- `goRoot`, `goUp`, `goRight` Navigate through annotation tree
- `createLeftChild`, `createRightChild`, `createSibling` Create new annotation nodes
- `isRoot`, `isLeaf` Test for root/leaf node

### Assignment 3

#### Type Application

**Objective** Again use the given GTF file `assignment_annotations.gtf` and create an iterator on the annotation tree. Now iterate to the first node of type “exon” and output the following features:

1. type
2. begin position
3. end position
4. its Id  
5. the Id of its parent  
6. the name of its parent  

**Solution** Click more... to see one possible solution.

```c++
#include <fstream>
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    FragmentStore< > store;
    std::ifstream file("assignment_annotations.gtf", std::ios_base::in | std::ios_base::binary);
    read(file, store, Gtf());
    // Create iterator
    Iterator<FragmentStore< >, AnnotationTree< > >::Type it;
    it = begin(store, AnnotationTree< >());
    // Iterate to the first annotation of type "exon"
    while (!atEnd(it) && getType(it) != "exon") goNext(it);
    // Output:
    std::cout << " type: " << getType(it) << std::endl;
    std::cout << " begin position: " << getAnnotation(it).beginPos << std::endl;
    std::cout << " end position: " << getAnnotation(it).endPos << std::endl;
    std::cout << " id: " << value(it) << std::endl;
    std::cout << " parent id: " << getAnnotation(it).parentId << std::endl;
    std::cout << " parent name: " << getParentName(it) << std::endl;
    return 0;
}
```

Assignment 4  

**Objective** Write a small statistic tool to analyse a given set of annotations.

1. Load the annotations given in the GTF file `assignment_annotations.gtf`.  
2. Output the average number of mRNAs for genes.  
3. Output the average number of exons for mRNAs.  
4. Additionally output the average exon length.  
5. Test your program also on large data, e.g. the annotation of the mouse genome [raw-attachment:Mus_musculus.NCBIM37.61.gtf.zip:wiki:Tutorial/SimpleRnaSeq Mus_musculus.NCBIM37.61.gtf.zip] (don’t forget to unzip first).

**Solution** Click more... to see one possible solution.
```cpp
#include <fstream>
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/store.h>
using namespace seqan;

int main()
{
    FragmentStore<> store;
    std::ifstream file("assignment_annotations.gtf", std::ios_base::in | std::ios_base::binary);
    read(file, store, Gtf());
    // Create iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
    unsigned countGenes = 0;
    unsigned countmRNAs = 0;
    unsigned countExons = 0;
    unsigned length = 0;
    // Iterate over annotation tree and count different elements and compute exon lengths
    while (!atEnd(it)) {
        if (getType(it) == "gene") ++countGenes;
        else if (getType(it) == "mRNA") ++countmRNAs;
        else if (getType(it) == "exon") {
            ++countExons;
            length += abs((int)getAnnotation(it).endPos - (int)getAnnotation(it).beginPos);
        }
        goNext(it);
    }
    // Output some stats:
    std::cout << "Average number of mRNAs for genes: " << (float)countmRNAs/(float)countGenes << std::endl;
    std::cout << "Average number of exons for mRNAs: " << (float)countExons/(float)countmRNAs << std::endl;
    std::cout << "Average length of exons: " << (float)length/(float)countExons << std::endl;
    return 0;
}
```

**Average number of mRNAs for genes:** 1.5  
**Average number of exons for mRNAs:** 3  
**Average length of exons:** 95.5556

### Write an Annotation File

To write an annotation to an open output stream use the function `write` and specify the file format with a tag `Gff()` or `Gtf()`.

```cpp
// Open output stream
std::ofstream fileOut("example_out.gtf", std::ios_base::out | std::ios_base::binary);
// Write annotations to GTF file
write(fileOut, store, Gtf());
```
Learning Objective  This article will give you an overview of the I/O infrastructure in SeqAn. You will learn how the different pieces work together and how to open files for reading/writing and create `RecordReader` objects for reading.

Difficulty  Advanced

Duration  20 min

Prerequisites  `Basic Sequence I/O, Indexed FASTA I/O, Basic SAM and BAM I/O`

This article is not a real tutorial but more of a survey of the I/O functionality in SeqAn. It is targeted at more advanced developers that want to learn more about how I/O works in SeqAn and get background information.

The article first gives a rough overview of the I/O functionality in SeqAn. This is then followed by an overview of the `StreamConcept` concept, the `StreamConcept` class and adaptions to the `StreamConcept` concept. Finally, the `RecordReader` class is introduced and you learn about the record and document reading API. After reading this tutorial, you will have an overview about the I/O layers in SeqAn. You can then proceed to read the `File I/O` and the `Parsing` tutorials to learn more about Streams and RecordReaders.

Overview

The following picture shows an overview of the I/O and parsing related Layers in SeqAn. Note that the aim of this tutorial is not explaining each detail of the I/O system in SeqAn but giving you a good first impression of the I/O layers. These impression should give you a good overview of the I/O system.

High-Level I/O

The topmost layer High-Level I/O provides easy-to-use APIs for reading and writing files. This comes at the potential cost of some performance and genericity. The reason for this cost is that programming techniques such as virtual function calls have to be used that may inflict performance costs. The loss of genericity is caused by the simpler APIs having fewer configuration options. Usually, code on this layer wraps the layer Lower-Level I/O which is one level lower and simplifies its usage. The API on this layer consists of one (or very few) classes and few functions for each supported file type.

Here is an example for using the high-level I/O:
We use the `SequenceStream` class to open a gzip-compressed FASTA file. If this succeeds, we read one record (identifier and sequence) from the sequence stream. The exact usage is explained in the Basic Sequence I/O Tutorial.

### Lower-Level I/O

The second-highest layer Lower-Level I/O provides a more basic and verbose API for reading and writing files. Usually, the main logic of the I/O code is implemented on this layer and this code uses templates and no virtual functions to implement functionality in a generic way. This leads to code that can be highly optimized by the compiler.

One characteristic of this layer is using tags such as `Fasta()` for `Fastq()` in function calls to distinguish between formats. In contrast, the format selection is hidden in the high-level API.

For many formats, we can simply write to a `Stream` (a concept from the Implementation Support Layer introduced below) but for reading, we need a `RecordReader` object.

In the following example we open a FASTA file directly using `std::fstream`. Then, we write one FASTA record to the stream. The example below looks similar to the example above that used `SequenceStream`. The main difference is that the file format is not detected automatically but the code is fixed to read FASTA only. Also, the code below does not allow reading from compressed files.

Note that it would be possible to change the code to also read FASTQ by first detecting the file format manually and
then calling `readRecord()` with the tag `seqan::Fastq()` instead of `seqan::Fasta()` if the file format is FASTQ. It would also be possible to first manually detect whether the file is compressed and use `GzFileStream` to read from a gzip compressed stream. In this case, the user has full control about the code and thus the performance but the code gets much more verbose.

```cpp
// Open file for writing using the standard C++ IOStreams library.
std::ofstream outStream("filename.fasta", std::ios::out | std::ios::binary);
if (!outStream)
    std::cerr << "Could not open filename.fasta\n";

// Write one record.
seqan::CharString id = "seq1";
seqan::Dna5String seq = "CGATCCAGATGT";
if (writeRecord(seqStream, id, seq, seqan::Fasta()) != 0)
    std::cerr << "There was an error writing to filename.fasta\n";
```

The `RecordReader` class provides important wrapping functionality for underlying file representations (let us stick to streams for now). You usually open a stream and then create a `Single-Pass RecordReader`. You then read using this reader:

```cpp
// Open file for reading using the standard C++ IOStreams library.
std::ifstream inStream("filename.fasta", std::ios::in | std::ios::binary);
if (!inStream)
    std::cerr << "Could not open filename.fasta\n";

// Create record reader.
seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(inStream);

// Read one record from the reader.
seqan::CharString id;
seqan::Dna5String seq;
if (!readRecord(id, seq, reader, Fasta()))
    std::cerr << "There was an error reading from filename.fasta\n";
```

These two layers are meant to be used by SeqAn library users directly. The layers below are used by the high-level I/O and lower level I/O routines. Library users only get in contact with them when they initiate reading or writing files using the lower-level API. When `writing`, users open a stream and then use the SeqAn I/O routines to write records to the stream. When `reading`, users open a stream, create a `RecordReader` and then read records from the stream via the `RecordReader`.

### Implementation Support

The Implementation Support layers contain:

1. Adaptions of system library provided file and stream routines to the SeqAn `StreamConcept` concept.
2. The `RecordReader` hierarchy for wrapping streams. They provide functionality that is important for the efficient implementation of I/O code.
3. Code for tokenization and parsing, based on `RecordReader` objects.
4. Conversion from textual number representations to numeric values (aka “lexical casting”).

The Implementation Support Layers will be described in detail in the `Parsing` Tutorial.
System / External Libraries

The System / External Libraries layer contains libraries provided by the operating system or third-party libraries such as zlib <http://zlib.net> for compression. These libraries are used by the layers above to actually access the files on disk.

Streams

Most data formats in bioinformatics simply contain a series of records, often preceded by a header. The most important use case if reading sequentially over such a data stream, with the occasional jump. In computer science, it is common to call the abstraction to such data sources streams. In SeqAn, the concept StreamConcept provides an interface for such stream data types.

SeqAn provides adaptions from the standardized C and C++ file interfaces to the StreamConcept concept. You can use the I/O streams from the <iostream> library, the string buffer streams from the <sstream> library as well as the FILE * type from the <stdio> library. Furthermore, SeqAn provides the Stream class and specializations for accessing char arrays and zlib and bzip compressed files as streams. The I/O Overview tutorial gives a more detailed introduction into this topic.

You already saw above how the streams from the <iostream> library can be used for reading and writing. Using the adaption in the SeqAn library, we can simply use FILE * instead of std::fstream. The following example uses std::fstream.

```
// Open file for writing using the standard C++ IOStreams library.
std::fstream outStream("filename.fasta", std::ios::binary | std::ios::out);
if (outStream.good())
    std::cerr << "Could not open filename.fasta\n";

// Write one record, this will write the following to filename.fasta:
// >id1
// CGATTTTT
if (write2(outStream, "id1", "CGATTTTT", Fasta()) != 0)
    std::cerr << "There was an error writing to filename.fasta\n";
fclose(outStream);
```

Now, we simply replace the constructor of std::fstream by calls to fopen() and of course have to close the file again manually using fclose(). We use ferror() instead of the member function good(). The modified example looks as follows.

```
// Open file for reading using the standard C++ IOStreams library.
FILE * inStream = fopen("filename.fasta", "rb");
if (ferror(inStream))
    std::cerr << "Could not open filename.fasta\n";

// Create record reader.
seqan::RecordReader<FILE *, seqan::SinglePass<> > reader(inStream);

// Read one record from the reader.
seqan::CharString id;
seqan::Dna5String seq;
if (readRecord(id, seq, reader, Fasta()))
    std::cerr << "There was an error reading from filename.fasta\n";
fclose(inStream);
```

Important: Always Use Binary File Open Mode
Note that we open the files in binary mode above in all cases. On Unix, it makes no difference whether binary mode is used or not: You always read exactly what is written on the disk and you write to the disk exactly what you have in memory.

On Windows, however, there is a difference. When opening a file in non-binary, and thus “text” mode, all line endings will automatically be converted into Windows style. If your file contains \n, you will actually read \r\n. “This wreaks havoc when getting or setting the position in the current file.”

Thus: “Always open your file in binary mode.” This will avoid weird crashes and malfunction on Windows.

Record Readers

The RecordReader class is a wrapper around file-like objects, such as Streams. However, as you will learn in the File I/O tutorial, it also provides a generalized interface for memory mapped files that are accessed as MMap String objects. Depending on the actual specialization, [RecordReader RecordReaders] also add a buffer which allows to re-read the first several thousand characters of a file which is very useful for automatic file type detection.

When using the RecordReader class in the Lower-Level I/O layer, you only have to know how to create a RecordReader for reading. You do not have to know how to implement parsing functionality with them.

Constructing RecordReader objects is easy. You parametrize the class template with the underlying stream type (e.g. FILE * or std::fstream) and choose the specialization SinglePass<>. To the constructor, you pass the stream object as the only argument. Note that the file already has to be opened for reading.

```cpp
std::fstream inputFile("filename.fasta", std::ios::input | std::ios::binary);
seqan::RecordReader<std::fstream, seqan::SinglePass<>> reader(inputFile);
```

You can learn about how to implement parsers using the RecordReader API in the Parsing Tutorial.

Record vs. Document I/O

Most file formats in bioinformatics are structured as lists of records. Often, they start out with a header that itself contains different header records. For example, the SAM format starts with an optional header where users can specify the contigs of their reference sequence. This header is then followed by a list of records storing the same type of information. In the case of SAM, for example, each record contains the information of a read alignment, in the case of FASTQ, each record contains a sequence identifier, the sequence itself, and base qualities.

Generally, there are two important use cases for I/O in bioinformatics. (1) Processing a file record by record, such that only one or a few records are stored in memory. This approach is useful when dealing with large files such as NGS read files or whole SAM or BAM alignment files. (2) Reading all of a file into main memory. This approach is useful for reading a smaller file into main memory, for example for multiple sequence alignment of proteins or for leading a chunk of a NGS read file to build a q-gram index of the reads.

This leads to the two types of APIs in SeqAn. Record-reading and document reading of files.

When reading a file record-by-record, the function readRecord() is overloaded. The function overloads are structured as follows:

- The first parameters are the output parameters where the read data is written to.
- Where applicable, this is followed by a state or context object for the given file type (see below for more detail).
- This is followed by the RecordReader object to read from.
- Where applicable, a tag is used to indicate the file format.
- The function returns an integer status code. 0 is returned to indicate that no error occurred while 1 is returned to indicate an error.
This is best explained with an example. When reading FASTQ, the first three parameters are the strings to store the identifier, sequence characters, and base qualities into. The fourth parameter is the RecordReader object to read the data from. The last parameter seqan::Fastq() is a tag indicating the file format.

```cpp
std::fstream in("file.fq", std::ios::binary | std::ios::in);
seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

seqan::CharString id;
seqan::Dna5String seq;
seqan::CharString qual;
int res = readRecord(id, seq, qual, reader, seqan::Fastq());
```

When reading SAM, we first have to read the header. Then, we can read a record from the file which is passed as the first parameter. For reading the record, we have to pass a BamIOContext object that is necessary for translating the reference name to its numeric id. This is then followed by the RecordReader object to read from and the tag seqan::Sam() to specify the format. The full initialization of the BamAlignmentRecord is a bit complex and omitted below.

```cpp
std::fstream in("file.sam", std::ios::binary | std::ios::in);
seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

seqan::BamIOContext<TNameStore> context(refNameStore, refNameStoreCache);
seqan::BamHeader header;
int res1 = readRecord(header, context, reader, seqan::Sam());

seqan::BamAlignmentRecord record;
int res2 = readRecord(record, context, reader, seqan::Sam());
```

When reading a whole file into main memory, an overload of the function read() is used. The parameter order is the same as in case of readRecord() but the types are Strings or StringSets of the types of the readRecord() variant. For example, consider the following example of reading a whole FASTA file into main memory.

```cpp
std::fstream in("file.fa", std::ios::binary | std::ios::in);
seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

seqan::StringSet<seqan::CharString> ids;
seqan::StringSet<seqan::Dna5String> seqs;
int res = read2(ids, seqs, reader, seqan::Fasta());
```

For writing files, SeqAn provides overloads of the function write() and writeRecord(). Their usage is similar. However, the parameter order is different, accordingly to the SeqAn coding style guide:

- The first parameter is the output stream to write the data to.
- This is followed by the data to write out.
- Where applicable, this is followed by a context object.
- Where applicable, a tag is used to indicate the file format.
- The function returns an integer status code. 0 is returned to indicate that no error occurred while 1 is returned to indicate an error.

**Next Steps**

If you want, you can now have a look at the API documentation of the StreamConcept concept and its implementations as well as the documentation of the RecordReader class.

There are two “tracks” in this section of the tutorials which you can follow. First, you can now read the tutorials for specific already supported file formats.
2.1.29 Sequence File I/O

**Learning Objective**  This tutorial will explain how to read and write sequence files in SeqAn using the `RecordReader` class and the `MultiSeqFile` API.

**Difficulty**  Advanced

**Duration**  30 min

**Prerequisites**  I/O Overview

This tutorial explains how to read using the `RecordReader` class and the `MultiSeqFile` API and how to write sequence files using `Streams`. Currently, SeqAn supports reading sequences (and qualities where it applies) in FASTA, FASTQ, EMBL and GenBank format.

This tutorial does not explain how to use the `SequenceStream` class. This class is covered in the *Basic Sequence I/O* tutorial.

**Reading Sequence Files**

Because sequences play a central role in the SeqAn, the library provides a comprehensive implementation of input routines. The sequence I/O facilities are available through the `seq_io` module. You get access to this module by

```
#include <seqan/seq_io.h>
```

We will first describe the API for record-wise reading of sequence files and then go over to reading all records in a file at once.
We start out by creating a `RecordReader` object. For most cases, the single-pass specialization is the most appropriate one. Then, we can read the file record-by-record using the function `readRecord`.

To this function, we pass the buffers we want to read the record identifier and the sequence into `(id` and `seq`). This is followed by the `reader` object we create with the `std::fstream` previously opened. The last argument is the tag `seqan::Fasta()` which uses the variant of `readRecord` for reading FASTA files.

```
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2) // Invalid number of arguments.
        return 1;

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file record-wise.
    seqan::CharString id;
    seqan::Dna5String seq;
    while (!atEnd(reader))
    {
        if (readRecord(id, seq, reader, seqan::Fasta()) != 0)
            return 1; // Could not record from file.

        std::cout << id << std::endl << seq << std::endl;
    }

    return 0;
}
```

When reading FASTQ files, we use the tag `seqan::Fastq()`. For storing the qualities, we can pass an optional third parameter of type `CharString` which stores the quality values.

```
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2) // Invalid number of arguments.
        return 1;

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file record-wise.
    seqan::CharString id;
    seqan::Dna5String seq;
    seqan::CharString qual;
    while (!atEnd(reader))
    ```
```cpp
int main(int argc, char const * argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file record-wise.
    seqan::CharString id;
    seqan::String<seqan::Dna5Q> seq;
    while (!atEnd(reader))
    {
        if (readRecord(id, seq, reader, seqan::Fastq()) != 0)
            return 1; // Could not record from file.

        std::cout << id << "\t" << seq << "\n";
    }

    return 0;
}
```

Optionally, we can also read the sequence into a string of [dox:Dna5Q Dna5Q] characters which will store the qualities directly in the string’s characters.

```cpp
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>
```

```cpp
int main(int argc, char const * argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file record-wise.
    seqan::CharString id;
    seqan::String<seqan::Dna5Q> seq;
    while (!atEnd(reader))
    {
        if (readRecord(id, seq, reader, seqan::Fastq()) != 0)
            return 1; // Could not record from file.

        std::cout << id << "\t" << seq << "\n";
    }

    return 0;
}
```

**Important:** Sequence Parsing Behaviour

- When using Dna5 or Dna5Q as the sequence’s alphabet type, the parsing routine will allow the characters ‘C’, ‘G’, ‘A’, ‘T’, and ‘N’ in the sequences of the file. This can make problems if the sequence contains different characters, for example when it contains IUPAC characters. In this case, you can simply use CharString as the seq parameter and then assign them to a Dna5String.

- Accordingly, when using Dna or DnaQ, only the characters ‘C’, ‘G’, ‘A’, and ‘T’ are allowed.

- When omitting the qual parameter when reading FASTQ, the quality values from the file will be ignored.

**Assignment 1**

Record-Wise Reading Sequences into CharString
Type Review

Objective Modify the example above to read the sequence into a CharString instead of a Dna5String.

Solution

```cpp
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file record-wise.
    seqan::CharString id;
    seqan::CharString seq;
    while (!atEnd(reader))
    {
        if (readRecord(id, seq, reader, seqan::Fastq()) != 0)
            return 1; // Could not record from file.

        std::cout << id << "\t" << seq << "\n";
    }

    return 0;
}
```

When we want to read a whole sequence (e.g. FASTA or FASTQ) file into memory then we only have to slightly adjust the example from above. For example, here is how we can read a whole FASTQ file into memory using the function read into StringSets of CharStrings and Dna5Strings.

```cpp
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file in one pass.
    seqan::StringSet<seqan::CharString> ids;
```
Assignment 2

Document-Wise Reading Sequences into CharString

Type Review

Objective Modify the example above to read the sequence into a StringSet of CharStrings instead of a Dna5Strings.

Solution

```cpp
#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Read file in one pass.
    seqan::StringSet<seqan::CharString> ids;
    seqan::StringSet<seqan::CharString> seqs;
    seqan::StringSet<seqan::CharString> quals;
    if (read2(ids, seqs, quals, reader, seqan::Fastq()) != 0)
        return 1; // Could not read file.

    for (unsigned i = 0; i < length(ids); ++i)
    {
        std::cout << ids[i] << ' ' << seqs[i] << ' ' << quals[i] << '
';
    }
    return 0;
}
```

Choice of Record Reader

In most cases, you will want to use a Single-Pass RecordReader for reading files. Mostly, it is the fastest and best way to read files and also all file formats have a single-pass implementation.

Using a double-pass record reader almost only makes sense if read a whole file into main memory using the document reading API. The file is read twice. In the first pass, the total length of ids and sequence characters is determined. When reading sequences into StringSets, the exact number of elements can be reserved. Even more, when using
Concat-Direct StringSet, no superfluous memory has to be allocated at all. The string sets are then filled in the second pass.

Using double-pass I/O also only makes sense for document reading when used in conjunction with MMap Strings. When using streams, the RecordReader has to buffer the read data in memory because not all stream implementation allow for jumping. In the case of MMap Strings, no buffer is used because the record reader directly operates on the memory mapped file (and thus directly on the disk buffers of the kernel).

**Assignment 3**

Using a Double-Pass RecordReader with a MMap String.

**Type** Application

**Objective** Change solution of Assignment 2 such that a Double-Pass RecordReader is used with a MMap String.

**Hint** You can open files into MMap Strings as follows (include the <seqan/file.h> header):

```cpp
typedef seqan::String<char, seqan::MMap<> > TMMapString;
TMMapString mmapString;
bool success = open(mmapString, "filename.fa", seqan::OPEN_RDONLY);
```

You can then define a DoublePassRecordReader wrapping the just opened `mmapString` as follows:

```cpp
typedef seqan::RecordReader<
    TMMapString,
    seqan::DoublePass<seqan::StringReader> > TReader;
TReader reader(mmapString);
```

**Solution**

```cpp
#include <fstream>
#include <iostream>
#include <seqan/file.h>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open memory mapped string.
    seqan::String<char, seqan::MMap<> > mmapString;
    if (!open(mmapString, argv[1], seqan::OPEN_RDONLY))
        return 1; // Could not open file.

    // Create RecordReader.
    seqan::RecordReader<seqan::String<char, seqan::MMap<> >, ,
        seqan::DoublePass<seqan::StringReader> > reader(mmapString);

    // Read file in one pass.
    seqan::StringSet<seqan::CharString> ids;
    seqan::StringSet<seqan::CharString> seqs;
    seqan::StringSet<seqan::CharString> quals;
    if (read2(ids, seqs, quals, reader, seqan::Fastq()) != 0)
        return 1; // Could not read file.
}
```
for (unsigned i = 0; i < length(ids); ++i)
    std::cout << ids[i] << ' ' << seqs[i] << ' ' << quals[i] << '
';

return 0;

Auto Format Detection

Passing the format as the tag is appropriate when the format is known beforehand. Otherwise, you can use a variable of type AutoSeqStreamFormat instead of the tag.

AutoSeqStreamFormat objects can be first passed to the function guessStreamFormat. This function tries to parse the file as different formats on the first some thousand bytes. When this succeeds, the successfully recognized file type is stored in the object.

You can then subsequently use the AutoSeqStreamFormat instead of a tag to the functions readRecord or read.

#include <fstream>
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open file and create RecordReader.
    std::fstream in(argv[1], std::ios::binary | std::ios::in);
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(in);

    // Create the AutoSeqStreamFormat object and guess the file format.
    seqan::AutoSeqStreamFormat formatTag;
    if (!guessStreamFormat(reader, formatTag))
        return 1; // Could not detect file format.

    // Read file record-wise.
    seqan::CharString id;
    seqan::String<seqan::Dna5Q> seq;
    while (!atEnd(reader))
    {
        if (readRecord(id, seq, reader, formatTag) != 0)
            return 1; // Could not record from file.

        std::cout << id << " \n " << seq << " \n ";
    }

    return 0;
}

Assignment 4

Using AutoSeqStreamFormat

Type Application
**Objective**  Adjust the solution of Assignment 3 to use an `AutoSeqStreamFormat` for format detection.

**Solution**

```cpp
#include <fstream>
#include <iostream>
#include <seqan/file.h>
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1; // Invalid number of arguments.

    // Open memory mapped string.
    seqan::String<
        char,
        seqan::MMap<> > mmapString;
    if (!open(mmapString, argv[1], seqan::OPEN_RDONLY))
        return 1; // Could not open file.

    // Create RecordReader.
    seqan::RecordReader<
        seqan::String<
            char,
            seqan::MMap<> >,
        seqan::DoublePass<seqan::StringReader> > reader(mmapString);

    // Create the AutoSeqStreamFormat object and guess the file format.
    seqan::AutoSeqStreamFormat formatTag;
    if (!guessStreamFormat(reader, formatTag))
        return 1; // Could not detect file format.

    // Read file in one pass.
    seqan::StringSet<seqan::CharString> ids;
    seqan::StringSet<seqan::CharString> seqs;
    seqan::StringSet<seqan::CharString> quals;
    if (read2(ids, seqs, quals, reader, formatTag) != 0)
        return 1; // Could not read file.

    for (unsigned i = 0; i < length(ids); ++i)
        std::cout << ids[i] << ' ' << seqs[i] << ' ' << quals[i] << '
';

    return 0;
}
```

**Note:** Qualities and FASTA files.

When passing a `qual` parameter to `readRecord` or `read` then this cannot be filled with qualities from the file since FASTA files do not contain any. Instead, the `qual` string will be empty after the call to `readRecord` and after the call to `read`, it will be a string set with empty entries. The string set will have a size that is equal to the number of records in the file.

**Writing Sequence Files**

Similar to reading, sequence files can be written record-by-record or as a whole.

For record-wise writing, we use the function `writeRecord`. This function expects as parameters, the `StreamConcept` to write to, the data to write, followed by the format tag. The following example writes an identifier and a sequence `StringSet` record-by-record to stdout.
#include <seqan/seq_io.h>
#include <iostream>

int main()
{
    seqan::StringSet<seqan::CharString> ids;
    appendValue(ids, "id1");
    appendValue(ids, "id2");
    seqan::StringSet<seqan::Dna5String> seqs;
    appendValue(seqs, "CGATCGATCGAT");
    appendValue(seqs, "AAAAAAAAAAAA");

    for (unsigned i = 0; i < length(ids); ++i)
    {
        if (seqan::writeRecord(std::cout, ids[i], seqs[i], seqan::Fasta()) != 0)
            return 1; // Error writing.
    }

    return 0;
}

The result on the console looks like this:

> id1
CGATCGATCGAT
> id2
AAAAAAAAAAAA

Assignment 5

Writing out FASTQ.

Type Application

Objective Change the example above such that the two sequences are written as FASTQ with qualities. Use the quality strings "IIIIIIIIIIIII" and "IIIIIIIIIIIIIIIIIIII".

Hint Simply use a new StringSet quals of CharString, append the quality strings, and modify the line with the writeRecord() call.

Solution

#include <seqan/seq_io.h>
#include <iostream>

int main()
{
    seqan::StringSet<seqan::CharString> ids;
    appendValue(ids, "id1");
    appendValue(ids, "id2");
    seqan::StringSet<seqan::Dna5String> seqs;
    appendValue(seqs, "CGATCGATCGAT");
    appendValue(seqs, "AAAAAAAAAAAA");
    seqan::StringSet<seqan::CharString> quals;
    appendValue(quals, "IIIIIIIIIIIII");
    appendValue(quals, "IIIIIIIIIIIIIIIIIIII");

    for (unsigned i = 0; i < length(ids); ++i)
    {
        if (seqan::writeRecord(std::cout, ids[i], seqs[i], quals[i], seqan::Fastq()) != 0)
            return 1; // Error writing.
The output looks as follows:

@id1
CGATCGATCGAT
+
IIIIIIIIIIHII
@id2
AAAAAAAAAAAA
+
IIIIIIIIIIII

For writing out whole string sets at once, we use the function write. The transition from record-wise writing to writing whole string sets is of similar simplicity as for reading:

```cpp
#include <seqan/seq_io.h>
#include <iostream>

int main()
{
    seqan::StringSet<seqan::CharString> ids;
    appendValue(ids, "id1");
    appendValue(ids, "id2");
    seqan::StringSet<seqan::Dna5String> seqs;
    appendValue(seqs, "CGATCGATCGAT");
    appendValue(seqs, "AAAAAAAAAAAA");
    seqan::StringSet<seqan::CharString> quals;
    appendValue(quals, "IIIIIIIIIIHII");
    appendValue(quals, "IIIIIIIIIIII");

    if (seqan::write2(std::cout, ids, seqs, quals, seqan::Fastq()) != 0)
        return 1;  // Error writing.

    return 0;
}
```

**Using MultiSeqFile**

**Warning:** Deprecate MultiSeqFile in favour of FaiIndex?

The class MultiSeqFile (which actually is a shortcut to a memory mapped string set) allows to read sequence files in a two-pass approach. First, the file is read and the start positions of each sequence record in the file is stored in memory. The file is kept open as a memory mapped file.

Then, we can access the identifier, sequence, and quality string of a record using functions such as assignSeqId.

Indexed reading can be done through MultiSeqFile which is a shortcut to a memory mapped string set. We open the file using open on its concat member (which is a MMap String). The function split then parses the file contents and sets the separating indexes of the StringSet. For this, we need the file format. We could give a specify format in the tag (e.g. seqan::Fastq()) or use AutoSeqFormat together with guessFormat.
The following example demonstrates how to use MultiSeqFile to read sequence files. First, we include the necessary headers and start our main() function.

```cpp
#include <seqan/file.h>
#include <iostream>

int main (int argc, char const ** argv)
{
    // Include necessary headers

    seqan::MultiSeqFile multiSeqFile;
    if (argc < 2 || !open(multiSeqFile.concat, argv[1], seqan::OPEN_RDONLY))
        return 1;

    seqan::AutoSeqFormat format;
    guessFormat(multiSeqFile.concat, format);

    split(multiSeqFile, format);

    unsigned seqCount = length(multiSeqFile);
    seqan::StringSet<seqan::String<seqan::Dna5Q> > seqs;
    seqan::StringSet<seqan::CharString> seqIDs;
    reserve(seqs, seqCount, seqan::Exact());
    reserve(seqIDs, seqCount, seqan::Exact());

    seqan::String<seqan::Dna5Q> seq;
    seqan::CharString qual;
    seqan::CharString id;

    for (unsigned i = 0; i < seqCount; ++i)
    {
        assignSeq(seq, multiSeqFile[i], format); // read sequence
        assignQual(qual, multiSeqFile[i], format); // read ascii quality values
        assignSeqId(id, multiSeqFile[i], format); // read sequence id

        for (unsigned j = 0; j < length(qual) && j < length(seq); ++j)
            assignQualityValue(seq[j], (int)(seqan::ordValue(qual[j]) - 33));
    }
}
```
Finally, we return the status code 0 at the end of our main() function.

    return 0;
}

Indexed reading has multiple advantages.

- Its performance is only slightly worse than when reading sequentially with a double-pass String RecordReader.
- The input file is mapped into main memory and otherwise complicated page-wise memory management is done by the operating system and does not have to be implemented by the user. The user can access the file almost at random and only the used parts will be loaded into main memory. This is quite efficient when only few sequences are needed.

If you need to have fast random access to all sequences in a file then loading it into a Concat-Direct StringSet with the batch-reading API is faster than using MultiSeqFile.

MultiSeqFile Review

Type Review

Objective Change the example above, so the sequence file that is read is written to the user in a TSV format. For each record in the input file with id \( \text{ID} \), sequence \( \text{SEQ} \), and quality string \( \text{QUAL} \), write out a line \( \text{ID}\t\text{SEQ}\t\text{QUAL} \).

Solution

```cpp
#include <seqan/file.h>
#include <iostream>

int main (int argc, char const ** argv)
{
    seqan::MultiSeqFile multiSeqFile;
    if (argc < 2 || !open(multiSeqFile.concat, argv[1], seqan::OPEN_RDONLY))
        return 1;

    seqan::AutoSeqFormat format;
    guessFormat(multiSeqFile.concat, format);
    split(multiSeqFile, format);

    seqan::String<seqan::Dna5> seq;
    seqan::CharString qual;
    seqan::CharString id;

    for (unsigned i = 0; i < seqCount; ++i)
    {
        assignSeq(seq, multiSeqFile[i], format); // read sequence
        assignQual(qual, multiSeqFile[i], format); // read ascii quality values
        assignSeqId(id, multiSeqFile[i], format); // read sequence id

        std::cout << id << '\t' << seq << '\t' << qual << '\n';
    }
}  
```
return 0;
}

Next Steps

- Read the Wikipedia articles about the FASTA file format and the FASTQ file format and quality values to refresh your knowledge.
- Read the Basic Sequence I/O tutorial to learn how to use the SequenceStream class.
- Read the Indexed FASTA I/O tutorial tutorial to learn how to read FASTA files efficiently in a random-access fashion.
- Continue with Tutorial.

2.1.30 SAM and BAM I/O

**Learning Objective**  This tutorial will explain how to use the lower-level API for reading from and writing to SAM and BAM files.

**Difficulty**  Advanced

**Duration**  20 min

**Prerequisites**  Basic SAM and BAM I/O, I/O Overview

After reading the tutorial-basic-sam-bam-io tutorial, you should have a good understanding of the representation of SAM/BAM records in SeqAn library. This tutorial will explain how the lower-level implementation of SAM and BAM I/O works and which design decisions lead to this. You will also learn about the classes BamIOContext and NameStoreCache. This article also explains how to write template-based code that can read both SAM files from RecordReaders and BAM files from compressed streams. Furthermore, you will learn how to use BAI indices for jumping in BAM files.

This tutorial will mostly work by showing full source code examples and explaining them. Do not feel intimidated by its length, it is inflated by the amount of embedded source code.

The usage of SAM and BAM tags is explained in the Basic SAM and BAM I/O tutorial and will not be discussed here further.
Important: Note that this tutorial is targeted at readers that already know about the SAM format. If you do not know about the SAM format yet then this tutorial will be harder for your to understand. Teaching the ins and outs of SAM is out of the scope of such a tutorial.

**BamAlignmentRecord Design Overview**

Besides the fact that BAM is a compressed, binary format, the main difference to the plain-text format SAM is that BAM has additional header information. The binary header of BAM files contains a plain-text SAM header with the same information as a SAM file. However, BAM files contain an additional binary header section that stores all reference sequence names and their length.

This is equivalent to the @SQ entries in a SAM header but the @SQ entries are optional in both SAM and the SAM header in a BAM file. Also, the @SQ entries in the SAM header of a BAM file can be out of sync with the binary reference information entries. The authoritative source for this information (most prominently the names and the order of the reference sequences). BAM records then only contain an integer rID for each reference that refers to the rID-th reference sequence in the binary header.

Because SAM and BAM files store the same information, there is only one record type to store it in SeqAn. The record type is the class **BamAlignmentRecord** that was already introduced in the *Basic SAM and BAM I/O* tutorial tutorial. The structure of this class is designed after the BAM file so conversion from the BAM file on disk to the in-memory representation is fast. For example, the tags are stored as a binary string, the same as in a BAM file. When reading SAM, the tags are converted into the BAM representation whereas BAM tags can be copied verbatim. There is one important deviation, though: The qualities are stored using a phred-style ASCII encoding for consistencies with the rest of the SeqAn library.

As a reminder, here is the synopsis of the **BamAlignmentRecord** class again.

```cpp
namespace seqan {

class BamAlignmentRecord {
{
public:
    CharString qName; // QNAME
    __uint16 flag; // FLAG
    __int32 rID; // REF
    __int32 beginPos; // POS
    __uint8 mapQ; // MAPQ mapping quality, 255 for */invalid
    __uint16 bin; // bin for indexing
    String<CigarElement<> > cigar; // CIGAR string
    __int32 rNextId; // RNEXT (0-based)
    __int32 pNext; // PNEXT (0-based)
    __int32 tLen; // TLEN
    CharString seq; // SEQ, as in SAM/BAM file.
    CharString qual; // Quality string as in SAM (Phred).
    CharString tags; // Tags, raw as in BAM.

    // Constants for marking pos, reference id and length members invalid (== */0).
    static __int32 const INVALID_POS = -1;
    static __int32 const INVALID_REFID = -1;
    static __int32 const INVALID_LEN = 0;
};
} // namespace seqan
```
Name Stores and Name Store Caches

In order to translate from numeric reference id (rID) to text reference sequence name, the names have to be stored in a StringSet which we will call a name store. For being able to translate back from a textual name (stored as a CharString, for example), we need a NameStoreCache that allows the fast lookup of numeric ids from textual names. Both the name store and the cache are then wrapped by a BamIOContext. This context object is used to prescient from the differences of SAM and BAM files when reading and writing.

For example, when writing out a BamAlignmentRecord to a SAM file, we need to look up the name of the reference from its numeric id to write it out as a string. When reading a record from a SAM file, we have to translate its name string into a numeric id. Even more, if the sequence is not know yet (remember, the @SQ headers are optional), we have to append it to the name store and register it with the cache.

Here is a minimal example of setting up a name store, name store cache, and a BamIOContext. We will build upon this example below when showing how to read and write SAM and BAM files.

```cpp
#include <seqan/bam_io.h>
#include <seqan/sequence.h>

int main()
{
    // Create some shortcuts to the types that we will use.
    typedef seqan::StringSet<seqan::CharString> TNameStore;
    typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
    typedef seqan::BamIOContext<TNameStore> TBamIOContext;

    // Setup the variables.
    TNameStore nameStore;
    TNameStoreCache nameStoreCache(nameStore);
    TBamIOContext bamIOContext(nameStore, nameStoreCache);

    return 0;
}
```

BGZF Files / Stream

By default, the BAM format is compressed using the BGZF compression scheme (originating from Tabix, but also described in the SAM standard). You can read BGZF files with tools for processing .gz files, e.g. gzip and zcat.

However, there is a big difference between files written in BGZF and .gz files. BGZF is a sequence of compressed blocks. If the offset of a block is known, it can be decompressed independent of the rest of the file. This information can then be used together with indices.

SeqAn provides the BGZF Stream class in the module <seqan/stream.h> to access such streams. Here is an example for using a Stream for reading:

```cpp
#include <seqan/stream.h>

int main()
{
    seqan::Stream<seqan::Bgzf> stream;
    if (!open(stream, "filename.bam", "r"))
        return 1; // Could not open for reading.

    return 0;
}
```

Using a BGZF Stream for writing:
Assignment 1

Uncompressing a BGZF file.

Type Review

Objective Write a program that reads in a BGZF compressed file using BGZF Stream and writes the uncompressed data out again.

Hint Use the function streamReadBlock and streamWriteBlock for reading and writing data into and from buffers.

Solution

```cpp
#include <iostream>
#include <fstream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv) {
    if (argc != 3) {
        std::cerr << "USAGE: " << argv[0] << " IN.bam OUT.bin\n";
        return 1;
    }

    // Open BGZF file for reading.
    seqan::Stream<seqan::Bgzf> inStream;
    if (!open(inStream, argv[1], "r")) {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Open std::fstream for writing.
    std::ofstream outStream(argv[2], std::ios::binary | std::ios::out);
    if (!outStream) {
        std::cerr << "ERROR: Could not open " << argv[2] << " for writing.\n";
        return 1;
    }

    // Copy over data.
    seqan::CharString buffer;
    resize(buffer, 1000);
    while (!seqan::atEnd(inStream) && seqan::streamError(inStream) == 0) {
```
Reading and Writing Headers

The data structure `BamHeader` has already been described in the *Basic SAM and BAM I/O* so we will not repeat that here. Instead, we will focus on how to read headers from SAM and BAM files.

Here is a minimal example of reading and writing a header from and to a SAM file. The example contains the creation of a `BamIOContext`, the necessary `RecordReader` and full error handling.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 3)
    {
        std::cerr << "USAGE: " << argv[0] << " IN.sam OUT.sam\n";
        return 1;
    }

    // Open std::fstream for reading.
    std::fstream inStream(argv[1], std::ios::binary | std::ios::in);
    if (!inStream.good())
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Open std::fstream for writing.
    std::fstream outStream(argv[2], std::ios::binary | std::ios::out);
    if (!outStream.good())
    {
        std::cerr << "ERROR: Could not open " << argv[2] << " for writing.\n";
        return 1;
    }

    // Setup RecordReader.
    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(inStream);

    // Setup name store, cache, and BAM I/O context.
    typedef seqan::StringSet<seqan::CharString> TNameStore;
    typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
    typedef seqan::BamIOContext<TNameStore> TBamIOContext;
    TNameStore nameStore;
    TNameStoreCache nameStoreCache(nameStore);
    TBamIOContext context(nameStore, nameStoreCache);
}
```
Reading and writing headers from and to BAM files is simple. We simply replace seqan::Sam() by seqan::Bam() and use BGZF Stream objects instead of uncompressed streams. Also, we do not need a RecordReader any more.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 3)
    {
        std::cerr << "USAGE: " << argv[0] << " IN.bam OUT.bam\n";
        return 1;
    }

    // Open BGZF Stream for reading.
    seqan::Stream<seqan::Bgzf> inStream;
    if (!open(inStream, argv[1], "r"))
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Open BGZF Stream for writing.
    seqan::Stream<seqan::Bgzf> outStream;
    if (!open(outStream, argv[2], "w"))
    {
        std::cerr << "ERROR: Could not open " << argv[2] << " for writing.\n";
        return 1;
    }

    // Setup name store, cache, and BAM I/O context.
    typedef seqan::StringSet<seqan::CharString> TNameStore;
    typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
    typedef seqan::BamIOContext<TNameStore> TBamIOContext;
    TNameStore nameStore;
```
TNameStoreCache nameStoreCache(nameStore);
TBamIOContext context(nameStore, nameStoreCache);

// Read header.
sean::BamHeader header;
if (readRecord(header, context, inStream, seqan::Bam()) != 0)
{
    std::cerr << "ERROR: Could not read header from BAM file " << argv[1] << "\n";
    return 1;
}

// Write out header again.
if (write2(outStream, header, context, seqan::Bam()) != 0)
{
    std::cerr << "ERROR: Could not write header to BAM file " << argv[2] << "\n";
    return 1;
}

return 0;

Note that except for the types, the signatures of the functions readRecord() and write() are the same. Thus, we can make copying of the header a template function copyHeader(). This function can now be used for both BAM and SAM.

#include <iostream>
#include <fstream>

#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

template <typename TOutStream, typename TInStreamOrReader, typename TTag>
int copyHeader(TOutStream & outStream, TInStreamOrReader & inStreamOrReader, seqan::BamIOContext<seqan::StringSet<seqan::CharString> > & context, TTag const & tag)
{

    // Read header.
    sean::BamHeader header;
    if (readRecord(header, context, inStreamOrReader, tag) != 0)
    {
        std::cerr << "ERROR: Could not read header\n";
        return 1;
    }

    // Write out header again.
    if (write2(outStream, header, context, tag) != 0)
    {
        std::cerr << "ERROR: Could not write header.\n";
        return 1;
    }

    return 0;
}

int main(int argc, char const ** argv)
```cpp
if (argc != 3)
{
    std::cerr << "USAGE: " << argv[0] << " IN.sam OUT.sam\n";
    return 1;
}

// Streams for SAM.
std::fstream inStreamSam, outStreamSam;
// Streams for BAM.
sefan::Stream<sefan::Bgzf> inStreamBam, outStreamBam;

if (sefan::endsWith(sefan::CharString(argv[1]), ".sam"))
{
inStreamSam.open(argv[1], std::ios::binary | std::ios::in);
if (!inStreamSam.good())
{
    std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
    return 1;
}
outStreamSam.open(argv[2], std::ios::binary | std::ios::out);
if (!outStreamSam.good())
{
    std::cerr << "ERROR: Could not open " << argv[2] << " for writing.\n";
    return 1;
}
}
else
{
    // Open BGZF Stream for reading.
    if (!open(inStreamBam, argv[1], "r"))
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Open BGZF Stream for writing.
    if (!open(outStreamBam, argv[2], "w"))
    {
        std::cerr << "ERROR: Could not open " << argv[2] << " for writing.\n";
        return 1;
    }
}

// Setup name store, cache, and BAM I/O context.
typedef sefan::StringSet<sefan::CharString> TNameStore;
typedef sefan::NameStoreCache<TNameStore> TNameStoreCache;
typedef sefan::BamIOContext<TNameStore> TBamIOContext;
TNameStore nameStore;
TNameStoreCache nameStoreCache(nameStore);
TBamIOContext context(nameStore, nameStoreCache);

if (endsWith(sefan::CharString(argv[1]), ".sam"))
{
    // Stream must be open before constructing reader, thus we define it here.
    sefan::RecordReader<std::fstream, sefan::SinglePass<>> reader(inStreamSam);
    return copyHeader(outStreamSam, reader, context, sefan::Sam());
}
```
Assignment 2

Converting BAM header to SAM.

Type  Application

Objective  Write a program that reads the header from a BAM file and writes it out as a SAM header to std::cout.

Solution

```cpp
#include <iostream>
#include <fstream>

#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
    {
        std::cerr << "USAGE: " << argv[0] << " IN.bam
";
        return 1;
    }

    // Open BGZF Stream for reading.
    seqan::Stream<seqan::Bgzf> inStream;
    if (!open(inStream, argv[1], "r"))
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Setup name store, cache, and BAM I/O context.
    typedef seqan::StringSet<seqan::CharString> TNameStore;
    typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
    typedef seqan::BamIOContext<TNameStore> TBamIOContext;
    TNameStore nameStore;
    TNameStoreCache nameStoreCache(nameStore);
    TBamIOContext context(nameStore, nameStoreCache);

    // Read header.
    seqan::BamHeader header;
    if (readRecord(header, context, inStream, seqan::Bam()) != 0)
    {
        std::cerr << "ERROR: Could not read header from BAM file " << argv[1] << "\n";
        return 1;
    }

    // Write out header again.
    if (write2(std::cout, header, context, seqan::Sam()) != 0)
    {
```
Reading and Writing Alignment Records

BamAlignmentRecords can be read and written the same way as BamHeader objects. Here is an example for reading and writing of alignment records from SAM and to files.

```
// Copy over records.
seqan::BamAlignmentRecord record;
while (!atEnd(reader))
{
    if (readRecord(record, context, reader, seqan::Sam()) != 0)
    {
        std::cerr << "ERROR: Could not read record from SAM file " << argv[1] << "\n";
        return 1;
    }

    if (write2(outStream, record, context, seqan::Sam()) != 0)
    {
        std::cerr << "ERROR: Could not write record to SAM file " << argv[2] << "\n";
        return 1;
    }
}
```

And here is the modified version for the BAM format. The only changes are that

- we do not read from a RecordReader but a BGZF Stream instead,
- we need to write to a BGZF Stream, and
- we need to use the tag `seqan::Bam()` instead of `seqan::Sam()`.

```
// Copy over records.
seqan::BamAlignmentRecord record;
while (!atEnd(reader))
{
    if (readRecord(record, context, inStream, seqan::Bam()) != 0)
    {
        std::cerr << "ERROR: Could not read record from BAM file " << argv[1] << "\n";
        return 1;
    }

    if (write2(outStream, record, context, seqan::Bam()) != 0)
    {
        std::cerr << "ERROR: Could not write record to BAM file " << argv[2] << "\n";
        return 1;
    }
}
```

Assignment 3

Converting whole BAM files to SAM.
Type  Application

Objective  Modify the solution of Assignment 2 to not only convert the header to BAM but also the alignment records.

Solution

```cpp
#include <iostream>
#include <fstream>
#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
    {
        std::cerr << "USAGE: " << argv[0] << " IN.bam\n";
        return 1;
    }

    // Open BGZF Stream for reading.
    seqan::Stream<seqan::Bgzf> inStream;
    if (!open(inStream, argv[1], "r"))
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Setup name store, cache, and BAM I/O context.
    typedef seqan::StringSet<seqan::CharString> TNameStore;
    typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
    typedef seqan::BamIOContext<TNameStore> TBamIOContext;
    TNameStore nameStore;
    TNameStoreCache nameStoreCache(nameStore);
    TBamIOContext context(nameStore, nameStoreCache);

    // Read header.
    seqan::BamHeader header;
    if (readRecord(header, context, inStream, seqan::Bam()) != 0)
    {
        std::cerr << "ERROR: Could not read header from BAM file " << argv[1] << "\n";
        return 1;
    }

    // Write out header again.
    if (write2(std::cout, header, context, seqan::Sam()) != 0)
    {
        std::cerr << "ERROR: Could not write header to stdout.\n";
        return 1;
    }

    // Copy over the alignment records.
    seqan::BamAlignmentRecord record;
    while (!atEnd(inStream))
    {
        if (readRecord(record, context, inStream, seqan::Bam()) != 0)
        {
            std::cerr << "ERROR: Could not read record from BAM File " << argv[1] << "\n";
            return 1;
        }
    }
}
```
Using Indices

SeqAn also contains support for reading BAM indices with the format .bai. These indices can be built using the samtools index command.

You can read such indices into a BAI BamIndex object with the function read. Then, you can use the function `seqan::Function.BamIndex#jumpToRegion` to jump within BAM files.

After jumping, the next record that is read is before at the given position. This means, you have to manually read as many records up until the one you are looking for is found. The reason for this is that the function `jumpToRegion` would have to read until it finds the first record that is right from or at the given position. This would lead to this record being lost.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/bam_io.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 7)
    {
        std::cerr << "USAGE: " << argv[0] << " IN.bam IN.bam.bai REF BEGIN END COUNT\n";
        return 1;
    }

    // Open BGZF Stream for reading.
    seqan::Stream<seqan::Bgzf> inStream;
    if (!open(inStream, argv[1], "r"))
    {
        std::cerr << "ERROR: Could not open " << argv[1] << " for reading.\n";
        return 1;
    }

    // Read BAI index.
    seqan::BamIndex<seqan::Bai> baiIndex;
    if (read(baiIndex, argv[2]) != 0)
    {
        std::cerr << "ERROR: Could not read BAI index file " << argv[2] << "\n";
        return 1;
    }

    // Setup name store, cache, and BAM I/O context.
```
typedef seqan::StringSet<seqan::CharString> TNameStore;
typedef seqan::NameStoreCache<TNameStore> TNameStoreCache;
typedef seqan::BamIOContext<TNameStore> TBamIOContext;
TNameStore nameStore;
TNameStoreCache nameStoreCache(nameStore);
TBamIOContext context(nameStore, nameStoreCache);

// Read header.
seqan::BamHeader header;
if (readRecord(header, context, inStream, seqan::Bam()) != 0)
    {std::cerr << "ERROR: Could not read header from BAM file " << argv[1] << "\n";
      return 1;}

// Translate from reference name to rID.
int rID = 0;
if (!getIdByName(nameStore, argv[3], rID, nameStoreCache))
    {std::cerr << "ERROR: Reference sequence named " << argv[3] << " not known.\n";
      return 1;}

// Translate BEGIN and END arguments to number, 1-based to 0-based.
int beginPos = 0, endPos = 0;
if (!seqan::lexicalCast2(beginPos, argv[4]) || beginPos <= 0)
    {std::cerr << "ERROR: Begin position " << argv[4] << " is invalid.\n";
      return 1;}
beginPos -= 1; // 1-based to 0-based.
if (!seqan::lexicalCast2(endPos, argv[5]) || endPos <= 0)
    {std::cerr << "ERROR: End position " << argv[5] << " is invalid.\n";
      return 1;}
endPos -= 1; // 1-based to 0-based.

// Translate number of elements to print to number.
int num = 0;
if (!seqan::lexicalCast2(num, argv[6]))
    {std::cerr << "ERROR: Count " << argv[6] << " is invalid.\n";
      return 1;}

// Jump the BGZF stream to this position.
bool hasAlignments = false;
if (!jumpToRegion(inStream, hasAlignments, context, rID, beginPos, endPos, baiIndex))
      return 1;}
if (!hasAlignments)
    return 0; // No alignments here.

// Seek linearly to the selected position.
seqan::BamAlignmentRecord record;
```cpp
int numPrinted = 0;
while (!atEnd(inStream) && numPrinted < num)
{
    if (readRecord(record, context, inStream, seqan::Bam()) != 0)
    {
        std::cerr << "ERROR: Could not read record from BAM file.\n";
        return 1;
    }

    // If we are on the next reference or at the end already then we stop.
    if (record.rID == -1 || record.rID > rID || record.beginPos >= endPos)
        break;
    // If we are left of the selected position then we skip this record.
    if (record.beginPos < beginPos)
        continue;

    // Otherwise, we print it to the user.
    numPrinted += 1;
    if (write2(std::cout, record, context, seqan::Sam()) != 0)
    {
        std::cerr << "ERROR: Could not write record to stdout.\n";
        return 1;
    }
}

return 0;
```

Next Steps

- Read the SAM Specification (pdf).
- Continue with the Tutorial.
2.1.31 File I/O

Learning Objective In this tutorial, you will learn about the new file I/O infrastructure in SeqAn. You will get an overview of the different layers in the library, an introduction on the StreamConcept concept, the Stream class, and MMap-Strings.

Difficulty Advanced

Duration 60 min

Prerequisites I/O Overview, Indexed FASTA I/O, Basic SAM and BAM I/O

This tutorial introduces the low-level facilities of file I/O in SeqAn:

• There is a concept called StreamConcept in the SeqAn library that stream data types have to implement. There also is the class Stream that provides implementations of the concept together with its specializations. (If you want to provide your own Stream implementation, you should specialize the class Stream).

• Particularly, there are the specializations GzFileStream and BZ2 FileStream that provide access to compressed files.

• Furthermore, SeqAn allows to access memory mapped files using the MMap String specialization.

The target audience consists of developers (1) who want to learn how to use memory mapped files and compressed streams, or (2) who want to have raw, byte-wise read and write access to files, or (3) who want to get a deeper understanding of the I/O system in the SeqAn library.

Note that this tutorial has more of a presentational character with fewer tasks.

Streams

The I/O Overview tutorial has already given you a good overview of streams in SeqAn and how to open them for reading and writing. As a reminder: Always open your streams in binary mode to circumvent problems with getting and setting positions within files on Windows. How exactly you can open files in binary mode depends on the library you are using. Consult the documentation of the library you are using for I/O.

The Stream Concept

The stream concept requires the following functions which work on already files (e.g. FILE *, std::fstream, or Stream objects).

<table>
<thead>
<tr>
<th>Function</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamEof</td>
<td>Return whether stream is at end of file.</td>
</tr>
<tr>
<td>streamError</td>
<td>Return error code of stream.</td>
</tr>
<tr>
<td>streamFlush</td>
<td>Flush stream buffer.</td>
</tr>
<tr>
<td>streamPeek</td>
<td>Get next character from stream without changing the position in the file.</td>
</tr>
<tr>
<td>streamPut</td>
<td>Write a value to the output, converted to string.</td>
</tr>
<tr>
<td>streamBlock</td>
<td>Read a block of char values from the stream.</td>
</tr>
<tr>
<td>streamReadChar</td>
<td>Read one character from the stream.</td>
</tr>
<tr>
<td>streamSeek</td>
<td>Set stream’s location.</td>
</tr>
<tr>
<td>streamTell</td>
<td>Retrieve stream’s location.</td>
</tr>
<tr>
<td>streamWriteBlock</td>
<td>Write an array of char to the stream.</td>
</tr>
</tbody>
</table>

Not all functions might be available for all streams. The metafunction HasStreamFeature provides information about the stream types.
Stream Adoptions

The following C/C++ I/O interfaces can be adapted to the StreamConcept concept.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE*</td>
<td>C standard library files.</td>
</tr>
<tr>
<td>std::fstream, std::ifstream, std::ofstream</td>
<td>C++ iostream library file streams</td>
</tr>
<tr>
<td>std::stringstream, std::istringstream, std::ostringstream</td>
<td>C++ iostream library string streams</td>
</tr>
</tbody>
</table>

This way, we can use the common C++ I/O types through a common interface. Also, we could add adaptions of other file and stream data types to the StreamConcept concept.

The following example shows how to use the StreamConcept global interface functions to copy the contents of the file in.txt to the file out.txt.

```cpp
#include <fstream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main()
{
    std::fstream in("in.txt", std::ios::binary | std::ios::in);
    std::fstream out("out.txt", std::ios::binary | std::ios::out);

    seqan::CharString buffer;
    resize(buffer, 1000);

    while (!seqan::streamEof(in) && seqan::streamError(in) == 0)
    {
        int num = seqan::streamReadBlock(&buffer[0], in, length(buffer));
        seqan::streamWriteBlock(out, &buffer[0], num);
    }

    return 0;
}
```

Assignment 1 Reading / Writing

Type Review

Objective Write a program that accepts three parameters from the command line. The first one should identify the stream type to use (e.g. "file" for FILE* and "fstream" for std::fstream). The second should be either 'r' or 'w' for reading/writing. The third one should be a file name. The program should, depending on the parameters, open the given file name in read/write mode using the given file type. When reading, it should display the file contents on stdout. When writing, it should put the string "Hello world!\n" into the file.

Hint You can encapsulate the reading and writing in their own function templates. This allows you to remove redundancy from the code.

Solution ::

```cpp
#include <iostream>
#include <fstream>
#include <cstdio>
#include <seqan/stream.h>

// This template function reads the contents from the given Stream in and
// writes it out to std::cout

template <typename TStream>
int doReading(TStream & in)
{
    seqan::CharString buffer;
    resize(buffer, 1000);

    while (!seqan::streamEof(in) && (seqan::streamError(in) == 0))
    {
        int num = seqan::streamReadBlock(&buffer[0], in, length(buffer));
        seqan::streamWriteBlock(std::cout, &buffer[0], num);
    }
    return 0;
}

// This template function writes out "Hello World!\n" to the given Stream.

template <typename TStream>
int doWriting(TStream & out)
{
    seqan::CharString buffer = "Hello World!\n"
    return (seqan::streamWriteBlock(out, &buffer[0], length(buffer)) != length(buffer));
}

// The main function parses the command line, opens the files in the
// appropriate modes with the appropriate stream types and then calls either
// doWriting() or doReading().

int main(int argc, char const ** argv)
{
    if (argc != 4)
    {
        std::cerr << "USAGE: " << argv[0] << " [file|fstream] [r|w] FILENAME\n";
        return 1;
    }

    // Check first argument.
    if (seqan::CharString(argv[1]) != "file" && seqan::CharString(argv[1]) != "fstream")
    {
        std::cerr << "ERROR: " << argv[1] << " is not a valid stream type name.\n";
        return 1;
    }
    bool useFile = (seqan::CharString(argv[1]) == "file");

    // Check second argument.
    if (seqan::CharString(argv[2]) != "r" && seqan::CharString(argv[2]) != "w")
    {
        std::cerr << "ERROR: " << argv[2] << " is not a valid operation name.\n";
        return 1;
    }
    bool doRead = (seqan::CharString(argv[2]) == "r");

    // Branches for stream and operation type.
    int res = 0;
    if (useFile) // FILE *
    {
FILE * fp;

if (doRead) // reading
    fp = fopen(argv[3], "rb");
else // writing
    fp = fopen(argv[3], "wb");

if (fp == 0)
{
    std::cerr << "ERROR: Could not open " << argv[3] << "\n";
    return 1;
}

if (doRead) // reading
    res = doReading(fp);
else // writing
    res = doWriting(fp);

fclose(fp);
else // std::fstream
{
    std::fstream stream;

    if (doRead) // reading
        stream.open(argv[3], std::ios::binary | std::ios::in);
    else // writing
        stream.open(argv[3], std::ios::binary | std::ios::out);

    if (!stream.good())
    {
        std::cerr << "ERROR: Could not open " << argv[3] << "\n";
        return 1;
    }

    if (doRead) // reading
        res = doReading(stream);
    else // writing
        res = doWriting(stream);
}

if (res != 0)
    std::cerr << "ERROR: There was an error accessing the file!\n";
return res;


Char Arrays As Streams

Sometimes it is useful to treat variables of type char * or char[] as streams, e.g., for parsing. You can use the Char-Array Stream specialization for this purpose.

char const * str = "me, myself and my pony";
seqan::Stream<seqan::CharArray<char const *>> > wrapper(str, str + strlen(str));
// We can now read from wrapper as if it was a stream.
Compressed Streams

For accessing .gz and .bz2 files, the stream module contains specializations of the class Stream. The main reason for being Stream specializations instead of adaptions is that zlib and bzlib use too generic data types, e.g., void*, where global functions might have unwanted side effects.

Use the following Stream specializations to read and write zlib and bzlib compressed files.

<table>
<thead>
<tr>
<th>Stream Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GZ File Stream</td>
<td>Wraps the zlib functionality for .gz files.</td>
</tr>
<tr>
<td>BZ2 File Stream</td>
<td>Wraps the bzlib functionality for .bz2 files.</td>
</tr>
</tbody>
</table>

zlib files have a decent compression ratio and support quite fast compression and decompression. bz2 files are fairly slow to read and write, although the compression ratio is better. For most bioinformatics applications, you will prefer zlib over bzlib.

If you are using SeqAn’s build system, zlib and libbz2 will be detected automatically. On Linux and Mac Os X, these libraries are usually already installed. If you are using Windows, then you can follow the instructions in Installing Contribs On Windows for installing the libraries. If you are using your own build system, see BuildManual/IntegrationWithYourOwnBuildSystem for the necessary configuration steps.

Both specializations can be constructed with an already open underlying compressed stream, e.g. you can pass the gzFile/BZFILE*, that you want to work on, to the stream. They are meant as very thin wrappers around the handle for the compressed stream. This has the advantage that you have full access to the compression settings etc. and the wrappers only add error flags and so on when necessary. For more convenience, you can also use the open function to open them.

The following example shows (1) how to conditionally enable zlib and bzlib support, (2) how to open gzFile and BZFILE* handles for reading and their corresponding wrappers and (3) the possibilities for error checking.

In the header of the program, we include the zlib and bzlib headers if the correct preprocessor symbols are set. Also, we’ll include the required SeqAn headers.

```cpp
#include <cstdio>
#include <fstream>
#if SEQAN_HAS_ZLIB
#include <zlib.h>
#endif // #if SEQAN_HAS_ZLIB
#if SEQAN_HAS_BZIP2
#include <bzlib.h>
#endif // #if SEQAN_HAS_BZIP2
#include <seqan/basic.h>
#include <seqan/stream.h>
```

The first routine demonstrates how to open a .gz file and write its contents to stdout with full error handling. Note that writing char-by-char is probably not the best idea in a real-world program.

```cpp
int openGz(char const * filename)
{
#if SEQAN_HAS_ZLIB
  seqan::Stream<seqan::GZFile> f;
  if (!open(f, filename, "rb"))
  {
    std::cerr << "ERROR: GZip file has the wrong format!" << std::endl;
    return 1;
  }

  while (!streamEof(f))
  {
    // Totally inefficient char-wise writing of characters from .gz file to stderr.
  }
#else
  // ... just some more code...
#endif
}```
The next routine demonstrates how to open a .bz2 file and write its contents to stdout, again with full error handling.

```cpp
int openBz2(char const * filename)
{
#if SEQAN_HAS_BZIP2
    seqan::Stream<seqan::BZ2File> f;
    if (!open(f, filename, "rb"))
    {
        std::cerr << "ERROR: BZ2 file has the wrong format!" << std::endl;
        return 1;
    }
    // Totally inefficient character-wise writing of characters from .bz2 file to stderr.
    while (!streamEof(f))
    {
        char c = '\0';
        int res = streamReadChar(c, f);
        if (res != 0)
        {
            std::cout << "ERROR: Reading byte from BZ2 file." << std::endl;
            return 1;
        }
        std::cerr << c;
    }
#else // #if SEQAN_HAS_BZIP2
    (void) filename;
    std::cerr << "BZLIB not available!" << std::endl;
#endif // #if SEQAN_HAS_BZIP2
    return 0;
}
```

And finally, the code that calls the functions from above.

```cpp
int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1;
    openGz(argv[1]);
    openBz2(argv[1]);
    return 0;
}
```
Now, let’s test the program. We’ll first create gzip and bzip2 compressed text files and an uncompressed text file. Then, we’ll run our demo program on these files. Note that the BZ2FileStream fails when reading from the file, not when opening the file.

```bash
# echo 'foo' > test.txt
# gzip test.txt
# echo 'bar' > test.txt
# bzip2 test.txt
# echo 'bz' > test.txt
# ./extras/demos/tutorial/stream/tutorial_stream_compression_formats test.txt
ERROR: GZip file has the wrong format!
ERROR: Reading byte from BZ2 file.
# ./extras/demos/tutorial/stream/tutorial_stream_compression_formats test.txt.gz
foo
ERROR: Reading byte from BZ2 file.
# ./extras/demos/tutorial/stream/tutorial_stream_compression_formats test.txt.bz2
ERROR: GZip file has the wrong format!
bar
```

Assignment 2  Writing a File Compression/Decompression Tool

Type  Application

Objective  Write a file compression/decompression tool. The first argument should be the format to read/write, e.g. "gz" for gzip and "bz2" for bzip2. The second argument should be the direction, i.e. “c” for “compress”, “x” for “extract”. The third and fourth arguments should be the source/target files.

Solution

```cpp
#include <iostream>
#include <fstream>
#include <seqan/stream.h>

#if SEQAN_HAS_ZLIB && SEQAN_HAS_BZIP2  // Guard against either not being installed.

// Copy from stream in to the stream out.

template <typename TInStream, typename TOutStream>
int copyStream(TInStream & in, TOutStream & out)
{
    seqan::CharString buffer;
    resize(buffer, 1000);

    while (!seqan::streamEof(in) && (seqan::streamError(in) == 0))
    {
        int num = seqan::streamReadBlock(&buffer[0], in, length(buffer));
        seqan::streamWriteBlock(out, &buffer[0], num);
    }

    return 0;
}

// The main function parses the command line, opens the files in the
// appropriate modes with the appropriate stream types and then calls either
// copyStream.

int main(int argc, char const ** argv)
{
```
if (argc != 5)
{
    std::cerr << "USAGE: " << argv[0] << " [gz|bz2] [c|x] FILE_IN FILE_OUT\n";
    return 1;
}

// Check first argument.
if (seqan::CharString(argv[1]) != "gz" && seqan::CharString(argv[1]) != "bz2")
{
    std::cerr << "ERROR: " << argv[1] << " is not a valid compression format.\n";
    return 1;
}
bool useGzip = (seqan::CharString(argv[1]) == "gz");

// Check second argument.
if (seqan::CharString(argv[2]) != "c" && seqan::CharString(argv[2]) != "x")
{
    std::cerr << "ERROR: " << argv[2] << " is not a valid operation name.\n";
    return 1;
}
bool doCompress = (seqan::CharString(argv[2]) == "c");

// Branches for stream and operation type.
int res = 0;
if (useGzip)
{
    seqan::Stream<seqan::GZFile> gzFileStream;
    std::fstream fileStream;

    if (doCompress)
    {
        fileStream.open(argv[3], std::ios::binary | std::ios::in);
        if (!fileStream.good())
        {
            std::cerr << "ERROR: Could not open file " << argv[3] << "\n";
            return 1;
        }

        if (!open(gzFileStream, argv[4], "w"))
        {
            std::cerr << "ERROR: Could not open file " << argv[4] << "\n";
            return 1;
        }

        res = copyStream(fileStream, gzFileStream);
    }
    else // extract
    {
        if (!open(gzFileStream, argv[3], "r"))
        {
            std::cerr << "ERROR: Could not open file " << argv[3] << "\n";
            return 1;
        }

        fileStream.open(argv[4], std::ios::binary | std::ios::out);
        if (!fileStream.good())
        {
            std::cerr << "ERROR: Could not open file " << argv[4] << "\n";
        }
    }
}
return 1;
}

res = copyStream(gzFileStream, fileStream);
}
else // bz2
{
  seqan::Stream<seqan::BZ2File> bz2FileStream;
  std::fstream fileStream;

  if (doCompress)
  {
    fileStream.open(argv[3], std::ios::binary | std::ios::in);
    if (!fileStream.good())
    {
      std::cerr << "ERROR: Could not open file " << argv[3] << "\n";
      return 1;
    }

    if (!open(bz2FileStream, argv[4], "w"))
    {
      std::cerr << "ERROR: Could not open file " << argv[4] << "\n";
      return 1;
    }

    res = copyStream(fileStream, bz2FileStream);
  }
  else // extract
  {
    if (!open(bz2FileStream, argv[3], "r"))
    {
      std::cerr << "ERROR: Could not open file " << argv[3] << "\n";
      return 1;
    }

    fileStream.open(argv[4], std::ios::binary | std::ios::out);
    if (!fileStream.good())
    {
      std::cerr << "ERROR: Could not open file " << argv[4] << "\n";
      return 1;
    }

    res = copyStream(bz2FileStream, fileStream);
  }

  if (res != 0)
  {
    std::cerr << "ERROR: There was an error reading/writing!\n";
    return res;
  }

#else // #if SEQAN_HAS_ZLIB && SEQAN_HAS_BZIP2

int main()
{
  return 0;
}
Memory Mapped Files

Memory mapped files allow very fast access to files since they enable you to read data with few, if any additional buffers. Wikipedia has a nice article on memory mapped files.

In SeqAn, you access memory mapped files using the MMapString specialization. After opening the mapped string using open, you can access its contents as if you were manipulating a normal String. The following shows a simple example:

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1;

    // Memory mapped string, automatically backed by temporary file.
    seqan::String<char, seqan::MMap<> > str1;
    str1 = "This is the first mapped string!";
    std::cout << str1 << std::endl;

    // Open file as memory mapped string.
    seqan::String<char, seqan::MMap<> > str2;
    if (!open(str2, argv[1], seqan::OPEN_RDONLY))
    {
        std::cerr << "Could not open file " << argv[1] << std::endl;
        return 1;
    }
    std::cout << str2 << std::endl;

    return 0;
}
```

An example execution of the program:

```
# echo 'foo' > test.txt
# ./extras/demos/tutorial/stream/tutorial_mmap_string_example test.txt
This is the first mapped string!
foo
```

Next Steps

- Read Wikipedia’s article on memory mapped files.
- Read the Lexical Casting tutorial to learn how to read text from files that represent numbers (e.g. "100") into values of numeric types such as int.
- Read the Parsing tutorial to learn how to write parsers for your own file formats.
- Continue with the Tutorial.
2.1.32 Lexical Casting

Learning Objective  In this tutorial, you will learn about the lexical casting functionality in SeqAn. Lexical casting allows you to convert numbers stored as strings to their numeric values.

Difficulty  Basic

Duration  15 min

Prerequisites  A First Example, Sequences

Lexical Casting

When reading data from text files all data is usually first stored as strings. For example, a genomic location is often described as a pair of the chromosome name and the position on the chromosome, e.g. "chr1" and "1000". To really make use of the "1000" we have to cast it to an integer.

For this purpose, SeqAn provides the lexicalCast and lexicalCast2 functions. These functions do not cast a value into a related type but convert a CharString into its numeric value.

The function is located in the stream module so we have to include <seqan/stream.h>.

```c
#include <seqan/stream.h>
```

The function lexicalCast converts a string into the type given in the template argument and returns the value after conversion. In case of errors, the result is undefined. This is mainly useful if you know that the value can be converted (e.g. you accepted a sequence of 1-4 digits and want to cast it to int).

```c
seqan::CharString valueAsStr = "1000";
int valueAsInt = lexicalCast<int>(valueAsStr);
// => valueAsInt == 1000
```

The function lexicalCast2 has two parameters. A reference to the numeric destination and the string source. The result is a bool and true indicates success.

```c
bool success = lexicalCast2<int>(valueAsInt, valueAsStr);
// => success == true
// => valueAsInt == 1000
```

Note: Strictness of Lexical Casting

A string such as "123XX" will be successfully cast into the int 123. The string -123 can be converted into an int using lexicalCast but it might not be converted correctly into an unsigned value.
Lexical casting in SeqAn uses the standard `<sstring>` library. The exact implementation of casting is library-dependent.

**A Full Example**

The following example program demonstrates the usage of both interfaces.

```cpp
#include <iostream>
#include <seqan/stream.h>

int main(int, char const **) {
    using namespace seqan;

    int resultI = 0;
    double resultD = 0;
    bool b = false;

    resultI = lexicalCast<int>("123");
    std::cerr << "lexicalCast<int>("123") --> " << resultI << std::endl;

    resultI = lexicalCast<int>("123XX");
    std::cerr << "lexicalCast<int>("123XX") --> " << resultI << std::endl;

    b = lexicalCast2<int>(resultI, "-123");
    std::cerr << "lexicalCast2<int>("-123") --> (" << b << ", " << resultI << ")" << std::endl;

    b = lexicalCast2<double>(resultD, "-123");
    std::cerr << "lexicalCast2<double>("-123") --> (" << b << ", " << resultD << ")" << std::endl;

    return 0;
}
```

This is the program’s output.

- `lexicalCast<int>("123")` --> 123
- `lexicalCast<int>("123XX")` --> 123
- `lexicalCast2<int>("-123")` --> (1, -123)
- `lexicalCast2<double>("-123")` --> (1, -123)

**Assignment 1  Using Lexical Casting**

**Type** Application

**Objective** Based on the example above, create a small program that takes one argument. This argument is then converted into an `int`, `unsigned`, and `double`. The program should display the results of `lexicalCast` and `lexicalCast2` as in the example above.

**Hints** The following shows an example session:

```
# tutorial_lexical_casting_solution1 10.3
lexicalCast<int>(10.3) == 10
lexicalCast<unsigned>(10.3) == 10
lexicalCast<double>(10.3) == 10
lexicalCast2<int>(10.3) == (1, 10)
lexicalCast2<unsigned>(10.3) == (1, 10)
```
lexicalCast2<double>({10.3}) == (1, 10.3)
# tutorial_lexical_casting_solution1 10
lexicalCast<int>({10}) == 10
lexicalCast<unsigned>({10}) == 10
lexicalCast<double>({10}) == 10
lexicalCast2<int>({10}) == (1, 10)
lexicalCast2<unsigned>({10}) == (1, 10)
lexicalCast2<double>({10}) == (1, 10)
# tutorial_lexical_casting_solution1 TEXT
lexicalCast<int>({TEXT}) == 0
lexicalCast<unsigned>({TEXT}) == 0
lexicalCast<double>({TEXT}) == 0
lexicalCast2<int>({TEXT}) == (0, 0)
lexicalCast2<unsigned>({TEXT}) == (0, 0)
lexicalCast2<double>({TEXT}) == (0, 0)

Solution

#include <iostream>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
    {
        std::cerr << "USAGE: tutorial_solution1 VALUE\n"
                  << " " << argv[1] << "\n"
        return 1;
    }

    // Lexical casting with lexicalCast().
    { int i = 0;
      unsigned u = 0;
      double d = 0;
      i = seqan::lexicalCast<int>({argv[1]});
      u = seqan::lexicalCast<unsigned>({argv[1]});
      d = seqan::lexicalCast<double>({argv[1]});
      std::cout << "lexicalCast<int>({argv[1]})) == " << i << "\n"
                 << "lexicalCast<unsigned>({argv[1]})) == " << u << "\n"
                 << "lexicalCast<double>({argv[1]})) == " << d << "\n"
    }

    // Lexical casting with lexicalCast2().
    { int i = 0;
      unsigned u = 0;
      double d = 0;
      bool bi = seqan::lexicalCast2<int>(i, {argv[1]});
      bool bu = seqan::lexicalCast2<unsigned>(u, {argv[1]});
      bool bd = seqan::lexicalCast2<double>(d, {argv[1]});
      std::cout << "lexicalCast2<int>({argv[1]})) == " << bi << "\n"
                 << "lexicalCast2<unsigned>({argv[1]})) == " << bu << "\n"
                 << "lexicalCast2<double>({argv[1]})) == " << bd << "\n"
    }

    return 0;
}
2.1.33 Parsing

Learning Objective  This tutorial will explain how to use the class RecordReader for the parsing of text file formats. You will see in-detail examples for parsing TSV-based formats such as GFF and BLAST tabular output and also for parsing the recursive Newick tree format.

Difficulty  Advanced

Duration  40 min

Prerequisites  I/O Overview, Lexical Casting

In this tutorial, you will learn how to use the RecordReader functions to easily create parsers for structured text file formats. We will first give a quick example for parsing a simple TSV format. Then, single-pass parsing will be explained (which is the most important variant) and the interface of the RecordReader class and the read*() and skip*() functions will be described. This is followed by extensive examples on parsing the GFF and BLAST tabular format and an example on how to parse the non-linear Newick format for phylogenetic trees. The tutorial closes with an explanation of how to write double-pass I/O code and in which situations it is useful to do so.

A First Example

Let us start off with a quick first example. The following program reads a two-column TSV file from the standard input. The first column contains keys, the second one contains values. The program prints the data as ${key} -> ${value} to stdout.

```cpp
#include <iostream>

#include <seqan/sequence.h>
#include <seqan/stream.h>
```

```cpp
2.1. Tutorial 317
```
As you can see, using the `RecordReader` is straightforward. First, we construct the `RecordReader` to wrap `std::cin` as also described in the `I/O Overview` tutorial.

Each iteration of the loop loads one record/line from standard input and writes out the record. We use `atEnd()` to check whether we are at the end of the file and loop. The function `readUntilChar()` reads the characters from the underlying file into a buffer `key` until a given character occurs, here the character is `'	'`. The reader will not copy the tabulator into `key` and stop on the character. The function `goNext()` can be used to go to the next character in the current file. The call to the function `readLine()` copies the data into `value` until the end of line, skipping the end-of-line marker (`'
'` or `'
'`) and does not copy the end-of-line marker to the `value`. Finally, we print the key/value pair to stdout.

### Assignment 1

Reading CSV instead of TSV.

**Type** Review

**Objective** Modify the example above to use a comma (',') instead of a tab character for separating columns.

**Hint** Yes, it is very easy.
Solution

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

// Read "<key>	<value>" map from stdin. Write out as "<key> -> <value>".

int main()
{
    // Define string and record reader types. We will read from std::cin which
    // is of type std::istream. We use a single-pass record reader.
    typedef seqan::RecordReader<std::istream, seqan::SinglePass<> > TRecordReader;

    int res = 0; // Used to store I/O results.
    // Create RecordReader reading from standard input.
    TRecordReader reader(std::cin);

    // Read the file line by line.
    while (!atEnd(reader))
    {
        // Read first column: The key.
        seqan::CharString key;
        res = readUntilChar(key, reader, ',');
        if (res != 0)
            return 1;
        goNext(reader); // Skip comma.

        // Read second column: The value.
        seqan::CharString value;
        res = readLine(value, reader); // EOL will not be stored in value.
        if (res != 0)
            return 1;

        // Print ${key} -> ${value}.
        std::cout << key << " -> " << value << std::endl;
    }

    return 0;
}
```

The Single-Pass RecordReader Class Interface

Single-pass record readers can simply be seen and used as an abstraction of streams. Read the file character-wise, from beginning to the end.

The low-level API for the single-pass reader is as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atEnd</td>
<td>Return true if the reader is at the end of the file, false otherwise.</td>
</tr>
<tr>
<td>goNext</td>
<td>Advance reader in file, return true if at end of file, false otherwise.</td>
</tr>
<tr>
<td>value</td>
<td>Return the character the reader points to at the moment.</td>
</tr>
<tr>
<td>resultCode</td>
<td>Return int with I/O status. 0 for no error, non-0 value for error when reading.</td>
</tr>
</tbody>
</table>

The following program shows another example of single-pass I/O. We read a text file line-by-line and append the
results to a String of CharStrings.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1;
    std::fstream stream(argv[1], std::ios::binary | std::ios::in);
    if (!stream.good())
        return 1;

    seqan::RecordReader<std::fstream, seqan::SinglePass<> > reader(stream);
    seqan::StringSet<seqan::CharString> result;

    while (!atEnd(reader))
    {
        resize(result, length(result) + 1);
        int res = readLine(back(result), reader);
        if (res != 0)
            return 1;
    }

    return 0;
}
```

Character Classes and the read* and skip* Functions

Character Classes And is*

In SeqAn, the same character classes are used as in the POSIX standard. See this list of character classes for a comprehensive list and description.

For example:

```cpp
printf("isdigit('a') == %d\n", isdigit('a')); // => "isdigit('a') == 0"
printf("isdigit('0') == %d\n", isdigit('0')); // => "isdigit('0') == 1"
printf("isblank(' ') == %d\n", isdigit(' ')); // => "isdigit(' ') == 0"
```

The read* And skip* Functions

The parsing functionality in SeqAn built on top of the StreamConcept concept and RecordReader class is optimized for reading bioinformatics text file formats.

These formats mostly consist of fairly flat data files, i.e. a sequence of records, each having very few levels of subrecords. A typical example are FASTQ files where one record consists of adjacent lines, containing the identifier, sequence, and qualities. Another example are TSV (tab-separated-values) files where each record spans a line and there possibly is a header. SAM is an example for a TSV file with a header at the top of the file.

The main challenge in reading bioinformatics files is their size. When parsing a word processor document file, a HTML document, or a computer program, the input file is typically not larger than some MB. In bioinformatics, files having multiple GB are not uncommon, e.g. NGS data or the sequence of the human genome.
Thus, in SeqAn, the files are parsed “on the fly” as they are read. Using compiler nomenclature, bioinformatics parsers often only have to be tokenizers. Making writing such simple parsers easy is the main aim of the read* and skip* functions in SeqAn. NB: By using Char Array Streams, you can also use the parsing infrastructure on in-memory data.

For each considered class of characters, there often is a read and a skip function. There are two big types of classes: White-listing/inclusion (read*X*) of certain characters and black-listing/exclusion (readUntil*X*) of certain characters. The inclusion functions stop after the last read/included character, the exclusion functions stop on the first excluded/not read character.

Most functions have the following interface. Note that all functions only append to the buffer argument, so you have to call clear yourself. This facilitates optimized reading into Concat Direct StringSets.

```c
int readUntilXXX (TBuffer & buffer, RecordReader<TStream, TPass> & reader);
int readXXX (TBuffer & buffer, RecordReader<TStream, TPass> & reader);
int skipUntilXXX (RecordReader<TStream, TPass> & reader);
int skipXXX (RecordReader<TStream, TPass> & reader);
```

**Tip: I/O Return Values and EOF_BEFORE_SUCCESS**

The read*() and skip*() functions return an int value. Consistent with C return codes, the return value is == 0 in case that the reading/skipping was successful and != 0 if reading/skipping was not successful.

The cases of unsuccessful reading/skipping include real errors (e.g. hardware problems) but also that the reader is at the end of the file. In this case seqan::EOF_BEFORE_SUCCESS is returned. This behaviour is required for file format guessing where a return value of seqan::EOF_BEFORE_SUCCESS is interpreted as success.

There are three cases in how code can handle the value seqan::EOF_BEFORE_SUCCESS: (1) interpret it as an error, (2) return seqan::EOF_BEFORE_SUCCESS itself, or (3) interpret it as “success”.

Here are some examples:

```
(1) Interpret as Error

// TRecordReader reader created above.
seqan::CharString buffer;
while (atEnd(reader))
{
    if (readLine(buffer, read) != 0)
        return 1; // handle as error
}
```

```
(2) Interpret as "seqan::EOF_BEFORE_SUCCESS"

Returning this code gives the caller the opportunity to handle end-of-file different from any other error. For example, a file format guesser can try to parse the first thousand bytes of a file and see whether they parse as valid. When EOF_BEFORE_SUCCESS is returned, it would count this as an access. Any other non-0 return code would be an error.

// TRecordReader reader created above.
seqan::CharString buffer;
int res = 0;
while (atEnd(reader))
{
    if ((res = readLine(buffer, read)) != 0)
        return res; // handle as error or EOF_BEFORE_SUCCESS
}
(3) Interpret as Success

In some cases, EOF is a valid event. For example, if you have a line-based file format such as TSV, the last line could end with an EOF instead of a line break.

```cpp
// TRecordReader reader created above.
seqan::CharString buffer;
int res = 0;
while (atEnd(reader))
{
    if ((res = readLine(buffer, read)) != 0 &&
        res != seqan::EOF_BEFORE_SUCCESS)
        return res; // line not reached in case of EOF
}
```

The following functions are available:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readDigits</td>
<td>Read digit characters.</td>
</tr>
<tr>
<td>readDna5IgnoringWhitespaces</td>
<td>Read DNA 5 characters, ignore whitespace.</td>
</tr>
<tr>
<td>readLetters</td>
<td>Read letter characters.</td>
</tr>
<tr>
<td>readLine</td>
<td>Read whole line, line break is not written into buffer.</td>
</tr>
<tr>
<td>readLineStripTrailingBlanks</td>
<td>Read whole line, trailing blanks are not written into buffer.</td>
</tr>
<tr>
<td>readNCChars</td>
<td>Read a fixed number of characters.</td>
</tr>
<tr>
<td>readNCCharsIgnoringWhitespace</td>
<td>Read a fixed number of characters, whitespace is not written into the buffer.</td>
</tr>
<tr>
<td>readUntilBlank</td>
<td>Read until a blank character occurs.</td>
</tr>
<tr>
<td>readUntilChar</td>
<td>Read until the given character occurs.</td>
</tr>
<tr>
<td>readUntilWhitespace</td>
<td>Read until a whitespace character occurs.</td>
</tr>
<tr>
<td>skipBlanks</td>
<td>Skip blank characters.</td>
</tr>
<tr>
<td>skipChar</td>
<td>Skip one given character.</td>
</tr>
<tr>
<td>skipLine</td>
<td>Skip from the current position to the end of the line.</td>
</tr>
<tr>
<td>skipNCChars</td>
<td>Skip a fixed number of characters.</td>
</tr>
<tr>
<td>skipNCCharsIgnoringWhitespace</td>
<td>Skip a fixed number of characters, ignore whitespace.</td>
</tr>
<tr>
<td>skipUntilBlank</td>
<td>Skip until a blank character occurs.</td>
</tr>
<tr>
<td>skipUntilChar</td>
<td>Skip until a certain character occurs.</td>
</tr>
<tr>
<td>skipUntilGraph</td>
<td>Skip until a graph character occurs.</td>
</tr>
<tr>
<td>skipUntilLineBeginsWithChar</td>
<td>Skip until a line begins with a certain character.</td>
</tr>
<tr>
<td>skipUntilLineBeginsWithOneCharOfStr</td>
<td>Skip until a line begins with one character of a given string/list.</td>
</tr>
<tr>
<td>skipUntilLineBeginsWithStr</td>
<td>Skip until a line begins with a certain string.</td>
</tr>
<tr>
<td>skipUntilString</td>
<td>Skip until a certain string is found.</td>
</tr>
<tr>
<td>skipUntilWhitespace</td>
<td>Skip until a whitespace character is found.</td>
</tr>
<tr>
<td>skipWhitespaces</td>
<td>Skip until a non-whitespace character is found.</td>
</tr>
</tbody>
</table>

In the following example, we read the first two fields of a TSV file from stdin and dump them to stdout.

```cpp
seqan::RecordReader<std::istream, seqan::SinglePass<> > reader(std::cin);
seqan::CharString buffer;

while (atEnd(reader))
{
    clear(buffer);
    int res = readUntilChar(buffer, reader, '\t');
    if (res != 0)
        return res;
    std::cout << buffer;
}
if (goNext(reader))
    return seqan::EOF_BEFORE_SUCCESS;

clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0)
    return res;
std::cout << buffer << std::endl;

res = skipLine(reader);
if (res != 0 && res != seqan::EOF_BEFORE_SUCCESS)
    return 1;
}

Writing Your Own read* and skip* Functions

Writing your own reading/skipping function is easy. As an example, we write functions for reading and skipping the characters from the set \{x, y, z\}. The functions follow the same pattern and use the functions _readHelper() and _skipHelper().

These functions read/skip characters as long as a specific overload of the predicate function _charCompare() (in the seqan namespace) returns true. The _charCompare() function gets two parameters: The character to test and a tag for selecting the specific _charCompare() overload. The character to test is of type int. The tag is defined by you as a developer and the tag given to _charCompare() is the same as given to _readHelper() and _skipHelper().

For good examples, you can look at the file core/include/seqan/stream/tokenize.h to see how the rest of the read* and skip* functions from above are implemented.

struct Xyz;
typedef seqan::Tag<Xyz> Xyz;

inline int
_charCompare(int const c, Xyz const & /* tag*/)
{
    return c == 'x' || c == 'y' || c == 'z';
}

template <typename TStream, typename TPass, typename TBuffer>
inline int
readXyz(TBuffer & buffer, seqan::RecordReader<TStream, TPass> & reader)
{
    return seqan::_readHelper(buffer, reader, Xyz(), false);
}

template <typename TBuffer, typename TStream, typename TPass>
inline int
readUntilXyz(TBuffer & buffer, seqan::RecordReader<TStream, TPass> & reader)
{
    return seqan::_readHelper(buffer, reader, Xyz(), true);
}

template <typename TStream, typename TPass>
inline int
skipXyz(seqan::RecordReader<TStream, TPass> & reader)
{
    return seqan::_skipHelper(reader, Xyz(), false);
Assignment 2  Writing `readHexNumber()`.

Type  Review

Objective  Write your own read and skip routines for hexadecimal numbers. Such numbers can only contain digits 0–9 and the characters a–f and A–F.

Solution  The following program reads from stdin as long as the input forms a valid hexadecimal number. Note that you can send an end-of-file character to your application by pressing Ctrl + d.

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

// The following few lines are the actual solution to the assignment.

struct HexNumChars_;
typedef seqan::Tag<HexNumChars_> HexNumChars;

inline int _charCompare(int const c, HexNumChars const & /* tag*/) {
    return isdigit(c) || (c >= 'a' && c <= 'f') || (c >= 'A' && c <= 'F');
}

template <typename TStream, typename TPass, typename TBuffer>
inline int readHexNumber(TBuffer & buffer, seqan::RecordReader<TStream, TPass> & reader) {
    return seqan::_readHelper(buffer, reader, HexNumChars(), false);
}

// This main routine is only some driver code that reads from stdin.
int main() {
    seqan::RecordReader<std::istream, seqan::SinglePass<> > reader(std::cin);
    while (!atEnd(reader)) {
        seqan::CharString buffer;
        int res = readHexNumber(buffer, reader);
        if (res != 0 && res != seqan::EOF_BEFORE_SUCCESS) {
            std::cerr << "ERROR: Could not read from standard input.\n";
            return 1;
        }
    }
}
```
An example session. The Ctrl + d is shown as ^D.

```
$ tutorial_parsing_solution2
foo
10
20
2a^D
RECOGNIZED f
RECOGNIZED 10
RECOGNIZED 20
RECOGNIZED 2a
```

Assignment 3 Writing `readPunctuation()`.

**Type Review**

**Objective** Modify the example above to read a sequence of punctuation characters in a function called `readPunctuation()`.

**Hint** You can use the function `ispunct()`.

**Solution**

```cpp
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

// The following few lines are the actual solution to the assignment.

struct Punct_;
typedef seqan::Tag<Punct_> Punct;

inline int _charCompare(int const c, Punct const & /* tag*/) {
    return ispunct(c);
}

template <typename TStream, typename TPass, typename TBuffer>
inline int readPunctuation(TBuffer & buffer, seqan::RecordReader<TStream, TPass> & reader) {
    return seqan::_readHelper(buffer, reader, Punct(), false);
}

// This main routine is only some driver code that reads from stdin.

int main()
```
```
{  
  seqan::RecordReader<std::istream, seqan::SinglePass<> > reader(std::cin);
  
  while (!atEnd(reader))
  {  
    seqan::CharString buffer;
    int res = readPunctuation(buffer, reader);
    if (res != 0 && res != seqan::EOF_BEFORE_SUCCESS)
    {  
      std::cerr << "ERROR: Could not read from standard input.\n";
      return 1;
    }

    // Print hexadecimal number back to the user.
    std::cout << "RECOGNIZED " << buffer << '\n';

    // Skip all trailing input.
    skipLine(reader);
  }

  return 0;
}
```

An example session. The Ctrl + d is shown as ^D.

...  
asdf  
!!@@%%^  
RECOGNIZED ...  
RECOGNIZED !!  
RECOGNIZED !!@@%^

### File Parsing Practice

This section will walk you through a parser for GFF, tabular BLAST output, and the Newick tree format.

#### Common Patterns

In order to support a new file format, you usually (1) introduce a struct type for storing records, (2) create tags for the file type and the records, and (3) provide overloads of the functions nextIs() and readRecord(). For example, for the GFF format, we

- create a struct GffRecord (1)
- create the tag Gff (2)
- create overloads of nextIs and readRecord for Gff (3).

#### A Simple GFF2 Parser

We will implement a simple parser for the GFF file format version 2. For the sake of simplicity, will not implement parsing of ## and will read the whole attributes field as one and not subdivide it further. Here, GFF2 files are TSV files with the following fields.

<seqname> <source> <feature> <start> <end> <score> <strand> <frame> [attributes] [comments]
The following example shows a GFF2 parser. First, include the necessary headers.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>
using namespace seqan;
```

Then, define Gff2 tag and record struct.

```cpp
struct Gff2;
typedef Tag<Gff2> Gff2;
```

```cpp
struct Gff2Record
{
    CharString seqName;
    CharString source;
    CharString feature;
    unsigned start;
    unsigned end;
    bool hasScore;
    double score;
    char strand;
    unsigned frame;
    CharString attributes;
    CharString comments;

    Gff2Record() : start(0), end(0), hasScore(false), score(0), strand('.'), frame(0) {}
};
```

```cpp
void clear(Gff2Record & record)
{
    clear(record.seqName);
    clear(record.source);
    clear(record.feature);
    clear(record.attributes);
    clear(record.comments);
}
```

We then implement a parser function for GFF records. Note that most of the code is error handling.

```cpp
template <typename TStream, typename TPass>
int readRecord(Gff2Record & record, RecordReader<TStream, TPass> & reader, Gff2 const & /*tag*/)
{
    clear(record);
    CharString buffer;
    char c = '\0';
    int res = 0;

    // GFF2 records look like this:
    // // <seqname> <source> <feature> <start> <end> <score> <strand> <frame> [attributes] [comments]
    // <seqname>

    res = readUntilChar(record.seqName, reader, '\t');
    if (res != 0)
```
return res;
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <source>
res = readUntilChar(record.source, reader, '	');
if (res != 0)
    return res;
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <feature>
res = readUntilChar(record.feature, reader, '	');
if (res != 0)
    return res;
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <start>
clear(buffer);
res = readUntilChar(buffer, reader, '	');
if (res != 0)
    return res;
if (!lexicalCast2<unsigned>(record.end, buffer))
    return 1; // Could not cast!
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <end>
clear(buffer);
res = readUntilChar(buffer, reader, '	');
if (res != 0)
    return res;
if (!lexicalCast2<unsigned>(record.end, buffer))
    return 1; // Could not cast!
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <score>
clear(buffer);
res = readUntilChar(buffer, reader, '	');
if (res != 0)
    return res;
record.hasScore = (buffer != '.');
if (record.hasScore && !lexicalCast2<double>(record.score, buffer))
    return 1; // Could not cast!
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <strand>
clear(buffer);
res = readUntilChar(buffer, reader, '	');
if (res != 0)
    return res;
if (length(buffer) != 1u)
    return 1; // More than one char or none.
c = front(buffer);
if (c != '.' && c != '+' && c != '-')
return 1; // Invalid strand.
record.strand = c;
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <frame>
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != EOF)
    return res;
if (length(buffer) != 1u)
    return 1; // More than one char or none.
c = front(buffer);
if (c != '.' && c != '0' && c != '1' && c != '2')
    return 1; // Invalid frame.
record.frame = c;
if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// <attributes>
res = readUntilTabOrLineBreak(record.attributes, reader);
if (res != EOF && res != EOF_BEFORE_SUCCESS)
    return res;
if (atEnd(reader))
    return 0;
if (value(reader) == '\t')
{
    if (goNext(reader))
        return EOF_BEFORE_SUCCESS;
}

// <comment>
res = readLine(record.seqName, reader);
if (res != EOF && res != EOF_BEFORE_SUCCESS)
    return res;
return 0;
}

On top of the record-reading routine, we implement reading of whole documents. This is quite simple.

```cpp
template <typename TGff2Records, typename TStream, typename TPass>
int read2(TGff2Records & records, RecordReader<TStream, TPass> & reader, Gff2 const & /*tag*/)
{
    Gff2Record record;
    while (!atEnd(reader))
    {
        clear(record);
        int res = readRecord(record, reader, Gff2());
        if (res != 0)
            return res;
        appendValue(records, record);
    }
    return 0;
}
```

Finally, some driver code to open a file and call the parser routine. In the end, we dump some of the information we just read.
```cpp
int main(int argc, char const ** argv)
{
    // Handle command line arguments, open files.
    if (argc != 2)
        return 1;
    std::fstream stream(argv[1], std::ios::binary | std::ios::in);
    if (!stream.good())
        return 1;

    // Read file.
    RecordReader<std::fstream, SinglePass<> > reader(stream);
    String<Gff2Record> gffRecords;
    int res = read2(gffRecords, reader, Gff2());
    if (res != 0)
        return res;

    // Write out some of the data to stdout.
    for (unsigned i = 0; i < length(gffRecords); ++i)
        std::cout << gffRecords[i].seqName << "\t" << gffRecords[i].strand << "\t" << gffRecords[i].start << "\t" << gffRecords[i].end << std::endl;

    return 0;
}
```

Let’s look at an example run of the program.

```
# cat extras/demos/tutorial/parsing /gff2_example.txt
IV curated mRNA 5506800 5508917 . + . Transcript B0273.1; Note "Zn-Finger"
IV curated 5'UTR 5506800 5508999 . + . Transcript B0273.1
IV curated exon 5506900 5506996 . + . Transcript B0273.1
IV curated exon 5506026 5506382 . + . Transcript B0273.1
IV curated exon 5506558 5506660 . + . Transcript B0273.1
IV curated exon 5506738 5506852 . + . Transcript B0273.1
IV curated 3'UTR 5506852 5508917 . + . Transcript B0273.1
# ./extras/demos/tutorial/parsing/tutorial_parse_gff2 extras/demos/tutorial/parsing/gff2_example.txt
IV + 0 5508917
IV + 0 5508999
IV + 0 5506996
IV + 0 5506382
IV + 0 5506660
IV + 0 5506852
IV + 0 5508917
```

**Newick Tree Parsing (Recursion Example)**

The newick tree format is used for encoding phylogenetic trees (see Newick Tree Format Standard for a formal specification). We will write a parser that reads Newick forest files (without allowing for comments).

Here is an example for the Newick format:

```latex
(((One:0.2,Two:0.3):0.3,(Three:0.5,Four:0.3):0.2):0.3,Five:0.7):0.0;
```

A file with this content encodes the following tree:

```
    +-- One
    |    +-- Two
    +-- Three
```
And here is the grammar of the Newick format in EBNF.

```
forest      = tree+;
tree        = node, ";";
node        = children, label?, distance?
             | children?, label, distance?;
children    = "(" node (",", node)* ")";
label       = quoted-list
             | unquoted-list;
distance    = ":", number;
quoted-list = "'", (qchar escaped-quote)*, "'";
escaped-quote = "''";
unquoted-list = uqchar;
```

The following demo shows the parsers, code to dump the tree from the internal data structures and a small driver program for the routines.

First, the necessary includes.

```cpp
#include <iostream>
#include <fstream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/graph_types.h>
using namespace seqan;
```

Then, we define a Newick tag and a struct for branch labels.

```cpp
struct Newick_;
typedef Tag<Newick_> Newick;
struct NewickBranchLabel
{
    bool isDistanceSet;
    double distance;

    NewickBranchLabel() : isDistanceSet(false), distance(0)
    {};
};
```

In a next step, we write a `readFloatLiteral()` helper function that is reusable.

```cpp
template <typename TCharSeq, typename TStream, typename TPass>
int _readExponentialPart(TCharSeq & buffer,
                         RecordReader<TStream, TPass> & reader)
{
    // Check preconditions.
    SEQAN_ASSERT_NOT(atEnd(reader));
    SEQAN_ASSERT(value(reader) == 'e' || value(reader) == 'E');

    // Read 'e' or 'E';
```
appendValue(buffer, value(reader));

if (goNext(reader))
    return EOF_BEFORE_SUCCESS;

// Possibly read '+' or '-'.
if (value(reader) == '+' || value(reader) == '-')
{
    appendValue(buffer, value(reader));
    if (goNext(reader))
        return EOF_BEFORE_SUCCESS;
}

// Possibly read '+' or '-'.
if (!isdigit(value(reader)))
    return 1; // Should have been a digit!

return readDigits(buffer, reader);
}

template <typename TCharSeq, typename TStream, typename TPass>
int readFloadLiteral(TCharSeq & buffer, RecordReader<TStream, TPass> & reader)
{
    if (atEnd(reader))
        return EOF_BEFORE_SUCCESS; // Empty field.

    // The EBNF for floating point integers is as follows:
    //
    // exponent-indicator = e | E
    // exponent-part = exponent-indicator [+]digits
    // floating-point-literal = digits exponent-part
    // | digits. [digits] exponent-part
    // | .digits [exponent-part]

    // Read one leading sign if it is there.
    if (value(reader) == '-' || value(reader) == '+')
    {
        appendValue(buffer, value(reader));
        if (goNext(reader))
            return EOF_BEFORE_SUCCESS; // At end after leading sign.
    }

    // Digits or dot?
    if (value(reader) == '.')
    {
        // Dot
        appendValue(buffer, '.');
        if (goNext(reader))
            return EOF BEFORE SUCCESS;

        if (!isdigit(value(reader)))
            return 1; // Invalid format, >= 1 digit have to follow.

        int res = readDigits(buffer, reader);
        if (res != 0)
            return res; // Error reading digits.

        // Optionally read exponential part.
        if (atEnd(reader))
            return 0;

        if (value(reader) == 'e' || value(reader) == 'E')
            return _readExponentialPart(buffer, reader);
    }
    else
// Digits
if (!isdigit(value(reader)))
    return 1; // >= 1 digit required!
int res = readDigits(buffer, reader);
if (res != 0)
    return res; // Error reading digits.
if (atEnd(reader)) // Stop if no more data.
    return 0;
if (value(reader) == '.')
{
    appendValue(buffer, '.');
    if (goNext(reader))
        return 0; // End of field.
    if (isdigit(value(reader)))
    {
        res = readDigits(buffer, reader);
        if (res != 0)
            return res; // Error reading digits.
    }
    // Optionally read exponential part.
    if (atEnd(reader))
        return 0;
    if (value(reader) == 'e' || value(reader) == 'E')
    {
        return _readExponentialPart(buffer, reader);
    }
}
else if (value(reader) == 'e' || value(reader) == 'E')
{
    return _readExponentialPart(buffer, reader);
}
return 0;
}

The code for reading a Newick forest is recursive and a bit lengthy but not too complex. We load such forests into strings of Tree objects. Additionally, we have a vertex map for the branch distances and the vertex labels for each tree.

```cpp
template<
typename TTree,
typename TRecordReader,
typename TVertexDescriptor>
int _readNewickTree(TTree & tree,
        String<CharString> & vertexLabels,
        String<NewickBranchLabel> & branchLabels,
        TRecordReader & reader,
        TVertexDescriptor v)
{
    if (atEnd(reader))
        return EOF_BEFORE_SUCCESS;
    CharString buffer;
    int res = 0;

    // Define SKIP_WHITESPACE
    #define SKIP_WHITESPACE
    do
    { 
        int res = skipWhitespaces(reader);
        if (res != 0 && res != EOF_BEFORE_SUCCESS)
            return res;
    } while(false)
```
if (value(reader) == '(') // CHILDREN
{
    if (goNext(reader))
        return EOF_BFOR_E_SUCCESS;
    // children
    bool first = true;
    while (true)
    {
        SKIP_WHITESPACE;

        // Skip leading comma.
        if (!first)
        {
            res = skipChar(reader, ',');
            if (res != 0)
                return res;
        }
        first = false;
        SKIP_WHITESPACE;

        // Read child.
        TVertexDescriptor x = addChild(tree, v);
        resizeVertexMap(tree, vertexLabels);
        resizeVertexMap(tree, branchLabels);
        res = _readNewickTree(tree, vertexLabels, branchLabels, reader, x);
        if (res != 0)
            return res;
        SKIP_WHITESPACE;

        // Exit loop.
        if (value(reader) == ')')
            break;
    }
    res = skipChar(reader, ')');
    if (res != 0)
        return res; // Could not close child list.
    SKIP_WHITESPACE;
}
if (value(reader) != ':') // LABEL
{
    SKIP_WHITE
    clear(buffer);
    if (value(reader) == '\'
    {
        // Read quoted label.
        if (goNext(reader))
            return EOF_BFOR_E_SUCCESS;
        while (!atEnd(reader))
        {
            char c = value(reader);
            if (c == '\') // Possibly break, if not ""'.
                {
                    if (goNext(reader))
                        break;
                    if (value(reader) != '\')
                        break;
        

334 Chapter 2. API Documentation
appendValue(buffer, value(reader));
if (goNext(reader))
    return 1;
}

else {
    // Read unquoted label.
    while (!atEnd(reader)) {
        char c = value(reader);
        if (isblank(c) || c == '(' || c == ')' || c == '[' ||
            c == ']' || c == '\' || c == '.' || c == ';' ||
            c == ',' || c == ':')
            break;
        appendValue(buffer, value(reader));
        if (goNext(reader))
            return 1;
    }

    assignProperty(vertexLabels, v, buffer);
    SKIP_WHITESPACE;
}

if (value(reader) == ':') // DISTANCE
    
    skipChar(reader, ':');
    SKIP_WHITESPACE;
    clear(buffer);
    res = readFloadLiteral(buffer, reader);
    if (res != 0)
        return res; // Invalid distance.
    property(branchLabels, v).isDistanceSet = true;
    property(branchLabels, v).distance = lexicalCast<double>(buffer);
    SKIP_WHITESPACE;
    
    return 0;
}

// Read forest.
while (!atEnd(reader)) {
    // Allocate graph and maps.
    resize(forest, length(forest) + 1);
    resize(vertexLabels, length(vertexLabels) + 1);
resize(branchLabels, length(branchLabels) + 1);
// Allocate root.
createRoot(back(forest));
TVertexDescriptor v = root(back(forest));
resizeVertexMap(back(forest), back(vertexLabels));
resizeVertexMap(back(forest), back(branchLabels));
// Read tree.
res = _readNewickTree(back(forest), back(vertexLabels), back(branchLabels), reader, v);
if (res != 0)
    return res;
// Skip trailing semicolon, must be there.
res = skipChar(reader, ’;’);
if (res != 0)
    return res;
SKIP_WHITESPACE;
}

#undef SKIP_WHITESPACE

    return 0;
}

The code for dumping a Newick forest is also quite simple, if lengthy because of error checks.

    template <typename TStream, typename TTree, typename TVertexDescriptor, typename TVertexLabels,
              typename TBranchLabels>
int _writeNewickRecurse(TStream & stream, TTree & tree, TVertexDescriptor v,
    TVertexLabels & vertexLabels, TBranchLabels & branchLabels)
{
    if (numChildren(tree, v) > 0u)
    {
        int res = streamPut(stream, ’{’);
        if (res != 0)
            return res;

typename Iterator<TTree, OutEdgeIterator>::Type it(tree, v);
bool first = true;
for (; !atEnd(it); goNext(it))
{
    if (!first)
    {
        res = streamPut(stream, ’,’);
        if (res != 0)
            return res;
    }
    first = false;
    res = _writeNewickRecurse(stream, tree, targetVertex(it), vertexLabels, branchLabels);
    if (res != 0)
        return res;
}

res = streamPut(stream, ’}’);
if (res != 0)
    return res;
}
// Write label if any, quoted if required.
if (length(property(vertexLabels, v)) > 0u)
{
    bool needsQuoting = false;

CharString const & label = property(vertexLabels, v);

// Write branch length if any is given.
if (property(branchLabels, v).isDistanceSet)
{
    int res = streamPut(stream, ':');
    if (res != 0)
        return res;
    res = streamPut(stream, property(branchLabels, v).distance);
    if (res != 0)
        return res;
}
else
{
    int res = streamPut(stream, label);
    if (res != 0)
        return res;
}

// Write branch length if any is given.
if (property(branchLabels, v).isDistanceSet)
{
    int res = streamPut(stream, ':');
    if (res != 0)
        return res;
    res = streamPut(stream, property(branchLabels, v).distance);
    if (res != 0)
        return res;
}
else
{
    int res = streamPut(stream, label);
    if (res != 0)
        return res;
}
inline int write2(TStream & stream, 
    Graph<Tree<> > & tree, 
    String<CharString> & vertexLabels, 
    String<NewickBranchLabel> & branchLabels, 
    Newick const & /*tag*/)
{
    // Write <tree>;
    int res = _writeNewickRecurse(stream, tree, getRoot(tree), vertexLabels, branchLabels);
    if (res != 0)
        return res;
    return streamPut(stream, ';');
}

Finally, the main() routine.

int main(int argc, char const ** argv)
{
    // Handle arguments, open file.
    if (argc != 2)
    {
        std::cerr << "Incorrect argument count!" << std::endl;
        std::cerr << "USAGE: tutorial_parse_newick INPUT.txt" << std::endl;
        return 1;
    }
    std::fstream stream(argv[1], std::ios::binary | std::ios::in);
    if (!stream.good())
    {
        std::cerr << "Could not open file " << argv[1] << std::endl;
        return 1;
    }
    RecordReader<std::fstream, SinglePass<> > reader(stream);

    // Load forest.
    String<Graph<Tree<> > > forest;
    String<String<CharString> > vertexLabels;
    String<String<NewickBranchLabel> > branchLabels;
    int res = read2(forest, vertexLabels, branchLabels, reader, Newick());
    if (res != 0)
    {
        std::cerr << "Could not read Newick file!" << std::endl;
        return res;
    }

    // Dump forests.
    for (unsigned i = 0; i < length(forest); ++i)
    {
        res = write2(std::cout, forest[i], vertexLabels[i], branchLabels[i], Newick());
        std::cout << "\n";
        if (res != 0)
        {
            std::cerr << "Error writing to stdout!" << std::endl;
            return 1;
        }
    }
    return 0;
}

Let’s look at an example run. Note that the children in SeqAn trees do not have a specific order and the Newick format
does not introduce any normalized order. In the written result, the order of the children has changed.

```c
# cat extras/demos/tutorial/parsing/newick_example.txt
(a,('Darwin''s Bulldog (Huxley)',c):-1.92e19)'The ''Root''':5;
(a_node,
   'another node',
   bird:0.3134)higher_node:4.5,
c):1.03e10;
({<sub>},,(<sub>},));
# tutorial_parse_newick extras/demos/tutorial/parsing/newick_example.txt
(c,(bird:0.3134,'another node',a_node)higher_node:4.5):1.03e10;
({<sub>},,(<sub>},));
```

### Parsing Tabular BLAST

The program **BLASTN** can be given an `-outfmt` parameter that makes it generate tabular output. This output is quite easy to parse (much easier than the human-readable BLAST reports) and looks as follows:

```bash
# blastn -subject NC_001405.fasta -query NC_001460.fasta -outfmt 7 > blast_example.txt
# cat blast_example.txt
# BLASTN 2.2.25+
# Query: gi|9626621|ref|NC_001460.1| Human adenovirus A, complete genome
# Subject: gi|9626158|ref|NC_001405.1| Human adenovirus C, complete genome
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens, q. start, q. end, s. start, s. end, evalue, bit score
# 3 hits found
| gi|9626621|ref|NC_001460.1| gi|9626158|ref|NC_001405.1| 81.13 408 66 11 17730 18131 18827 19229 5e-87 316 |
| gi|9626621|ref|NC_001460.1| gi|9626158|ref|NC_001405.1| 81.63 98 12 6 383 476 433 528 9e-15 76.8 |
| gi|9626621|ref|NC_001460.1| gi|9626158|ref|NC_001405.1| 76.27 118 22 6 25147 25261 26644 26758 3e-09 58.4 |
# BLAST processed 1 queries
```

The following example program takes the name of such a blastn output, reads it into record data structures and then prints it out in a different format again. To do this, we will first implement a record-reading API that allows streaming through the file. Then, we build a batch-reading API that reads such a file into a sequence of records that are all kept in main memory.

The program starts with including the required headers.

```c
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/stream.h>

using namespace seqan;
```

Then, we define a record for the file format **BlastnTab** and tabs for the comment and alignment record types.

```c
struct BlastnTab_;
typedef Tag<BlastnTab_> BlastnTab;

struct BlastnTabComment_;
typedef Tag<BlastnTabComment_> BlastnTabComment;

struct BlastnTabAlignment_;
typedef Tag<BlastnTabAlignment_> BlastnTabAlignment;
```

Next, we define a record type. Note that this record type is very specialized to the `blastn -outfmt 7` format.

---

2.1. Tutorial 339
When writing I/O code for multiple format for similar data, you might want to consider writing one record type for all of them. See the (upcoming, TODO) SAM record I/O for the implementation of one record type for the SAM and then BAM format.

We also create a simple function to dump the record to a stream.

```cpp
struct BlastnTabAlignmentRecord
{
    CharString queryName;
    CharString subjectName;
    double identity;
    unsigned alignmentLength;
    unsigned mismatches;
    unsigned gapOpens;
    unsigned queryBegin;
    unsigned queryEnd;
    unsigned subjectBegin;
    unsigned subjectEnd;
    double eValue;
    double bitScore;

    BlastnTabAlignmentRecord() :
        identity(0), alignmentLength(0), mismatches(0), gapOpens(0),
        queryBegin(0), queryEnd(0), subjectBegin(0), subjectEnd(0),
        eValue(0), bitScore(0)
    {};
};

template <typename TStream>
int write(TStream & stream, BlastnTabAlignmentRecord & record)
{
    streamPut(stream, "query name: ");
    streamPut(stream, record.queryName);
    streamPut(stream, " subject name: ");
    streamPut(stream, record.subjectName);
    streamPut(stream, " identity: ");
    streamPut(stream, record.identity);
    streamPut(stream, " alignment length: ");
    streamPut(stream, record.alignmentLength);
    streamPut(stream, " mismatches: ");
    streamPut(stream, record.mismatches);
    streamPut(stream, " gap opens: ");
    streamPut(stream, record.gapOpens);
    streamPut(stream, " query begin: ");
    streamPut(stream, record.queryBegin);
    streamPut(stream, " query end: ");
    streamPut(stream, record.queryEnd);
    streamPut(stream, " subject begin: ");
    streamPut(stream, record.subjectBegin);
    streamPut(stream, " subject end: ");
    streamPut(stream, record.subjectEnd);
    streamPut(stream, " evalue: ");
    streamPut(stream, record.eValue);
    streamPut(stream, " bit score: ");
    streamPut(stream, record.bitScore);
    int res = streamPut(stream, "\n\n");
    return res;
}
```
void clear(BlastnTabAlignmentRecord & record)
{
    clear(record.queryName);
    clear(record.subjectName);
}

Then, we define nextIs functions for the BlastnTabComment and BlastnTabAlignment tags, and their represented record types.

```cpp
template <typename TStream, typename TSpec>
inline bool nextIs(RecordReader<TStream, SinglePass<TSpec> > & reader, BlastnTabComment const & /*tag*/)
{
    return !atEnd(reader) && value(reader) == '#';
}

template <typename TStream, typename TSpec>
inline bool nextIs(RecordReader<TStream, SinglePass<TSpec> > & reader, BlastnTabAlignment const & /*tag*/)
{
    return !atEnd(reader) && value(reader) != '#';
}
```

Then, we implement a record-reading API on top of the skip* and read* functions. Note that the error handling bloats up the number of required lines but is necessary.

```cpp
template <typename TCharSequence, typename TStream, typename TSpec>
inline int readRecord(TCharSequence & buffer, RecordReader<TStream, SinglePass<TSpec> > const & reader, BlastnTabComment const & /*tag*/)
{
    SEQAN_ASSERT(nextIs(reader, BlastnTabComment()));
    clear(buffer);
    return readLine(buffer, reader);
}

template <typename TStream, typename TSpec>
inline bool readRecord(BlastnTabAlignmentRecord & record, RecordReader<TStream, SinglePass<TSpec> > & reader, BlastnTabAlignment const & /*tag*/)
{
    SEQAN_ASSERT(nextIs(reader, BlastnTabAlignment()));
    int res = 0;
    CharString buffer;
    clear(record);

    // Read query name.
    res = readUntilChar(record.queryName, reader, '\t');
    if (res != 0)
        return res; //Or we can return true here.
    goNext(reader);

    // Read subject name.
    res = readUntilChar(record.subjectName, reader, '\t');
    if (res != 0)
        return res; //Or we can return true here.
    goNext(reader);

    // Read identity.
    clear(buffer);
    res = readUntilChar(buffer, reader, '\t');
}
if (res != 0) return res;
if (!lexicalCast2<double>(record.identity, buffer))
    return 1; // Could not cast identity to double.
goNext(reader);

// Read alignment length.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.alignmentLength, buffer))
    return 1; // Could not cast alignment length to unsigned.
goNext(reader);

// Read mismatches.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.mismatches, buffer))
    return 1; // Could not cast mismatches to unsigned.
goNext(reader);

// Read gap opens.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.gapOpens, buffer))
    return 1; // Could not cast gap opens to unsigned.
goNext(reader);

// Read query begin.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.queryBegin, buffer))
    return 1; // Could not cast query begin to unsigned.
goNext(reader);

// Read query end.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.queryEnd, buffer))
    return 1; // Could not cast query end to unsigned.
goNext(reader);

// Read subject begin.
clear(buffer);
res = readUntilChar(buffer, reader, '\t');
if (res != 0) return res;
if (!lexicalCast2<unsigned>(record.subjectBegin, buffer))
    return 1; // Could not cast subject begin to unsigned.
goNext(reader);

    // Read subject end.
    clear(buffer);
    res = readUntilChar(buffer, reader, '\t');
    if (res != 0)
        return res;
    if (!lexicalCast2<unsigned>(record.subjectEnd, buffer))
        return 1; // Could not cast subject end to unsigned.
    goNext(reader);

    // Read evalue.
    clear(buffer);
    res = readUntilChar(buffer, reader, '\t');
    if (res != 0)
        return res;
    if (!lexicalCast2<double>(record.eValue, buffer))
        return 1; // Could not cast evalue to double.
    goNext(reader);

    // Read bit score, up to end of the line.
    clear(buffer);
    res = readLine(buffer, reader);
    if (res != 0)
        return res;
    if (!lexicalCast2<double>(record.bitScore, buffer))
        return 1; // Could not cast bit score to double.
    return 0;
}

On top of the record-reading API, we implement a batch-reading function. This function turns out to be fairly simple.

```cpp
template <typename TBlastnTabRecords, typename TStream, typename TSpec>
int read(TBlastnTabRecords & records, RecordReader<TStream, SinglePass<TSpec>> & reader, BlastnTab const & /*tag*/) {
    BlastnTabAlignmentRecord record;
    while (!atEnd(reader))
    {
        if (nextIs(reader, BlastnTabComment()))
        {
            skipRecord(reader, BlastnTabComment());
            continue;
        }
        if (!nextIs(reader, BlastnTabAlignment()))
            return 1;
        if (readRecord(record, reader, BlastnTabAlignment()) != 0)
            return 1;
        appendValue(records, record);
    }
    return 0;
}
```

In the main() routine, we can then simply open a std::fstream, create a RecordReader. Then, use the batch-reading API to read the whole file into main memory and write it to stdout again.

```cpp
int main(int argc, char const ** argv) {
    // Process command line arguments, open file.
```
if (argc != 2)
{
    std::cerr << "Incorrect argument count!" << std::endl;
    std::cerr << "USAGE: tutorial_parse_blastn INPUT.txt" << std::endl;
    return 1;
}

std::fstream stream(argv[1], std::ios::binary | std::ios::in);
if (!stream.good())
{
    std::cerr << "Could not open file " << argv[1] << std::endl;
    return 1;
}

// Read file.
RecordReader<std::fstream, SinglePass<> > reader(stream);
String<BlastnTabAlignmentRecord> records;
int res = read(records, reader, BlastnTab());
if (res != 0)
{
    std::cerr << "Could not read BLASTN records." << std::endl;
    return 1;
}

// Write read records.
for (unsigned i = 0; i < length(records); ++i)
    write(std::cout, records[i]);

return 0;

The program’s output looks as follow:

```
# ./extras/demos/tutorial/parsing/tutorial_parse_blastn ../../extras/demos/tutorial/parsing/blast_example.txt
query name: gi|9626621|ref|NC_001460.1|
subject name: gi|9626158|ref|NC_001405.1|
identity: 81.13
alignment length: 408
mismatches: 66
gap opens: 11
query begin: 17730
query end: 18131
subject begin: 18827
subject end: 19229
evalue: 5e-87
bit score: 316

query name: gi|9626621|ref|NC_001460.1|
subject name: gi|9626158|ref|NC_001405.1|
identity: 81.63
alignment length: 98
mismatches: 12
gap opens: 6
query begin: 383
query end: 476
subject begin: 433
subject end: 528
evalue: 9e-15
bit score: 76.8
```
Double-Pass I/O Using the RecordReader

The Double-Pass RecordReader reader's API extends the function described above for the Single-Pass RecordReader. It provides the following additional global interface functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startFirstPass</td>
<td>Start first pass of reading.</td>
</tr>
<tr>
<td>startSecondPass</td>
<td>Second pass of reading.</td>
</tr>
</tbody>
</table>

It is used as follows: For each section of the file that is to be read in the next step (one or multiple records), you first call startFirstPass. This memoizes the current position in the file. Then, you use the same API as for the single-pass reader to read the file. When you are done with this section, you call startSecondPass. This will reset the position of the reader to the one where startFirstPass was called.

Here is an example for using double-pass I/O:

```cpp
#include <iostream>
#include <fstream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>
using namespace seqan;

int main(int argc, char const ** argv)
{
    if (argc != 2)
        return 1;
    std::fstream stream(argv[1], std::ios::binary | std::ios::in);
    if (!stream.good())
        return 1;

    RecordReader<std::fstream, DoublePass<> > reader(stream);
    String<CharString> result;
    CharString buffer;

    while (!atEnd(reader))
    {
        startFirstPass(reader);
        clear(buffer);
        int res = readLine(buffer, reader);
        if (res != 0)
            return 1;
    }
}
```
startSecondPass(reader);
resize(result, length(result) + 1);
resize(back(result), length(buffer), Exact());
res = readLine(back(result), reader);
if (res != 0)
    return 1;

return 0;
}

Note that all file contents read in the first pass are buffered when operating on streams. Thus, double-pass I/O can have
a high memory usage on streams when having large passes. In this case, using memory mapped strings to read from
can be more efficient. However, in order to allow double-pass I/O when reading from compressed streams or stdin,
this buffering is designed to lead to better performance or is even required.

Double-pass I/O has the advantage that the exact amount of memory can be allocated for the target data structures.
This can lead to reduced memory usage since no memory is pre-allocated and then left unused. Thus, this is useful if
the life span of your target data structures is long and a lot of memory is saved.

The disadvantage is the higher memory usage when reading the file itself. All data read in the first pass has to be
buffered if using streams.

So, when should you use double-pass I/O? A good rule of thumb is: If you need to read a whole large file into main
memory (e.g. NGS read set or a genome) and it is uncompressed then use a double-pass record reader with a memory
mapped string. Otherwise, use single-pass I/O.
2.1.34 Fragment Store

Learning Objective  You will learn about the SeqAn FragmentStore for handling fragments. The “fragments” are reads and the data structure is useful in the context of read mapping, genome assembly, and gene annotation. After completing this tutorial, you will be able to use the most relevant functionality of the FragmentStore class.

Difficulty  Advanced

Duration  1 h

Prerequisites  The basic tutorials.

The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation. These tasks typically require lots of data structures that are related to each other like:

- reads, mate-pairs, reference genome
- pairwise alignments
- genome annotation

The Fragment Store subsumes all these data structures in an easy to use interface. It represents a multiple alignment of millions of reads or mate-pairs against a reference genome consisting of multiple contigs. Additionally, regions of the reference genome can be annotated with features like ‘gene’, ‘mRNA’, ‘exon’, ‘intron’ or custom features. The Fragment Store supports I/O functions to read/write a read alignment in SAM or AMOS format and to read/write annotations in GFF or GTF format.

The Fragment Store can be compared with a database where each table (called “store”) is implemented as a String member of the FragmentStore class. The rows of each table (implemented as structs) are referred by their ids which are their positions in the string and not stored explicitly (marked with * in the Figures 2 and 5). The only exception is the alignedReadStore whose elements of type AlignedReadStoreElement contain an id-member as they may be rearranged in arbitrary order, e.g. by increasing genomic positions or by readId. Many stores have an associated name store to store element names. Each name store is a StringSet that stores the element name at the position of its id. All stores are present in the Fragment Store and empty if unused. The concrete types, e.g. the position types or read/contig alphabet, can be easily changed by defining a custom config struct which is a template parameter of the Fragment Store class.

Multiple Read Alignment

The Fragment Store can represent a multiple read alignment, i.e. is an alignment between the contigs and the set of reads, where one read can be aligned at zero, one or multiple positions of a contig. In the multiple alignment the contig is represented by one line with gaps (-) and the remaining lines are to reads or read segments with gaps aligned to the contig. The following figure shows one contig (blue line at the top) and multiple reads aligned to it arranged as stairs (reads in lower-case align to the reverse strand):

```
TGAAAACTATATTTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaa ctaatatagtgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
TGAAAACTATATTATGCTATTCAGTTCTAAATAGAAATTGAAACACGCTGTGTTTAGGCTCCTTTGTCCA------ACCCCTTTGCAA/CAACCTTGAGAACAG
tgaaaactatatgctttcagttctaaatagaaatgaaacacgccctttgttcaa------acccctttgcaa/acaacaccttgagaac
```

The Fragment Store can be compared with a database where each table (called “store”) is implemented as a String member of the FragmentStore class. The rows of each table (implemented as structs) are referred by their ids which are their positions in the string and not stored explicitly (marked with * in the Figures 2 and 5). The only exception is the alignedReadStore whose elements of type AlignedReadStoreElement contain an id-member as they may be rearranged in arbitrary order, e.g. by increasing genomic positions or by readId. Many stores have an associated name store to store element names. Each name store is a StringSet that stores the element name at the position of its id. All stores are present in the Fragment Store and empty if unused. The concrete types, e.g. the position types or read/contig alphabet, can be easily changed by defining a custom config struct which is a template parameter of the Fragment Store class.
• **Figure 1:** Multiple read alignment

The following figure shows which tables represent the multiple read alignment:

![Diagram of stores used to represent a multiple read alignment]

The main table is the alignedReadStore which stores AlignedReadStoreElements. Each entry is an alignment of a read (readId) and a contig (contigId). Introduced gaps are stored as a string of gap anchors in the gaps member of the alignedReadStore entry and the contigStore entry. The begin and end positions of the alignment are given by the beginPos and endPos members which are 0-based positions on the forward strand in gap space, i.e. positions in the gapped contig sequence. If the read is aligned to the reverse strand it holds endPos < beginPos. However, the gaps are always related to the forward strand. Additional information, e.g. the number of errors, an alignment score or additional alignment tags, are stored in the tables alignQualityStore and alignedReadTagStore at position id, where id is a unique id of the AlignedReadStoreElement. Paired-end or mate pair alignments are represented by two entries in the alignedReadStore that have the same pairMatchId value (unequal to INVALID_ID). For orphaned read alignments holds pairMatchId == INVALID_ID.

```
012345556789 sequence space
012345678901 gap space
contig ACCAC--GTTTG
read1 ACACGGT [2-9]
read2 ACGGTT-G [4-12]
```

The alignedReadStore is the only store where the id (alignId in the figure) of an element is not implicitly given by its position. The reason for this is that it is necessary in many cases to rearrange the elements of the alignedReadStore,
e.g. increasingly by (contigId,beginPos), by readId or pairMatchId. This can be done by sortAlignedReads. If it is necessary to address an element by its id, the elements must be sorted by id first. In the case that ids are not contiguously increasing, e.g. because some elements where removed, they must be renamed by a prior call of compactAlignedReads. Analogously the function compactPairMatchIds renames pairMatchId values contiguously and replaces values that occur in only one alignment by INVALID_ID.

**Display Aligned Reads**

The multiple read alignment can be displayed in text form or in a scalable graphics format (SVG). Therefore first a stairs layout of the reads must be computed via layoutAlignment and stored in an AlignedReadLayout. The function printAlignment can then be used to output a window (beginPos,endPos,firstLine,lastLine) of the read alignment against a contig either to a stream or SVGFile. The following small example demonstrates how to first load two contigs from a Fasta file and then import a read alignment given in SAM format:

```cpp
#include <iostream>
#include <seqan/store.h>
#include <seqan/misc/misc_svg.h>

using namespace seqan;

int main ()
{
    FragmentStore<> store;
    loadContigs(store, "ex1.fa");
    std::ifstream file("ex1.sam");
    read(file, store, Sam());

    Then we create a stairs layout of the aligned reads and output a window from gapped position 0 to 150 and line 0 to 36 of the multiple alignments below contig 1 to standard out.

    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, Raw(), layout, store, 1, 0, 150, 0, 36);
}
```

Then we create a stairs layout of the aligned reads and output a window from gapped position 0 to 150 and line 0 to 36 of the multiple alignments below contig 1 to standard out.
The same window can also be exported as a scalable vector graphic in SVG format (supported by Browsers, Inkscape; see original file):

```cpp
SVGFile svg("layout.svg");
printAlignment(svg, Raw(), layout, store, 1, 0, 150, 0, 36);
return 0;
```

Figure 2.7: SVG export of a multiple read alignment

### Accessing Pairwise Alignments

In the next step, we want to access several pairwise alignments between reads and contig segments. Therefore we first need to get the associated types that the Fragment Store uses to store contig and read sequences and gaps. This can be done by the following typedefs:

```cpp
typedef Value<TStore::TReadSeqStore>::Type TReadSeq;
typedef Value<TStore::TContigStore>::Type TContig;
typedef Value<TStore::TAlignedReadStore>::Type TAlignedRead;

typedef Gaps<TContig::TContigSeq, AnchorGaps<TContig::TGapAnchors> > TContigGaps;
typedef Gaps<TStore::TReadSeq, AnchorGaps<TAlignedRead::TGapAnchors> > TReadGaps;
```

Now we want to extract and output the alignments from the `alignedReadStore` at position 140,144,...,156. First we store a reference of the alignedRead in `ar` as we need to access it multiple times. The read sequence is neither stored in the `readStore` or `alignedReadStore` as many short sequences can more efficiently be stored in a separate `StringSet` like the `readSeqStore`. We copy the read sequence into a local variable (defined outside the loop to save
allocations/deallocations) as we need to compute the reverse-complement for reads that align to the reverse strand. Then we create a \texttt{gaps} object that represent the alignment rows of the contig and the aligned read in the multiple sequence alignment. The \texttt{Gaps} object requires references of the sequence and the gap-anchor string stored in the \texttt{contigStore} and the \texttt{alignedReadStore}. We need to limit the view of the contig alignment row to the interval the read aligns to, i.e. the gap position interval \texttt{[beginPos,endPos]}. After that we output both alignment rows.

\textbf{Tip:} The \texttt{Gaps} contains two \texttt{Holder} references to the sequence and the inserted gaps. In our example these Holders are dependent and changes made to the \texttt{Gaps} object like the insertion/deletion of gaps would immediately be persistent in the \texttt{Fragment Store}.

\begin{verbatim}
for (int i = 140; i < 160; i += 4)
{
    TAlignedRead &ar = store.alignedReadStore[i];
    readSeq = store.readSeqStore[ar.readId];
    if (ar.endPos < ar.beginPos)
        reverseComplement(readSeq);
    TContigGaps contigGaps(
        store.contigStore[ar.contigId].seq,
        store.contigStore[ar.contigId].gaps);
    TReadGaps readGaps(
        readSeq,
        ar.gaps);
    setPosition(contigGaps, std::min(ar.beginPos, ar.endPos));
    setEndPosition(contigGaps, std::max(ar.beginPos, ar.endPos));
    std::cout << "ALIGNMENT " << i << std::endl;
    std::cout << "contig " << ar.contigId << ": " << contigGaps;
    std::cout << std::endl;
}
\end{verbatim}

ALIGNMENT 140
contig 0: CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAAC [266..306[
read 149: CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAAC

ALIGNMENT 144
contig 0: AGTGCCTTTGTTCACATAGACCCCCTTGCAACAACC [274..310[
read 153: AGTGCCTTTGTTCACATAGACCCCCTTGCAACAACC

ALIGNMENT 148
contig 0: TTCA-----ACCCCCCTTGCAACAACCTTGAGAACCCCAGG [284..324[
read 157: ATAG-----ACCCCCCTTGCAACAACCTTGAGAACCCCAGG

ALIGNMENT 152
contig 0: CA-----ACCCCCCTTGCAACAACCTTGAGAACCCCAGG [286..326[
read 161: CA-----ACCCCCCTTGCAACAACCTTGAGAACCCCAGG

ALIGNMENT 156
contig 0: CCCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT [294..329[
read 165: CCCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT
Assignment 1

Type Rview

Objective Modify the example above, such that reads that align to the reverse strand are displayed in lower-case letters.

Difficulty Easy

Hint The Dna alphabet used in the fragment store doesn’t support lower-case letters. You have to use a string of chars for readSeq.

Solution As we copy the read sequence, it suffices to change the type of the target string readSeq and the sequence type of the read Gaps object into CharString, i.e. a String of char.

```
typedef Value<TStore::TReadSeqStore>::Type TReadSeq;
typedef Value<TStore::TContigStore>::Type TContig;
typedef Value<TStore::TAlignedReadStore>::Type TAlignedRead;

typedef Gaps<TContig::TContigSeq, AnchorGaps<TContig::TGapAnchors> > TContigGaps;
typedef Gaps<CharString, AnchorGaps<TAlignedRead::TGapAnchors> > TReadGaps;

CharString readSeq;
```

Then, we not only need to reverse-complement readSeq if the read aligns to the reverse strand (endPos < beginPos) but also need to convert its letters into lower-case. Therefore SeqAn provides the function toLower. Alternatively, we could iterate over readSeq and add (‘a’-‘A’) to its elements.

```
for (int i = 140; i < 160; i += 4)
{
    TAlignedRead &ar = store.alignedReadStore[i];

    readSeq = store.readSeqStore[ar.readId];
    if (ar.endPos < ar.beginPos)
    {
        reverseComplement(readSeq);
        toLower(readSeq);
    }

    TContigGaps contigGaps(
        store.contigStore[ar.contigId].seq,
        store.contigStore[ar.contigId].gaps);

    TReadGaps readGaps(
        readSeq,
        ar.gaps);

    setBeginPosition(contigGaps, std::min(ar.beginPos, ar.endPos));
    setEndPosition(contigGaps, std::max(ar.beginPos, ar.endPos));

    std::cout << "ALIGNMENT " << i << std::endl;
    std::cout << "\tcontig " << ar.contigId << ":\t" << contigGaps;
    std::cout << "\t[" << beginPosition(contigGaps) << ":" << endPosition(contigGaps) << "]\tread " << ar.readId << ":\t" << readGaps << std::endl;
    std::cout << std::endl;
}
```

Running this program results in the following output.
ALIGNMENT 140  contig 0: CTGTGTAGTGCTTTGGGTTCA—ACCCCCCTTGCAAC [266..306] read 149: ctgtgtagtgctttgcaac—acccccttgcaac

ALIGNMENT 144  contig 0: AGTGCCTTTGGTCA—ACCCCCCTTGCAACAACC [274..310] read 153: AGTCCTTTGGTTCAACAGACCCCCCTTGCAACAACC

ALIGNMENT 148  contig 0: TTCA—ACCCCCCTTGCAACAACC [284..324] read 157: ATAG—ACCCCCCTTGCAACAACCTTGAGAGAACC

ALIGNMENT 152  contig 0: CA—ACCCCCCTTGCAACAACCTTGAGAGAACC [286..326] read 161: CA—ACCCCCCTTGCAACAACCTTGAGAGAACC

ALIGNMENT 156  contig 0: CCCCCCTTGCAACAACCTTGAGAGAACC [294..329] read 165: ccccccttgcaacaaccttgagaggaacc

Gene Annotation

Annotations are represented as a tree that at least contains a root node where all annotations of children or grandchildren of. A typical annotation tree looks as follows:

![Annotation tree example](image)

The following figure shows which tables represent the annotation tree:

Traversing the Annotation Tree

The annotation tree can be traversed and accessed with the AnnotationTree Iterator. A new iterator can be created with begin given a reference to the FragmentStore and the tag AnnotationTree:

```
Iterator<FragmentStore<>, AnnotationTree<>>::Type it;
it = begin(store, AnnotationTree<>());
```

It starts at the root node and can be moved to adjacent tree nodes with the functions goDown, goUp, and goRight. These functions return a boolean value that indicates whether the iterator could be moved. The functions isLeaf, isRoot, isLastChild return the same boolean without moving the iterator. With goRoot or goTo it can be moved to the root node or an arbitrary node given its annotationId. If the iterator should not be moved but a new iterator at an adjacent nodes is required, the functions nodeDown, nodeUp, nodeRight can be used.
The AnnotationTree iterator supports a preorder DFS traversal and therefore can also be used in typical begin-end loops with the functions `goBegin` (== `goRoot`), `goEnd`, `goNext`, `atBegin`, `atEnd`. During a preorder DFS, the descent into subtree can be skipped by `goNextRight`, or `goNextUp` which proceeds with the next sibling or returns to the parent node and proceeds with the next node in preorder DFS.

**Accessing the Annotation Tree**

To access or modify the node an iterator points at, the iterator returns the node’s annotationId by the value function (== `operator*`). With the annotationId the corresponding entry in the annotationStore could be modified manually or by using convenience functions. The function `getAnnotation` returns a reference to the corresponding entry in the annotationStore. `getName` and `setName` can be used to retrieve or change the identifier of the annotation element. As some annotation file formats don’t give every annotation a name, the function `getUniqueName` returns the name if non-empty or generates one using the type and id. The name of the parent node in the tree can be determined with `getParentName`. The name of the annotation type, e.g. ‘mRNA’ or ‘exon’, can be determined and modified with `getType` and `setType`.

An annotation can not only reference a region of a contig but also contain additional information given as key-value pairs. The value of a key can be retrieved or set by `getValueByKey` and `assignValueByKey`. The values of a node can be cleared with `clearValues`.

A new node can be created as first child, last child, or right sibling of the current node with `createLeftChild`, `createRightChild`, or `createSibling`. All three functions return an iterator to the newly created node.

The following tables summarizes the functions provided by the AnnotationTree iterator:
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAnnotation, value</code></td>
<td>Return annotation object/id of current node</td>
</tr>
<tr>
<td><code>[get/set]Name, [get/set]Type</code></td>
<td>Access name or type of current annotation object</td>
</tr>
<tr>
<td><code>clearValues, [get/set]ValueByKey</code></td>
<td>Access associated values</td>
</tr>
<tr>
<td><code>goBegin, goEnd, atBegin, atEnd</code></td>
<td>Go to or test for begin/end of DFS traversal</td>
</tr>
<tr>
<td><code>goNext, goNextRight, goNextUp</code></td>
<td>go next, skip subtree or siblings during DFS traversal</td>
</tr>
<tr>
<td><code>goRoot, goUp, goDown, goRight</code></td>
<td>Navigate through annotation tree</td>
</tr>
<tr>
<td><code>create[Left/Right]Child, createSibling</code></td>
<td>Create new annotation nodes</td>
</tr>
<tr>
<td><code>isRoot, isLeaf</code></td>
<td>Test for root/leaf node</td>
</tr>
</tbody>
</table>

### File I/O

#### Reads and Contigs

To efficiently load reads, use the function `loadReads` which auto-detects the file format, supporting Fasta, Fastq, QSeq and Raw (see `AutoSeqFormat`), and uses memory mapping to efficiently load millions of reads, their names and quality values. If not only one but two file names are given, `loadReads` loads mate pairs or paired-end reads stored in two separate files. Both files are required to contain the same number or reads and reads stored at the same line in both files are interpreted as pairs. The function internally uses `appendRead` or `appendMatePair` and reads distributed over multiple files can be loaded with consecutive calls of `loadReads`.

Contigs can be loaded with the function `loadContigs`. The function loads all contigs given in a single file or multiple files given a single file name or a `StringSet` of file names. The function has an additional boolean parameter `loadSeqs` to load immediately load the contig sequence or if `false` load the sequence later with `loadContig` to save memory, given the corresponding `contigId`. If the contig is accessed by multiple instances/threads the functions `lockContig` and `unlockContig` can be used to ensure that the contig is loaded and release it after use. The function `unlockAndFreeContig` can be used to clear the contig sequence and save memory if the contig is not locked by any instance.

To write all contigs to an open output stream use `writeContigs`.

### Multiple Read Alignments

A multiple read alignment can be loaded from an open input stream with:

```java
read(file, store, Sam()); // reads a SAM file
read(file, store, Amos()); // reads a file in the AMOS assembler format
```

and written to an open output stream with:

```java
write(file, store, Sam()); // writes a SAM file
write(file, store, Amos()); // writes a file in the AMOS assembler format
```

As SAM supports a multiple read alignment (with padding operations in the CIGAR string) but does not enforce its use. That means that a typical SAM file represents a set of pairwise (not multiple) alignments. To convert all the pairwise alignments into a multiple alignments of all reads, `read` internally calls the function `convertPairWiseToGlobalAlignment`. A prior call to `loadReads` is not necessary (but possible) as SAM contains the read names, sequences and quality values. Contigs can be loaded at any time. If they are not loaded before reading a SAM file, empty sequences are created with the names referred in the SAM file. A subsequent call of `loadContigs` would load the sequences of these contigs, if they have the same identifier in the contig file.

### Annotations

A annotation file can be read from an open input stream with:
The GFF-reader is also able to detect and read GTF files. As the knownGene.txt and knownIsoforms.txt files are two separate files used by the UCSC Genome Browser, they must be read by two consecutive calls of `read` (first knownGene.txt then knownIsoforms.txt). An annotation can be loaded without loading the corresponding contigs. In that case empty contigs are created in the contigStore with names given in the annotation. A subsequent call of `loadContigs` would load the sequences of these contigs, if they have the same identifier in the contig file.

To write an annotation to an open output stream use:

```cpp
write(file, store, Gff()); // writes a GFF file
write(file, store, Gtf()); // writes a GTF file
```

Please note, that UCSC files cannot be written due to limitations of the file format.

**Stores**

The Fragment Store consists of the following tables:

**Read Stores**

<table>
<thead>
<tr>
<th>Store</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>readStore</td>
<td>Reads</td>
<td>String mapping from <code>readId</code> to <code>matePairId</code></td>
</tr>
<tr>
<td>readSeqStore</td>
<td>Read sequences</td>
<td>String mapping from <code>readId</code> to <code>readSeq</code></td>
</tr>
<tr>
<td>matePairStore</td>
<td>Mate-pairs / pairs of reads</td>
<td>String mapping from <code>matePairId</code> to <code>&lt;readId[2], libId&gt;</code></td>
</tr>
<tr>
<td>libraryStore</td>
<td>Mate-pair libraries</td>
<td>String mapping from <code>libId</code> to <code>&lt;mean, std&gt;</code></td>
</tr>
</tbody>
</table>

**Contig Stores**

<table>
<thead>
<tr>
<th>Store</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>contigStore</td>
<td>Contig sequences with gaps</td>
<td>String that maps from <code>contigId</code> to <code>&lt;contigSeq, contigGaps, contigFileId&gt;</code></td>
</tr>
<tr>
<td>contigFileStore</td>
<td>Stores information how to load contigs on-demand</td>
<td>String that maps from <code>contigFileId</code> to <code>&lt;fileName, firstContigId&gt;</code></td>
</tr>
</tbody>
</table>

**Read Alignment Stores**

<table>
<thead>
<tr>
<th>Store</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignedReadStore</td>
<td>Alignments of reads against contigs</td>
<td>String that stores <code>&lt;alignId, readId, contigId, pairMatchId, beginPos, endPos, gaps&gt;</code></td>
</tr>
<tr>
<td>alignedReadTagStore</td>
<td>Additional alignment tags (used in SAM)</td>
<td>String that maps from <code>alignId</code> to <code>alignTag</code></td>
</tr>
<tr>
<td>alignQualityStore</td>
<td>Mapping quality of read alignments</td>
<td>String that maps from <code>alignId</code> to <code>&lt;pairScore, score, errors&gt;</code></td>
</tr>
</tbody>
</table>
Annotation Stores

<table>
<thead>
<tr>
<th>Store</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotation-Store</td>
<td>Annotations of contig regions</td>
<td>String that maps from annoId to &lt;contigId, typeId, beginPos, endPos, parentId, lastChildId, nextSiblingId, values&gt;</td>
</tr>
</tbody>
</table>

Name Stores

<table>
<thead>
<tr>
<th>Store</th>
<th>Annotation names</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotationNameStore</td>
<td>Annotation names</td>
<td>String that maps from annoId to annoName</td>
</tr>
<tr>
<td>readNameStore</td>
<td>Read identifiers (Fasta ID)</td>
<td>String that maps from readId to readName</td>
</tr>
<tr>
<td>contigNameStore</td>
<td>Contig identifiers (Fasta ID)</td>
<td>String that maps from contigId to contigName</td>
</tr>
<tr>
<td>matePairNameStore</td>
<td>Mate-pair identifiers</td>
<td>String that maps from contigId to contigName</td>
</tr>
<tr>
<td>libraryNameStore</td>
<td>Mate-pair library identifiers</td>
<td>String that maps from libId to libName</td>
</tr>
</tbody>
</table>

ToC

Contents

- Simple RNA-Seq
  - Introduction to the used Data Structures
    - Fragment Store
    - Annotation Tree
    - Interval Tree
    - Import Alignments and Gene Annotations from File
      - Assignment 1
    - Extract Gene Intervals
      - Assignment 2
    - Construct Interval Trees
      - Assignment 3
    - Compute Gene Coverage
      - Assignment 4
    - Output RPKM Values
      - Assignment 5
  - Next Steps

2.1.35 Simple RNA-Seq

Learning Objective You will learn how to write a simple gene quantification tool based on RNA-Seq data.

Difficulty Hard

Duration 2h

Prerequisites Genome Annotations, Fragment Store, experience with OpenMP (optional)

RNA-Seq refers to high-throughput sequencing of cDNA in order to get information about the RNA molecules available in a sample. Knowing the sequence and abundance of mRNA allows to determine the (differential) expression of genes, to detect alternative splicing variants, or to annotate yet unknown genes.

In the following tutorial you will develop a simple gene quantification tool. It will load a file containing gene annotations and a file with RNA-Seq read alignments, computes abundances, and outputs RPKM values for each expressed
gene.

Albeit its simplicity, this example can be seen as a starting point for more complex applications, e.g. to extend the tool from quantification of genes to the quantification of (alternatively spliced) isoforms, or to de-novo detect yet unannotated isoforms/genes.

You will learn how to use the FragmentStore to access gene annotations and alignments and how to use the IntervalTree to efficiently determine which genes overlap a read alignment.

Introduction to the used Data Structures

This section introduces the FragmentStore and the IntervalTree, which are the fundamental data structures used in this tutorial to represent annotations and read alignments and to efficiently find overlaps between them. You may skip one or both subsections if you are already familiar with one or both data structures.

Fragment Store

The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation. These tasks typically require lots of data structures that are related to each other such as

- reads, mate-pairs, reference genome
- pairwise alignments, and
- genome annotation.

The Fragment Store subsumes all these data structures in an easy to use interface. It represents a multiple alignment of millions of reads or mate-pairs against a reference genome consisting of multiple contigs. Additionally, regions of the reference genome can be annotated with features like ‘gene’, ‘mRNA’, ‘exon’, ‘intro’ or custom features. The Fragment Store supports I/O functionality to read/write a read alignment in SAM or AMOS format and to read/write annotations in GFF or GTF format.

The Fragment Store can be compared with a database where each table (called “store”) is implemented as a String member of the FragmentStore class. The rows of each table (implemented as structs) are referred by their ids which are their positions in the string and not stored explicitly. The only exception is the alignedReadStore whose elements of type AlignedReadStoreElement contain an id-member as they may be rearranged in arbitrary order, e.g. by increasing genomic positions or by readId. Many stores have an associated name store to store element names. Each name store is a StringSet that stores the element name at the position of its id. All stores are present in the Fragment Store and empty if unused. The concrete types, e.g. the position types or read/contig alphabet, can be easily changed by defining a custom config struct which is a template parameter of the Fragment Store class.

Annotation Tree

Annotations are represented as a tree that at least contains a root node where all annotations are children or grandchildren of. A typical annotation tree looks as follows:

In the Fragment Store the tree is represented by annotationStore, annotationTypeStore, annotationKeyStore, and others. Instead of accessing these tables directly, the AnnotationTree Iterator provides a high-level interface to traverse and access the annotation tree.

Interval Tree

The IntervalTree is a data structure that stores one-dimensional intervals in a balanced tree and efficiently answers range queries. A range query is an operation that returns all tree intervals that overlap a given query point or interval.
The interval tree implementation provided in SeqAn is based on a Tree which is balanced if all intervals are given at construction time. Interval tree nodes are objects of the IntervalAndCargo class and consist of 2 interval boundaries and additional user-defined information, called cargo. To construct the tree on a set of given interval nodes use the function createIntervalTree. The functions addInterval and removeInterval should only be used if the interval tree needs to be changed dynamically (as they not yet balance the tree).

Import Alignments and Gene Annotations from File

At first, our application should create an empty FragmentStore object into which we import a gene annotation file and a file with RNA-Seq alignments. An empty FragmentStore can simply be created with:

```
FragmentStore<> store;
```

Files can be read from disk with the function read that expects an open stream (e.g. a STL ifstream), a FragmentStore object, and a File Format tag. The contents of different files can be loaded with subsequent calls of read. As we want the user to specify the files via command line, our application will parse them using the ArgumentParser and store them in an option object.

In your first assignment you need to complete a given code template and implement a function that loads a SAM file and a GTF file into the FragmentStore.

Assignment 1

**Type** Application

**Objective** Use the code template below (click more...) and implement the function loadFiles to load the annotation and alignment files. Use the file paths given in the options object and report an error if the files could not be opened.

```
#include <iostream>
#include <seqan/store.h>
#include <seqan/arg_parse.h>
#include <seqan/misc/misc_interval_tree.h>
#include <seqan/parallel.h>

using namespace seqan;
```
// define used types
typedef FragmentStore<> TStore;

// define options
struct Options {
    std::string annotationFileName;
    std::string alignmentFileName;
};

// 1. Parse command line and fill Options object
//
ArgumentParser::ParseResult parseOptions(Options & options, int argc, char const * argv[]) {
    ArgumentParser parser("gene_quant");
    setShortDescription(parser, "A simple gene quantification tool");
    setVersion(parser, "1.0");
    setDate(parser, "Sep 2012");

    addArgument(parser, ArgParseArgument(ArgParseArgument::INPUTFILE));
    addArgument(parser, ArgParseArgument(ArgParseArgument::INPUTFILE));
    addUsageLine(parser, "[\fIOPTION\fP] <\fIANNOTATION FILE\fP> <\fIREAD ALIGNMENT FILE\fP>");

    // Parse command line
    ArgumentParser::ParseResult res = parse(parser, argc, argv);

    if (res == ArgumentParser::PARSE_OK) {
        // Extract option values
        getArgumentValue(options.annotationFileName, parser, 0);
        getArgumentValue(options.alignmentFileName, parser, 1);
    }

    return res;
}

// 2. Load annotations and alignments from files
//
bool loadFiles(TStore & store, Options const & options) {
    // INSERT YOUR CODE HERE ...
    //
    return true;
}

int main(int argc, char const * argv[]) {
    Options options;
    TStore store;

    ArgumentParser::ParseResult res = parseOptions(options, argc, argv);

if (res != ArgumentParser::PARSE_OK)
    return res == ArgumentParser::PARSE_ERROR;

if (!loadFiles(store, options))
    return 1;
    return 0;
}

Hint

• Open STL std::fstream objects and use the function read with a SAM or GTF tag.
• ifstream::open requires the file path to be given as a C-style string (const char *).
• Use string::c_str to convert the option strings into C-style strings.
• The function read expects a stream, a FragmentStore and a tag, i.e. Sam() or Gtf().

Solution

// 2. Load annotations and alignments from files

bool loadFiles(TStore & store, Options const & options)
{
    std::ifstream alignmentFile(options.alignmentFileName.c_str());
    if (!alignmentFile.good())
    {
        std::cerr << "Couldn’t open alignment file " << options.alignmentFileName << std::endl;
        return false;
    }
    std::cerr << "Loading read alignments ..... " << std::flush;
    read(alignmentFile, store, Sam());
    std::cerr << "[" << length(store.alignedReadStore) << "]" << std::endl;

    // load annotations
    std::ifstream annotationFile(options.annotationFileName.c_str());
    if (!annotationFile.good())
    {
        std::cerr << "Couldn’t open annotation file" << options.annotationFileName << std::endl;
        return false;
    }
    std::cerr << "Loading genome annotation ... " << std::flush;
    read(annotationFile, store, Gtf());
    std::cerr << "[" << length(store.annotationStore) << "]" << std::endl;

    return true;
}

Extract Gene Intervals

Now that the Fragment Store contains the whole annotation tree, we want to traverse the genes and extract the genomic
ranges they span. In the annotation tree, genes are (the only) children of the root node. To efficiently retrieve the genes
that overlap read alignments later, we want to use interval trees, one for each contig. To construct an interval tree,
we first need to collect IntervalAndCargo objects in a string and pass them to createIntervalTree. See the interval
tree demo in core/demos/interval_tree.cpp for more details. As cargo we use the gene’s annotation id
to later retrieve all gene specific information. The strings of IntervalAndCargo objects should be grouped by
contigId and stored in an (outer) string of strings. For the sake of simplicity we don’t differ between genes on the forward or reverse strand and instead always consider the corresponding intervals on the forward strand.

To define this string of strings of IntervalAndCargo objects, we first need to determine the types used to represent an annotation. All annotations are stored in the annotationStore which is a Fragment Store member and whose type is TAnnotationStore. The value type of the annotation store is the class AnnotationStoreElement. Its member typedefs TPos and TId define the types it uses to represent a genomic position or the annotation or contig id:

```cpp
typedef FragmentStore<> TStore;
typedef Value<TStore::TAnnotationStore>::Type TAnnotation;
typedef TAnnotation::TId TId;
typedef TAnnotation::TPos TPos;
typedef IntervalAndCargo<TPos, TId> TInterval;
```

The string of strings of intervals can now be defined as:

```cpp
String<String<TInterval> > intervals;
```

In your second assignment you should use an AnnotationTree Iterator annotation tree iterator] to traverse all genes in the annotation tree. For each gene, determine its genomic range (projected to the forward strand) and add a new TInterval object to the intervals[contigId] string, where contigId is the id of the contig containing that gene.

**Assignment 2**

**Type** Application

**Objective** Use the code template below (click more..). Implement the function `extractGeneIntervals` that should extract genes from the annotation tree (see AnnotationTree Iterator) and create strings of IntervalAndCargo objects - one for each config - that contains the interval on the forward contig strand and the gene’s annotation id.

Extend the definitions:

```cpp
// define used types
typedef FragmentStore<> TStore;
typedef Value<TStore::TAnnotationStore>::Type TAnnotation;
typedef TAnnotation::TId TId;
typedef TAnnotation::TPos TPos;
typedef IntervalAndCargo<TPos, TId> TInterval;
```

Add a function:

```cpp
// 3. Extract intervals from gene annotations (grouped by contigId)
void extractGeneIntervals(String<String<TInterval> > & intervals, TStore const & store)
{
    // INSERT YOUR CODE HERE ...
}
```

Extend the main function:

```cpp
TStore store;
String<String<TInterval> > intervals;
```

and
if (!loadFiles(store, options))
    return 1;

extractGeneIntervals(intervals, store);

**Hint** You can assume that all genes are children of the root node, i.e. create an `AnnotationTree` iterator, go down to the first gene and go right to visit all other genes. Use `getAnnotation` to access the gene annotation and `value` to get the annotation id.

Make sure that you append `IntervalAndCargo` objects, where \(i_1 < i_2\) holds, as opposed to annotations where \(\text{beginPos} > \text{endPos}\) is possible. Remember to ensure that `intervals` is of appropriate size, e.g. with `resize(intervals, length(store.contigStore));`

Use `appendValue` to add a new `TInterval` object to the inner string, see `IntervalAndCargo constructor` for the constructor.

**Solution**

```cpp
// 3. Extract intervals from gene annotations (grouped by contigId)

void extractGeneIntervals(String<String<TInterval>> & intervals, TStore const & store) {
    // extract intervals from gene annotations (grouped by contigId)
    resize(intervals, length(store.contigStore));

    Iterator<TStore const, AnnotationTree<> >::Type it = begin(store, AnnotationTree<>());
    if (!goDown(it))
        return;

    do {
        SEQAN_ASSERT_EQ(getType(it), "gene");

        TPos beginPos = getAnnotation(it).beginPos;
        TPos endPos = getAnnotation(it).endPos;
        TId contigId = getAnnotation(it).contigId;

        if (beginPos > endPos)
            std::swap(beginPos, endPos);

        // insert forward-strand interval of the gene and its annotation id
        appendValue(intervals[contigId], TInterval(beginPos, endPos, value(it)));
    } while (goRight(it));
}
```

**Construct Interval Trees**

With the strings of gene intervals - one for each contig - we now can construct interval trees. Therefore, we specialize an `IntervalTree` with the same position and cargo types as used for the `IntervalAndCargo` objects. As we need an interval tree for each contig, we instantiate a string of interval trees:

```cpp
typedef IntervalTree<TPos, TId> TIntervalTree;
String<TIntervalTree> intervalTrees;
```
Your third assignment is to implement a function that constructs the interval trees for all contigs given the string of interval strings.

Assignment 3

Type  Application

Objective  Use the code template below (click more...). Implement the function `constructIntervalTrees` that uses the interval strings to construct for each contig an interval tree. Optional: Use OpenMP to parallelize the construction over the contigs, see SEQUAN_OMP_PRAGMA.

Extend the definitions:

```cpp
// define used types
typedef FragmentStore<> TStore;
typedef Value<TStore::TAnnotationStore>::Type TAnnotation;
typedef TAnnotation::TId TId;
typedef TAnnotation::TPos TPos;
typedef IntervalAndCargo<TPos, TId> TInterval;
typedef IntervalTree<TPos, TId> TIntervalTree;
```

Add a function:

```cpp
// 4. Construct interval trees

void constructIntervalTrees(String<TIntervalTree> & intervalTrees, String<String<TInterval> > & intervals)
{
    // INSERT YOUR CODE HERE ...
    //
}
```

Extend the main function:

```cpp
String<String<TInterval> > intervals;
String<TIntervalTree> intervalTrees;

and

    extractGeneIntervals(intervals, store);
    constructIntervalTrees(intervalTrees, intervals);
```

Hint  First, resize the string of interval trees accordingly:

```cpp
resize(intervalTrees, length(intervals));
```

Hint  Use the function `createIntervalTree`.

Optional: Construct the trees in parallel over all contigs with an OpenMP parallel for-loop, see here for more information about OpenMP.

Solution

```cpp
// 4. Construct interval trees

void constructIntervalTrees(String<TIntervalTree> & intervalTrees, String<String<TInterval> > & intervals)
{
    int numContigs = length(intervals);
```
Compute Gene Coverage

To determine gene expression levels, we first need to compute the read coverage, i.e. the total number of reads overlapping a gene. Therefore we use a string of counters addressed by the annotation id.

```cpp
String<unsigned> readsPerGene;
```

For each read alignment we want to determine the overlapping genes by conducting a range query via `findIntervals` and then increment their counters by 1. To address the counter of a gene, we use its annotation id stored as cargo in the interval tree.

Read alignments are stored in the `alignedReadStore`, a string of `AlignedReadStoreElements` objects. Their actual type can simply be determined as follows:

```cpp
typedef Value<TStore::TAlignedReadStore>::Type TAlignedRead;
```

Given the `contigId`, `beginPos`, and `endPos` we will retrieve the annotation ids of overlapping genes from the corresponding interval tree.

Your fourth assignment is to implement the count function that performs all the above described steps. Optionally, use OpenMP to parallelize the counting.

Assignment 4

**Type** Application

**Objective** Use the code template below (click more...). Implement the function `countReadsPerGene` that counts for each gene the number of overlapping reads. Therefore determine for each `AlignedReadStoreElement` begin and end positions (on forward strand) of the alignment and increment the `readsPerGene` counter for each overlapping gene.

**Optional:** Use OpenMP to parallelize the function, see `SEQAN_OMP_PRAGMA`.

Extend the definitions:

```cpp
// define used types
typedef FragmentStore<> TStore;
typedef Value<TStore::TAnnotationStore>::Type TAnnotation;
typedef TAnnotation::TId TId;
typedef TAnnotation::TPos TPos;
typedef IntervalAndCargo<TPos, TId> TInterval;
typedef IntervalTree<TPos, TId> TIntervalTree;
typedef Value<TStore::TAlignedReadStore>::Type TAlignedRead;
```

Add a function:

```cpp
// 5. Count reads per gene
//
void countReadsPerGene(String<unsigned> & readsPerGene, String<TIntervalTree> const & intervalTrees) {
```
// INSERT YOUR CODE HERE ...

Extend the main function:

```c++
String<TIntervalTree> intervalTrees;
String<unsigned> readsPerGene;
```

and

```c++
extractGeneIntervals(intervals, store);
constructIntervalTrees(intervalTrees, intervals);
countReadsPerGene(readsPerGene, intervalTrees, store);
```

**Hint**

```c++
resize(readsPerGene, length(store.annotationStore), 0);
```

Make sure that you search with `findIntervals` where `query_begin < query_end` holds, as opposed to read alignments where `beginPos > endPos` is possible.

**Hint**

The result of a range query is a string of annotation ids given to `findIntervals` by-reference:

```c++
String<TId> result;
```

Reuse the result string for multiple queries (of the same thread, use `private(result)` for OpenMP).

**Solution**

```c++
// 5. Count reads per gene

void countReadsPerGene(String<unsigned> & readsPerGene, String<TIntervalTree> const & intervalTrees, TStore const & store)
{
    resize(readsPerGene, length(store.annotationStore), 0);
    String<TId> result;
    int numAlignments = length(store.alignedReadStore);

    // iterate aligned reads and get search their begin and end positions
    SEQAN_OMP_PRAGMA(parallel for private(result))
    for (int i = 0; i < numAlignments; ++i)
    {
        TAlignedRead const & ar = store.alignedReadStore[i];
        TPos queryBegin = _min(ar.beginPos, ar.endPos);
        TPos queryEnd = _max(ar.beginPos, ar.endPos);

        // search read-overlapping genes
        findIntervals(intervalTrees[ar.contigId], queryBegin, queryEnd, result);

        // increase read counter for each overlapping annotation given the id in the interval tree
        for (unsigned j = 0; j < length(result); ++j)
        {
            SEQAN_OMP_PRAGMA(atomic)
            readsPerGene[result[j]] += 1;
        }
    }
}
```
Output RPKM Values

In the final step, we want to output the gene expression levels in a normalized measure. We therefore use **RPKM** values, i.e. the number of reads per kilobase of exon model per million mapped reads (1). One advantage of RPKM values is their independence of the sequencing throughput (normalized by total mapped reads), and that they allow to compare the expression of short with long transcripts (normalized by exon length).

The exon length of an mRNA is the sum of lengths of all its exons. As a gene may have multiple mRNA, we will simply use the maximum of all their exon lengths.

Your final assignment is to output the RPKM value for genes with a read counter \( > 0 \). To compute the exon length of the gene (maximal exon length of all mRNA) use an `AnnotationTree Iterator` and iterate over all mRNA (children of the gene) and all exons (children of mRNA). For the number of total mapped reads simply use the number of alignments in the `alignedReadStore`. Output the gene names and their RPKM values separated by tabs as follows:

```
#gene name   RPKM value
ENSMUSG00000053211 5932.12
ENSMUSG00000069053 10540.1
ENSMUSG00000056673 12271.3
ENSMUSG00000069049 10742.2
ENSMUSG00000091749 7287.66
ENSMUSG00000068457 37162.8
ENSMUSG00000069045 13675
ENSMUSG00000069044 6380.36
ENSMUSG00000077793 2088.62
ENSMUSG0000000103 7704.74
ENSMUSG00000091571 10965.2
ENSMUSG00000069036 127128
ENSMUSG00000090405 10965.2
ENSMUSG00000090652 35271.2
ENSMUSG00000052831 68211.2
ENSMUSG00000069031 37564.2
ENSMUSG00000071960 34984
ENSMUSG00000091987 37056.3
ENSMUSG00000090600 2310.18
```

Download and decompress the attached mouse annotation ([raw-attachment:Mus_musculus.NCBIM37.61.gtf.zip](Mus_musculus.NCBIM37.61.gtf.zip)) and the alignment file of RNA-Seq reads aligned to chromosome Y ([raw-attachment:sim40mio_onlyY.sam.zip](sim40mio_onlyY.sam.zip)). Test your program and compare your output with the output above.

Assignment 5

**Type**  Application

**Objective** Use the code template below (click more...). Implement the function `outputGeneCoverage` that outputs for each expressed gene the gene name and the expression level as RPKM as tab-separated values.

```
// 6. Output RPKM values

void outputGeneCoverage(String<unsigned> const & readsPerGene, TStore const & store)
{
    // INSERT YOUR CODE HERE ...
}
```

Add a function:

```cpp
void outputGeneCoverage(String<unsigned> const & readsPerGene, TStore const & store) {
    // INSERT YOUR CODE HERE ...
}
```
Extend the main function:

```c
extractGeneIntervals(intervals, store);
constructIntervalTrees(intervalTrees, intervals);
countReadsPerGene(readsPerGene, intervalTrees, store);
outputGeneCoverage(readsPerGene, store);
```

**Hint** To compute the maximal exon length use three nested loops: (1) enumerate all genes, (2) enumerate all mRNA of the gene, and (3) enumerate all exons of the mRNA and sum up their lengths.

**Hint** Remember that exons are not the only children of mRNA.

**Solution**

```c
// 6. Output RPKM values
void outputGeneCoverage(String<unsigned> const & readsPerGene, TStore const & store) {
    // output abundances for covered genes
    Iterator<TStore const, AnnotationTree<>>::Type transIt = begin(store, AnnotationTree<>());
    Iterator<TStore const, AnnotationTree<>>::Type exonIt;
    double millionMappedReads = length(store.alignedReadStore) / 1000000.0;

    std::cout << "#gene name\tRPKM value" << std::endl;
    for (unsigned j = 0; j < length(readsPerGene); ++j) {
        if (readsPerGene[j] == 0) continue;

        unsigned mRNALengthMax = 0;
        goTo(transIt, j);

        // determine maximal mRNA length (which we use as gene length)
        SEQAN_ASSERT_NOT(isLeaf(transIt));
        goDown(transIt);
        do {
            exonIt = nodeDown(transIt);
            unsigned mRNALength = 0;

            // determine mRNA length, sum up the lengths of its exons
            do {
                if (getAnnotation(exonIt).typeId == store.ANNO_EXON)
                    mRNALength += abs((int)getAnnotation(exonIt).beginPos - (int)getAnnotation(exonIt).endPos);
            } while (goRight(exonIt));

            if (mRNALengthMax < mRNALength)
                mRNALengthMax = mRNALength;
        } while (goRight(transIt));

        // RPKM is number of reads mapped to a gene divided by its gene length in kbps
        // and divided by millions of total mapped reads
        std::cout << store.annotationNameStore[j] << ' ' << readsPerGene[j] / (mRNALengthMax / 1000.0) / millionMappedReads << std::endl;
    }
```

368 Chapter 2. API Documentation
Next Steps

- See [MWM+08] for further reading.
- Read the Basic SAM and BAM I/O Tutorial and change your program to stream a SAM file instead of loading it as a whole.
- Change the program such that it attaches the RPKM value as a key-value pair (see assignValueByKey) to the annotation of each gene and output a GFF file.
- Continue with the Tutorial rest of the tutorials]].

ToC

Contents

- Simple Read Mapping
  - Try It
  - Code Walkthrough
  - Hands On!
    - Task 1: Use the ArgumentParser
    - Task 2: Allow Edit Distance for Verification
    - Task 3: Find Matches On Reverse-Complement
    - Task 4: Allow Other Output Formats

2.1.36 Simple Read Mapping

Learning Objective  You will be able to write read mappers using SeqAn.

Difficulty  Hard

Duration  2 h

Prerequisites  Indices, Fragment Store

In this tutorial, we will walk you through the code of a simple read mapper minimapper that uses the SWIFT filter and uses approximate string search for verification. There are severe limitations to its capabilities but it’s a read mapper in 12 effective lines of code (ignoring includes, comments, typedefs, I/O and lines with closing braces).

Try It

You can find the source code in the directory `core/demos/tutorial/read_mapping/core/demos/tutorial/read_mapping`

Copy over the FASTA files into your build directory and test it:

```
$ cp .../core/demos/tutorial/read_mapping/.*.fasta .
$ make demo_tutorial_minimapper
...
$ ./core/demos/tutorial/read_mapping/demo_tutorial_minimapper
Invalid number of arguments.
USAGE: minimapper GENOME.fasta READS.fasta OUT.sam
$ ./core/demos/tutorial/read_mapping/tutorial_minimapper nc_001454.fasta reads_hamming.fasta out.sam
```
Code Walkthrough

First, include the headers of the SeqAn modules we will use.

```cpp
#include <cstdio>
#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>
using namespace seqan;
```

We will now use some typedefs for the FragmentStore and SWIFT filter to get shortcuts to types used below.

```cpp
// Some typedefs.
typedef FragmentStore<>::TReadSeqStore TReadSeqStore;
typedef Value<TReadSeqStore>::Type TReadSeq;
typedef FragmentStore<>::TContigStore TContigStore;
typedef Value<TContigStore>::Type TContigStoreElement;
typedef TContigStoreElement::TContigSeq TContigSeq;
typedef Index<TReadSeqStore, IndexQGram<Shape<Dna, UngappedShape<11> >, OpenAddressing> > TIndex;
typedef Pattern<TIndex, Swift<SwiftSemiGlobal> > TPattern;
typedef Finder<TContigSeq, Swift<SwiftSemiGlobal> > TFinder;
typedef FragmentStore<>::TAlignedReadStore TAlignedReadStore;
typedef Value<TAlignedReadStore>::Type TAlignedRead;
```

We define the global constant `EPSILON()`, the allowed error rate.

```cpp
const double EPSILON = 0.08;
```

Evaluate the arguments from the command line. Use the functions `loadContigs` and `loadReads` to load the reference sequence (possibly more than one if the FASTA file contains more than one sequence) and reads into the FragmentStore. Note that these functions will automatically guess the file type for you.

```cpp
int main(int argc, char *argv[]) {
    // 0) Handle command line arguments.
    if (argc < 3) {
        std::cerr << "Invalid number of arguments." << std::endl
                  << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
        return 1;
    }

    // 1) Load contigs and reads.
    FragmentStore<> fragStore;
    if (!loadContigs(fragStore, argv[1])) return 1;
    if (!loadReads(fragStore, argv[2])) return 1;
```

Initialize `Finder` and `Pattern` for the q-gram index used by the swift filter.
// 2) Build an index over all reads and a SWIFT pattern over this index.
TIndex index(fragStore.readSeqStore);
TPattern pattern(index);

Now, iterate over all input sequence contigs and enumerate all SWIFT hits. These hits will contain all possible matches of the reads in the FragmentStore with up to $\varepsilon \cdot \ell$ (with $\ell =$ length(read)) errors. Mismatches and indels are taken into consideration.

// 3) Enumerate all epsilon matches.
for (unsigned i = 0; i < length(fragStore.contigStore); ++i) {
    TFinder finder(fragStore.contigStore[i].seq);
    while (find(finder, pattern, EPSILON)) {

Now, verify each possible match using a HammingSimplePattern. The verified matches will have Hamming distance $< \lfloor \varepsilon \cdot \ell \rfloor$, edit distance is not considered.

// Verify match.
Finder<TContigSeq> verifyFinder(fragStore.contigStore[i].seq);
setPosition(verifyFinder, beginPosition(finder));
Pattern<TReadSeq, HammingSimple> verifyPattern(fragStore.readSeqStore[position(pattern).i1]);
unsigned readLength = length(fragStore.readSeqStore[position(pattern).i1]);
int minScore = -static_cast<int>(EPSILON * readLength);
while (find(verifyFinder, verifyPattern, minScore) && position(verifyFinder) < endPosition(infix(finder))) {
    TAlignedRead match(length(fragStore.alignedReadStore), position(pattern).i1, i,
                        beginPosition(verifyFinder), endPosition(verifyFinder));
    appendValue(fragStore.alignedReadStore, match);
}
}

Finally, write out the resulting multiple read alignment to the SAM file with the file name on the command line.

// 4) Write out Sam file.
std::ofstream samFile(argv[3], std::ios_base::out);
write(samFile, fragStore, Sam());
return 0;
}

Hands On!

Programming can only be learned by programming, so let’s get started. We create a new sandbox and a new app for the minimapper. If you already have a sandbox, then you can skip the first step

$ ./util/bin/skel.py repository sandbox/my_sandbox
$ ./util/bin/skel.py app minimapper sandbox/my_sandbox

Now, we copy over the code from the original location into our new app and build it.

$ cp core/demos/tutorial/read_mapping/minimapper.cpp sandbox/my_sandbox/apps/minimapper/minimapper.cpp
$ cd build/Debug
$ cmake .
$ make minimapper
$ ./sandbox/my_sandbox/apps/minimapper/minimapper

Invalid number of arguments.
USAGE: minimapper GENOME.fasta READS.fasta OUT.sam

2.1. Tutorial
Now, play around with the source code. Here are some examples for things to try out. There are no solutions, and they are merely thought to get you started playing...

**Task 1: Use the ArgumentParser**

Global constants are kind of inflexible. Instead of the global constant EPSILON, create an Options struct with a member variable epsilon, initialize it to 0.8 in the constructor and use an Option struct in the main program. Make the value for configurable using the class ArgumentParser described in the Parsing Command Line Arguments Tutorial.

**Task 2: Allow Edit Distance for Verification**

Currently, the read mapper can only find reads with mismatches but not with indels. The SWIFT filter will already create hits for positions with indels so you only have to adjust the verification step.

**Hint** Use the Myers Pattern for the approximate search. Don’t forget to call findBegin using the score (getScore) of the last hit as the find begin score. You can use one Myers Pattern object per read sequence to only perform the precomputation once. If you reuse your finder object, don’t forget to call clear.

**Task 3: Find Matches On Reverse-Complement**

Another limitation is that only reads from the forward strand will be found. Either reverse-complement all reads or the contigs to find reads from the reverse strand.

**Hint** Reverse-complementing the contigs will be faster in practice: First, an index is built over the reads which would have to be built twice if the reads were complemented. Second, there will usually be more reads data than genome data if the coverage is greater than 1.

**Task 4: Allow Other Output Formats**

Read the documentation on the function write of the class FragmentStore.
Background

Bowtie is based on the FM index [FM01] which uses the Burrows-Wheeler transform (BWT) to search for patterns in a text in $O(m)$ time with $m$ being the pattern length. Furthermore the index only consumes minute space because it can be compressed using favorable features of the BWT.

Since this tutorial serves as an example of SeqAn’s functionality, we will not completely re-implement Bowtie. Instead we will concentrate on providing a simple yet functional read mapper. We will start by reading the reference sequence and reads, then search approximately each read in the reference allowing at most one mismatch, and finally write the result into an informative SAM file.

We do make use of the FM index, but we use only one key feature of Bowtie, namely a two phase search. Such two phase search speeds up approximate search on the FM index but requires two indices, i.e. one of the forward and one of the backward reference sequence. Instead of searching each read in one phase, allowing one mismatch in every possible position, we search it twice, but in two phases having some restrictions. We first search each read in the forward index by allowing one mismatch only in the second half of the read. Then we reverse each read and search it in the backward index, again by allowing one mismatch only in the second half of the read. The advantage of a two phase search will become clear in the following.

**Tip:** The FM index implicitly represent a prefix trie of a text. Searching on the FM index is performed backwards, starting with the last character of the pattern and working its way to the first.

In the first search phase we search backwards the right half of the read without mismatches. If we succeed, in the second phase we continue backwards allowing one mismatch in the left half. The whole procedure is based on a depth-first traversal on the prefix trie of the reference. Trie’s edges are labeled with single characters and each path following some edges spells a substring of the reference. Thus, first we start in the root node and follow the path spelling the right half of the read. If we succeed, we continue in the subtree rooted in the previously reached node by following each possible path that spells the left half of the read with one substitution.

The red arrows indicate the exact search of the right half of the read, the green arrows the approximate search of the left half.

From the figure above you can see the advantage of using a two phase approach. By starting from the root node and performing exact search, we follow only one path and avoid visiting the top of the trie. Approximate search is more involved, since we have to try every path until we find a second mismatch, but it is limited to a small subtree.

So far we have found all locations of a read having at most one mismatch in its left half. In order to find also all locations having at most one mismatch in the right half, we simply reverse the reference and the read and repeat the procedure from above. Note that we only reverse but do not complement the reference or the reads.

**Assignment 1**

**Type** Application

**Objective** Copy the code below and rearrange the rows such that they give a functional order.
#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>

using namespace seqan;

void search() {};

int main(int argc, char *argv[]) {
    // type definitions
    typedef String<Dna5> TString;
    typedef StringSet<TString> TStringSet;
    typedef Index<StringSet<TString>, FMIndex<> > TIndex;
    typedef Iterator<TIndex, TopDown<ParentLinks<> > >::Type TIter;

    // reversing the sequences for backward search
    // backward search
    // reading the command line arguments
    // declaration and initialization of the fragment store
    // forward search
    // combining the contigs of the reference into one string set
    appendValue(text, fragStore.contigStore[i].seq);
    std::cerr << "Invalid number of arguments." << std::endl
}
<< "USAGE: mini_bowtie GENOME.fasta READS.fasta OUT.sam" << std::endl;
}
if (argc < 3) {
if (loadContigs(fragStore, argv[1])) return 1;
if (loadReads(fragStore, argv[2])) return 1;
clear(fmIndex);
clear(fmIndex);
StringSet<TString> text;
for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
fmIndex = TIndex(text);
TIndex fmIndex(text);
TIter it(fmIndex);
search();
search();
clear(it);
clear(it);
reverse(text);
reverse(fragStore.readSeqStore);
it = TIter(fmIndex);
FragmentStore<> fragStore;
return 0;
return 1;
}

Hint We make use of the FragmentStore. While we can access the pattern/reads as if using a StringSet, we need to
create a StringSet of the contigs, because in the FragmentStore the contigs are not stored in a StringSet.

Hint The correct order of the comments is:

// reading the command line arguments
// declaration and initialization of the fragment store
// combining the contigs of the reference into one string set
// forward search
// reversing the sequences for backward search
// backward search

Solution

// ==========================================================================
// mini_bowtie
// ==========================================================================

#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>

using namespace seqan;

void search() {}

int main(int argc, char *argv[]) {
  // type definitions
typedef String<Dna5> TString;
typedef StringSet<TString> TStringSet;
typedef Index<StringSet<TString>, FMIndex<> > TIndex;
typedef Iterator<TIndex, TopDown<ParentLinks<> > >::Type TIter;

// reading the command line arguments
if (argc < 3) {
    std::cerr << "Invalid number of arguments."
    << std::endl
    << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
    return 1;
}

// declaration and initialization of the fragment store
FragmentStore<> fragStore;
if (!loadContigs(fragStore, argv[1])) return 1;
if (!loadReads(fragStore, argv[2])) return 1;

StringSet<TString> text;
for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
    appendString(text, fragStore.contigStore[i].seq);

TIndex fmIndex(text);
TIter it(fmIndex);
search();
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);

fmIndex = TIndex(text);
it = TIter(fmIndex);
search();
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);
clear(fmIndex);
clear(it);

return 0;
}

Now that we have the backbone of our program we can start to implement the fundamental part, the search routine. The search function requires two input arguments, namely the iterator used to traverse the FM index of the reference sequence and the string set containing the reads.

The search function iterates over the reads and searches them in the already mentioned two phase fashion. In the first phase the right half of the pattern is searched exactly. The second phase is more involved and will be discussed after the second assignment.

Assignment 2

Type Application

Objective Expand the solution to the last assignment by implementing the backbone of the search routine. The backbone should consist of function definition, an outer loop traversing the pattern (using a standard iterator) and the first step of the search, namely the exact search of the right pattern half.

Hint The template function header could look like this:
template <typename TIter, typename TStringSet>
void search(TIter & it, TStringSet const & pattern)

Hint You can obtain the correct iterator type using the metafunction Iterator.

typedef typename Iterator<TStringSet const, Standard>::Type TPatternIter;

Solution

// ==========================================================================
// mini_bowtie
// ==========================================================================

#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>

using namespace seqan;

template <typename TIter, typename TStringSet>
void search(TIter & it, TStringSet const & pattern)
{
    typedef typename Iterator<TStringSet const, Standard>::Type TPatternIter;

    for (TPatternIter patternIt = begin(pattern, Standard()); patternIt != end(pattern, Standard()); ++patternIt)
    {
        unsigned startApproxSearch = length(value(patternIt)) / 2;
        goDown(it, infix(value(patternIt), 0, startApproxSearch - 1));
        goRoot(it);
    }
}

int main(int argc, char *argv[])
{
    typedef String<Dna5> TString;
    typedef StringSet<String<Dna5> > TStringSet;
    typedef Index<StringSet<TString>, FMIndex<>> TIndex;
    typedef Iterator<TIndex, TopDown<ParentLinks<>>> TIter;

    /*String<Dna> text = "ACGTACGT";
    Index<String<Dna>, FMIndex<>> index(text);
    Finder<Index<String<Dna>> > finder(index);*/

    find(finder, "AC");
    std::cerr << position("AC") << std::endl;
    
    // reading the command line arguments
    if (argc < 3) {
        std::cerr << "Invalid number of arguments."
                  << std::endl
                  << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
        return 1;
    }

    // declaration and initialization of the fragment store
    FragmentStore<> fragStore;
    if (!loadContigs(fragStore, argv[1])) return 1;
if (!loadReads(fragStore, argv[2])) return 1;

// combining the contigs of the reference into one string set
StringSet<TString> text;
for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
    appendValue(text, fragStore.contigStore[i].seq);

// forward search
reverse(text);
TIndex fmIndex(text);
TIter it(fmIndex);
search(it, fragStore.readSeqStore);
clear(fmIndex);
clear(it);

// reversing the sequences for backward search
reverse(text);
reverse(fragStore.readSeqStore);

// backward search
fmIndex = TIndex(text);
it = TIter(fmIndex);
search(it, fragStore.readSeqStore);
clear(fmIndex);
clear(it);

return 0;
}

At this point we need to implement the most critical part of the our program, which is the second step of the search phase. In our case this step works as follows:

Assume that we have already found a path in the trie representing the pattern from position $i$ to $(m - 1)$ with $m$ being the pattern length and $i < m/2$. Then we substitute the character of the pattern at position $i - 1$ with every character of the alphabet and search for the remaining characters exact. We repeat those two steps until we processed every character of the pattern.

The corresponding pseudo code could look like this:

```cpp
unsigned startApproxSearch = length(pattern) / 2;
for i = startApproxSearch to 0
{
    for all c in the alphabet
    {
        if (goDown(it, c))
        {
            if (goDown(it, pattern[0..i - 1]))
            {
                HIT
            }
            goBack to last start
        }
    }
    goDown correct character
}
```
Assignment 3

Type Application

Objective Include the pseudo code from above into the search function.

Hint Make a copy of the iterator before following the path of the substituted character. Doing so saves time and keeps
the code simple because you do not need to use goUp.

Hint goDown returns a boolean indicating if a path exists or not. In addition, you do not need to go through the steps
of the pseudo code if the second pattern half was not found!

Hint MinValue returns the lowest value of an alphabet, while ValueSize returns the number of different values of a
data type.

Solution

// ==========================================================================
// mini_bowtie
// ==========================================================================

#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>
using namespace seqan;

template<typename TIter, typename TStringSet>
void search(TIter & it, TStringSet const & pattern)
{
    typedef typename Iterator<TStringSet const, Standard>::Type TPatternIter;
    for (TPatternIter patternIt = begin(pattern, Standard()); patternIt != end(pattern, Standard()); ++patternIt)
    {
        unsigned startApproxSearch = length(value(patternIt)) / 2;
        if (goDown(it, infix(value(patternIt), 0, startApproxSearch - 1)))
        {
            for (unsigned i = startApproxSearch; ; ++i)
            {
                for (Dna5 c = MinValue<Dna>::VALUE; c < valueSize<Dna>(); ++c)
                {
                    TIter localIt = it;
                    if (goDown(localIt, c))
                    {
                        if (goDown(localIt, infix(value(patternIt), i + 1, length(value(patternIt))))
                        {
                            // HIT
                        }
                    }
                }
                if (i == length(value(patternIt)) - 1 || !goDown(it, getValue(patternIt)[i]))
                {
                    break;
                }
            }
        }
    }
    goRoot(it);
}
int main(int argc, char *argv[])
{
    typedef String<Dna5> TString;
    typedef StringSet<String<Dna5> > TStringSet;
    typedef Index<StringSet<TString>, FMIndex<> > TIndex;
    typedef Iterator<TIndex, TopDown<> >::Type TIter;

    // 0) Handle command line arguments.
    if (argc < 3) {
        std::cerr << "Invalid number of arguments." << std::endl
                  << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
        return 1;
    }

    // 1) Load contigs and reads.
    FragmentStore<> fragStore;
    if (!loadContigs(fragStore, argv[1])) return 1;
    if (!loadReads(fragStore, argv[2])) return 1;

    StringSet<TString> text;
    for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
        appendValue(text, fragStore.contigStore[i].seq);
    reverse(text);
    TIndex fmIndex(text);
    TIter it(fmIndex);
    search(it, fragStore.readSeqStore);
    clear(fmIndex);
    clear(it);

    reverse(text);
    reverse(fragStore.readSeqStore);

    fmIndex = TIndex(text);
    it = TIter(fmIndex);
    search(it, fragStore.readSeqStore);
    clear(fmIndex);
    clear(it);

    return 0;
}

So far so good. But there is a slight mistake. While substituting we also substitute the character in the pattern with itself. Therefore we find locations of exact matches multiple times.

**Assignment 4**

**Type** Application

**Objective** Adjust the code to cope with the problem mentioned above.

**Solution**

// ==========================================================================
// mini_bowtie
// ==========================================================================

380 Chapter 2. API Documentation
```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>
using namespace seqan;

template <typename TIter, typename TStringSet>
void search(TIter & it, TStringSet const & pattern)
{
    typedef typename Iterator<TStringSet const, Standard>::Type TPatternIter;

    for (TPatternIter patternIt = begin(pattern, Standard()); patternIt != end(pattern, Standard()); ++patternIt)
    {
        // exact search on pattern half
        unsigned startApproxSearch = length(value(patternIt)) / 2;
        if (goDown(it, infix(value(patternIt), 0, startApproxSearch - 1)))
        {
            for (unsigned i = startApproxSearch; ; ++i)
            {
                Dna character = getValue(patternIt)[i];
                for (Dna5 c = MinValue<Dna>::VALUE; c < valueSize<Dna>(); ++c)
                {
                    if (c != character)
                    {
                        TIter localIt = it;
                        if (goDown(localIt, c))
                        {
                            if (goDown(localIt, infix(value(patternIt), i + 1, length(value(patternIt))))
                            {
                                // HIT
                            }
                        }
                        else if (i == length(value(patternIt)) - 1)
                        {
                            // HIT
                            break;
                        }
                    }
                    else
                    {
                        goRoot(it);
                        break;
                    }
                }
            }
        }
    }
}

int main(int argc, char *argv[])
{
    typedef String<Dna5> TString;
    typedef StringSet<String<Dna5> > TStringSet;
    typedef Index<StringSet<TString>, FMIndex<> > TIndex;
    typedef Iterator<TIndex, TopDown<> >::Type TIter;

    // 0) Handle command line arguments.
    
    // ...
if (argc < 3) {
    std::cerr << "Invalid number of arguments." << std::endl
    << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
    return 1;
}
// 1) Load contigs and reads.
FragmentStore<> fragStore;
if (!loadContigs(fragStore, argv[1])) return 1;
if (!loadReads(fragStore, argv[2])) return 1;

StringSet<TString> text;
for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
    appendValue(text, fragStore.contigStore[i].seq);

reverse(text);
TIndex fmIndex(text);
TIter it(fmIndex);
search(it, fragStore.readSeqStore);
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);

fmIndex = TIndex(text);
it = TIter(fmIndex);
search(it, fragStore.readSeqStore);
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);

std::ofstream samFile(argv[3], std::ios_base::out);
write(samFile, fragStore, Sam());
return 0;
}

So this is already the fundamental part of our program. What’s left to do is to write the result into a SAM file.
In order to do so, we make use of the FragmentStore. Everything we need to do is to fill the alignedReadStore
which is a member of the FragmentStore. This is very easy, because we only need to append a new value of type
AlignedReadStoreElement specifying the match id, the pattern id, the id of the contig, as well as the begin and end
position of the match in the reference.

An addMatchToStore function could look like this:

template <typename TStore, typename TIter, typename TPatternIt>
void addMatchToStore(TStore & fragStore, TPatternIt const & patternIt, TIter const & localIt)
{
    typedef FragmentStore<>::TAlignedReadStore TAlignedReadStore;
    typedef Value<TAlignedReadStore>::Type TAlignedRead;

    for (unsigned num = 0; num < countOccurrences(localIt); ++num)
    {
        unsigned pos = getOccurrences(localIt)[num].i2;
        TAlignedRead match(length(fragStore.alignedReadStore), position(patternIt), getOccurrences(localIt)[num].i1, pos, pos + length(value(patternIt)));
        appendValue(fragStore.alignedReadStore, match);
One could think we are done now. Unfortunately we are not. There are two problems. Recall, that in the second search phase we reverse the text and pattern, therefore we messed up start and end positions in the original reference. Furthermore we found exact matches twice, once in the forward index and once in the reversed index.

**Assignment 5**

**Type** Application

**Objective** Include the following two lines into your code:

```cpp
struct ForwardTag {}
struct ReverseTag {}
```

and write a second ``addMatchToStore`` function that is called when we search in the reversed reference. In addition, make all necessary changes to the code such that exact matches are only added once.

**Hint** The metafunction IsSameType can be used to determine whether two types are equal or not.

**Solution**

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>
using namespace seqan;

struct ForwardTag {};
struct ReverseTag {};

template <typename TStore, typename TIter, typename TPatternIt>
void addMatchToStore(TStore & fragStore, TPatternIt const & patternIt, TIter const & localIt, ForwardTag)
{
    typedef FragmentStore<>::TAlignedReadStore TAlignedReadStore;
    typedef Value<TAlignedReadStore>::Type TAlignedRead;
    for (unsigned num = 0; num < countOccurrences(localIt); ++num)
    {
        unsigned pos = getOccurrences(localIt)[num].i2;
        TAlignedRead match(length(fragStore.alignedReadStore), position(patternIt), getOccurrences(localIt)[num].i1, pos, pos + length(value(patternIt)));
        appendValue(fragStore.alignedReadStore, match);
    }
}

template <typename TStore, typename TIter, typename TPatternIt>
void addMatchToStore(TStore & fragStore, TPatternIt const & patternIt, TIter const & localIt, ReverseTag)
{
    typedef FragmentStore<>::TAlignedReadStore TAlignedReadStore;
    typedef Value<TAlignedReadStore>::Type TAlignedRead;
    for (unsigned num = 0; num < countOccurrences(localIt); ++num)
    {
        unsigned pos = getOccurrences(localIt)[num].i2;
        TAlignedRead match(length(fragStore.alignedReadStore), position(patternIt), getOccurrences(localIt)[num].i1, pos, pos + length(value(patternIt)));
        appendValue(fragStore.alignedReadStore, match);
    }
}
```
typedef Value<TAlignedReadStore>::Type TAlignedRead;

for (unsigned num = 0; num < countOccurrences(localIt); ++num)
{
    unsigned contigLength = length(fragStore.contigStore[getOccurrences(localIt)[num].i1].seq);
    unsigned pos = contigLength - getOccurrences(localIt)[num].i2 - length(value(patternIt));
    TAlignedRead match(length(fragStore.alignedReadStore), position(patternIt), getOccurrences(localIt)[num].i1, pos, pos + length(value(patternIt)));
    appendValue(fragStore.alignedReadStore, match);
}

template<
typename TIter, typename TStringSet, typename TStore, typename DirectionTag>
void search(TIter & it, TStringSet const & pattern, TStore & fragStore, DirectionTag /*tag*/)
{
    typedef typename Iterator<TStringSet const, Standard>::Type TPatternIter;

    for (TPatternIter patternIt = begin(pattern, Standard()); patternIt != end(pattern, Standard()); ++patternIt)
    {
        // exact search on pattern half
        unsigned startApproxSearch = length(value(patternIt)) / 2;
        if (goDown(it, infix(value(patternIt), 0, startApproxSearch - 1)))
        {
            for (unsigned i = startApproxSearch; ; ++i)
            {
                Dna character = getValue(patternIt)[i];
                for (Dna5 c = MinValue<Dna>::VALUE; c < valueSize<Dna>(); ++c)
                {
                    if (c != character)
                    {
                        TIter localIt = it;
                        if (goDown(localIt, c))
                        {
                            if (goDown(localIt, infix(value(patternIt), i + 1, length(value(patternIt))))
                            {
                                addMatchToStore(fragStore, patternIt, localIt, DirectionTag());
                            }
                        }
                    }
                }
                if (!goDown(it, character))
                    break;
                else if (i == length(value(patternIt)) - 1)
                {
                    if (IsSameType<DirectionTag, ForwardTag>::VALUE)
                        addMatchToStore(fragStore, patternIt, it, DirectionTag());
                    break;
                }
            }
            goRoot(it);
        }
    }
}

int main(int argc, char *argv[])
{
    typedef String<Dna5> TString;
    typedef StringSet<String<Dna5> > TStringSet;
```cpp
typedef Index<StringSet<TString>, FMIndex<> > TIndex;
typedef Iterator<TIndex, TopDown<> >::Type TIter;

// 0) Handle command line arguments.
if (argc < 3) {
    std::cerr << "Invalid number of arguments." << std::endl
    << "USAGE: minimapper GENOME.fasta READS.fasta OUT.sam" << std::endl;
    return 1;
}

// 1) Load contigs and reads.
FragmentStore<> fragStore;
if (!loadContigs(fragStore, argv[1])) return 1;
if (!loadReads(fragStore, argv[2])) return 1;

StringSet<TString> text;
for (unsigned i = 0; i < length(fragStore.contigStore); ++i)
    appendValue(text, fragStore.contigStore[i].seq);

reverse(text);
TIndex fmIndex(text);
TIter it(fmIndex);
search(it, fragStore.readSeqStore, fragStore, ForwardTag());
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);

fmIndex = TIndex(text);
it = TIter(fmIndex);
search(it, fragStore.readSeqStore, fragStore, ReverseTag());
clear(fmIndex);
clear(it);

reverse(text);
reverse(fragStore.readSeqStore);
std::ofstream samFile(argv[3], std::ios_base::out);
write(samFile, fragStore, Sam());
return 0;
}
```

Done? Not quite.

We need to copy the following four lines of code into our code in order to write the correct result in the SAM file. Calling the reverse function is necessary because an alignment must be computed for every match to be written into the SAM file.

```cpp
reverse(text);
reverse(fragStore.readSeqStore);
std::ofstream samFile(argv[3], std::ios_base::out);
write(samFile, fragStore, Sam());
```
2.1.38 Journaled Set

Learning Objective  This tutorial introduces you to the new data structures Journaled Set and Journaled String. You will learn how to use them and how to exploit these data structures for an efficient analysis, while implementing a native online search.

Difficulty  Advanced

Duration  2 h

Prerequisites  Sequences, String Sets, Iterators

A typical task in bioinformatics is to find patterns in biological sequences e.g. transcription factors, or to examine different biological traits and the effects of modifications on such traits. With the current advances in sequencing technologies, sequences of whole populations have been made available. But the time for searching in all these sequences is proportional to the number of sequences searched. That’s why it is important to find novel strategies to cope with the deluge of sequencing data. Since, many biological problems often involve the analysis of sequences of the same species, one effective strategy would be to exploit the similarities of such sequences.

For this special purpose we provide two data structures that are designed to improve the algorithmic tasks. The first one is the JournaledString and the second is the JournaledSet.

In this tutorial, we will introduce you to both data structures and implement a simple online search step by step.

Journaled String

The JournaledString data structure behaves like a normal String in SeqAn, except that it is composed of two data structures.

1. The first data structure is a Holder which stores a sequence.

2. The second data structure stores modifications that are made to this particular sequence using a journal (see Journaling Filesystems for more information). This journal contains a list of deletions and insertions. The inserted characters are stored in an additional insertion buffer.

The advantage of this data structure lies in representing a String as a “patch” to another String. The journaled data structure can be modified without loosing the original context. We want to show you how to work with these data structures so you can build your own algorithms based on this. For this reason we keep the applicational background simple and implement an native online-search algorithm by which we examine different properties of the data structures.
Before we start implementing our online search algorithm, we show you how to work with the Journaled String to learn the basic principles. To get access to the Journaled String implementation you have to include the `<seqan/sequence_journaled.h>` header file. Note that you will need the `<seqan/file.h>` too in order to print the sequences.

```
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence_journaled.h>

using namespace seqan;

int main()
{
    In the next step we define the Journaled String type. A Journaled String is a specialization of the String class and is defined as `String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec>>`. The specialization takes two parameters: (1) `TValue` defines the alphabet type used for the Journaled String and (2) `Journaled<>` selects the Journaled String specialization of the String class.

`Journaled<>` is further specialized with

- `THostSpec` selects the specialization of the underlying host sequence (`Alloc<>` for `[dox:AllocString Alloc String`),
- `TJournalSpec` selects the used implementation to manage the journaled differences (here: `SortedArray`), and
- `TBufferSpec` selects the used specialization for the internally managed insertion buffer (here: `Alloc<>` as well).

In our scenario we use a `char` alphabet and `[dox:AllocString Alloc String` for the host string and the insertion buffer. Additionally, we use a `Sorted Array` as the model to manage the recorded differences.

We use the metafunction `Host` to get the type of the underlying host string used for the Journaled String.

```
typedef String<char, Journaled<Alloc<>>, SortedArray, Alloc<> >> TJournaledString;
typedef Host<TJournaledString>::Type THost;
```

Now we can define the variables holding data structures. First, we construct our host sequence and after that we construct the Journaled String. Then, we set the host sequence using the function `setHost`. Afterwards, we examine the data structure in more detail and print the host sequence the constructed journaled sequence and the nodes of it.

```
String<char> hostStr = "thisisahostsequence";
TJournaledString journalStr;
setHost(journalStr, hostStr);

std::cout << "After creating the Journaled String:" << std::endl;
std::cout << "Host: " << host(journalStr) << std::endl;
std::cout << "Journal: " << journalStr << std::endl;
std::cout << "Nodes: " << journalStr._journalEntries << std::endl;
```

**Tip:** The Journal

A node in the Journaled String represents either a part of the host sequence or a part of the insertion buffer. The type of a node is distinguished by the member variable `segmentSource` and can be of value `SOURCE_ORIGINAL` to refere to a part in the host or `SOURCE_PATCH` to refere to a part in the insertion buffer. A node further consists of three variables which specify the virtual position, the physical position and the length of this part. The virtual position gives the relative position of the Journaled String after all modifications before this position have been “virtually”
applied. The physical position gives the absolute position where this part of the journal maps to either the host sequence or the insertion buffer.

This is followed by modifying our Journaled String. We insert the string "modified" at position 7 and delete the suffix "sequence" at position 19. Note that position 19 refers to the string after the insertion of "modified" at position 7. Again we print the host, the journaled sequence and the nodes that represent the modifications to see how our changes affect the host and the journaled sequence.

```cpp
insert(journalStr, 7, "modified");
erase(journalStr, 19, 27);
```

```cpp
std::cout << "After modifying the Journaled String:" << std::endl;
std::cout << "Host: " << host(journalStr) << std::endl;
std::cout << "Journal: " << journalStr << std::endl;
std::cout << "Nodes: " << journalStr._journalEntries << std::endl;
```

All of this is followed by calling flatten on our journeld string. This call applies all journaled changes to the host sequence. Again we print the sequences to see the effects.

```cpp
flatten(journalStr);
```

```cpp
std::cout << "After flatten the Journaled String:" << std::endl;
std::cout << "Host: " << host(journalStr) << std::endl;
std::cout << "Journal: " << journalStr << std::endl;
std::cout << "Nodes: " << journalStr._journalEntries << std::endl;
```

Here is the output of our small program.

After creating the Journaled String:
Host: thisisahostsequence
Journal: thisisahostsequence
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0, length=19})

After modifying the Journaled String:
Host: thisisahostsequence
Journal: thisisamodifiedhost
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0, length=7}, {segmentSource=1, virtualPosition=7, physicalPosition=0, length=8})

After flatten the Journaled String:
Host: thisisamodifiedhost
Journal: thisisamodifiedhost
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0, length=19})

**Important:** Be careful when using the flatten function as it modifies the underlying host sequence. This might affect other journaled sequences that share the same host sequence. This becomes important especially when working with Journal Sets where a whole set of sequences is journaled to the same reference.

**Journaled Set**

The JournaledSet is a specialization of the StringSet which can be used exactly as such but also provides some additional functions optimized to work with JournaledStrings. The general interface is equal to the interface of the StringSet. But it also provides some interfaces specialized for the use of Journaled Strings. One of these interfaces is the join function which journales a contained Journaled String to the previously set global reference. The following
code snippet demonstrates the usage of the Journal Set and how to join a sequence to the previously set reference sequence.

As usual we include the necessary headers. We need the header `<seqan/journal_set.h>` to get access to the Journal Set. Then we define a type for journaled sequences. After that we define our Journal Set. The Journal Set is a specialization of the Owner concept of StringSets and is defined as `StringSet<TJournalString, Owner<JournaledSet>>`.

```cpp
#include <iostream>
#include <seqan/file.h>
#include <seqan/journaled_set.h>

using namespace seqan;

int main()
{

typedef String<char, Journaled<Alloc<>, SortedArray, Alloc<> > > TJournalString;
typedef Host<TJournalString>::Type THost;
typedef StringSet<TJournalString, Owner<JournaledSet> > TJournaledSet;

TJournaledSet journaledSet;

THost reference = "DPKKPRGKMSSYAFFVQTSSREHKKHPDASVNFSEFSKCCSERWKTMSAEEKGFEDMAKADKARYEREMKTYIPPKGE";
THost seq0 = "DPKKPRGKMSSYAFFVQTSSREHKKHPDASVNFSEFSKCCSERWKTMSAEEKGFEDMAKADKARYEREMKTYIPPKGE";
THost seq1 = "DPKKPRGKMSSYAFFVQTSSREHKKHPDASVNFSEFSKCCSERWKTMSAEEKGFEDMAKADKARYEREMKTYIPPKGE";
THost seq2 = "DPKKPRGKMSSYAFFVQTSSREHKKHPDASVNFSEFSKCCSERWKTMSAEEKGFEDMAKADKARYEREMKTYIPPKGE";

In the subsequent steps we want to set a reference sequence to the Journal Set and add some sequences to it. We can set the reference sequence by using the function `setHost`. This function stores only a pointer to the given sequence. In some cases it might be necessary to copy the reference sequence instead. For this purpose you can use the function `createHost`.

```cpp
setGlobalReference(journaledSet, reference);
appendValue(journaledSet, TJournalString(seq0));
appendValue(journaledSet, TJournalString(seq1));
appendValue(journaledSet, TJournalString(seq2));
```

Just adding sequences to the Journal Set does not automatically journal them to the global reference sequence of the set. One can explicitly trigger this process using the function `join`. This function takes as parameters the Journal Set and the position of the contained Journaled String which is to be journaled to the reference sequence. Thus, the programmer is free in the decision which sequence has to be journaled and which not. Furthermore, we can use the `JoinConfig` object to specify the method that shall be used for the journaling process.

```cpp
join(journaledSet, 0, JoinConfig<GlobalAlign<JournaledManhatten> >()); // Simply inserts the
join(journaledSet, 1, JoinConfig<GlobalAlign<JournaledCompact> >()); // Uses default scoring
JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;
setScoringScheme(joinConfig, Score<int, BiAffine>(0,-1,-1)); // Note the mismatch score is for
join(journaledSet, 2, joinConfig); // Compute journal using Levenshstein distance.
```

```cpp
std::cout << "Reference: " << globalReference(journaledSet) << std::endl;
for(unsigned i = 0; i < length(journaledSet); ++i)
    std::cout << "Journaled Sequence " << i << ": " " << value(journaledSet,i) << std::endl;
return 0;
}
```

**Tip:** Configuration of the Join Methods

The `JoinConfig` object differentiates two methods in general and each method further differs in the used strategy.
The two methods are the GlobalAlign and the GlobalChain method. They differ in the approach of computing the alignment which is necessary to construct the journal. The first method uses a global alignment algorithm while the second one uses an anchor approach in which first exact seeds are found using a q-gram index and after that the optimal chain between the identified anchors is computed. For each method the user can specify a different strategy. The first strategy is triggered by using JournaledManhatten. This means for the the GlobalAlign method, that the complete sequence is inserted and the complete reference is deleted, while for the GlobalChain methods this means that the gaps between the anchors are connected through the Manhattan distance. The second strategy is specified using the JournaledCompact tag. It computes the most compact form of a journal by means of memory requirements.

Here is the output of the program.

Reference: DPKKPRGMSYAFFVQTSREEHKKHPDASVNFSEFSKCSERWKTMSAKEKGKFEDMAKADKARYEREMKTYIPFKGE
Journaled Sequence 0: DPKKPRGKMVSPPAFFVQTSREEHKKHPDASVFSKCSERWKTMSAKEKGKFEDMAKADKARYEREMKTYIPFKGE
Journaled Sequence 1: DPHHPKPGRKMVNSPPAFFVQTSREEHKPDASVFSKCSERMPHNHTMSAKEKGKFEDMAKADKARYEREMKTYIPFKGE
Journaled Sequence 2: DPKKPRGMSYAFFVQTSREEHKKHPKCDFSKCSERWKTMSAKEKGKFEDARYEREMKTYIPFKGE

Implementing an Online-Search

Now we have all foundations laid down to implement the online-search algorithm. Let us begin with the first assignment where we read in some sequences and use the currently learned things about the Journal Set.

Assignment 1

Type Review, Application

Objective Download the fasta file sequences.fasta which contains some DNA sequences. Write a method called loadAndJoin that gets a Journal Set and a stream file pointing to the downloaded fasta file. The method reads in the sequences one after another using SeqAn’s RecordReader. The first read sequences is set to the reference sequence. All following sequences are first appended to the StringSet and afterwards joined to the StringSet using a global alignment strategy and the most compact form.

Hints

You can start using the following code snippet. Replace the path of the iostream such that it points to your path and fill in the missing parts A, B and C in the function loadAndJoin (Altogether, you will need 4 lines of code.).

```cpp
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>

using namespace seqan;

template <typename TString, typename TStream, typename TSpec>
inline int loadAndJoin(StringSet<TString, Owner<JournaledSet> > & /*journalSet*/,
                        TStream & stream,
                        JoinConfig<TSpec> const & /*joinConfig*/
) {
    typedef typename Host<TString>::Type THost;
    // Define the RecordReader.
    RecordReader<std::ifstream, SinglePass<> > reader(stream);

    // [A] Ensure the Journal Set is not occupied by other sequences.

    // Construct the temporary buffers for the read id and sequence.
```
String<char> tempSeqId;
THost tempSeq;

// No sequences in the fasta file!
if (atEnd(reader)) {
    std::cerr << "Empty FASTA file." << std::endl;
    return -1;
}

// First read sequence for reference sequence.
if (readRecord(tempSeqId, tempSeq, reader, Fasta()) != 0) {
    std::cerr << "ERROR reading FASTA." << std::endl;
    return 1;
}

// (B) Set the reference sequence to the Journal Set

// Read remaining sequences.
while (!atEnd(reader)) {
    if (readRecord(tempSeqId, tempSeq, reader, Fasta()) != 0) {
        std::cerr << "ERROR reading FASTA." << std::endl;
        return 1;
    }

    // (C) Append and join the current read sequence.
}
return 0;

int main() {

    // Definition of the used types.
    typedef String<Dna,Alloc<> > TSequence;
    typedef String<Dna,Journaled<Alloc<>,SortedArray,Alloc<> > > TJournal;
    typedef StringSet<TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/path/to/your/fasta/file/sequences.fasta";
    std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
    if (!databaseFile.good()) {
        std::cerr << "Cannot open file " << seqDatabasePath << "!" << std::endl;
    }

    // Reading each sequence and journal them.
    // (D) Construct Journaled Set and call loadAndJoin

    databaseFile.close();
    return 0;
}

Solution
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>
using namespace seqan;

template <typename TString, typename TStream, typename TSpec>
inline int loadAndJoin(StringSet<TString, Owner<JournaledSet>> & journalSet,
                        TStream & stream,
                        JoinConfig<TSpec> const & joinConfig)
{
    typedef typename Host<TString>::Type THost;
    // Define the RecordReader.
    RecordReader<std::ifstream, SinglePass<> > reader(stream);
    // [A]
    clear(journalSet);

    // Construct the temporary buffers for the read id and sequence.
    String<char> tempSeqId;
    THost sequence;

    // No sequences in the fasta file!
    if (atEnd(reader))
    {
        std::cerr << "Empty FASTA file." << std::endl;
        return -1;
    }
    // First read sequence for reference sequence.
    if (readRecord(tempSeqId, sequence, reader, Fasta()) != 0)
    {
        std::cerr << "ERROR reading FASTA." << std::endl;
        return 1;
    }
    // [B]
    createGlobalReference(journalSet, sequence); // When using create we copy the reference instead of storing a pointer.

    // Read remaining sequences.
    while (!atEnd(reader))
    {
        if (readRecord(tempSeqId, sequence, reader, Fasta()) != 0)
        {
            std::cerr << "ERROR reading FASTA." << std::endl;
            return 1;
        }
        // [C]
        appendValue(journalSet, TString(sequence)); // First we append the sequence to the set.
        join(journalSet, length(journalSet) - 1, joinConfig); // Second we join it to the set.
    }
    return 0;
}

int main()
{
    // Definition of the used types.
    typedef String<Dna,Alloc<> > TSequence;
    typedef String<Dna,Journaled<Alloc<> ,SortedListArray,Alloc<> > > TJournal;
    typedef StringSet< TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/Users/rahn_r/Downloads/sequences.fasta";
std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
if(!databaseFile.good())
{
    std::cerr << "Cannot open file " << seqDatabasePath << "!" << std::endl;
}

// Reading each sequence and journal them.
TJournaledSet journalSet;
JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;
loadAndJoin(journalSet, databaseFile, joinConfig);
databaseFile.close();

return 0;
}

Now we have loaded and journaled our sequences and we use the minimal possible memory requirements for our sequences. Let’s continue and implement the exact online-search on the Journal Set. For this purpose we write a function called searchPattern which takes a StringSet of String<int> which we use to store each hit for each sequence in the Journal Set including the reference sequence. First we have to check whether the reference sequence is set. If not we abort the search since we cannot guarantee a correct search when the reference is not set. We also have to clear our hitSet to ensure there remain no phantom hits from previous searches. Then we resize it to the number of contained Journaled Strings plus an additional space for the global reference sequence.

```cpp
template<typename TString, typename TPattern>
void searchPattern(StringSet<String<int>> & hitSet,
    StringSet<TString, Owner<JournaledSet>> const & journalSet,
    TPattern const & pattern)
{
    typedef typename Host<TString>::Type THost;

    // Check for valid initial state.
    if (empty(globalReference(journalSet)))
    {
        std::cout << "No reference set. Aborted search!" << std::endl;
        return;
    }

    // Reset the hitSet to avoid phantom hits coming from a previous search.
    clear(hitSet);
    resize(hitSet, length(journalSet) + 1);

    Before we can search for the pattern in the Journaled Strings we have to find all occurrences in the reference sequence. Therefore we call the function findPatternInReference which takes a String<int> which we use to store the hits, the global reference and the pattern.

    // Access the reference sequence.
    THost & globalRef = globalReference(journalSet);
    // Search for pattern in the reference sequence.
    findPatternInReference(hitSet[0], globalRef, pattern);

    After that we implement the body to search for occurrences in the Journaled Strings. Therefore we use for-loop to iterate over all contained sequences and call for each sequence the function findPatternInJournalString. The function gets as parameters the corresponding String<int> from the hitSet, the journaled sequence the pattern and the identified hits in the reference sequence.

    // Search for pattern in the journaled sequences.
    for (unsigned i = 0; i < length(journalSet); ++i)
    {
        findPatternInJournalString(hitSet[i+1], journalSet[i], pattern, hitSet[0]);
    }
```
So far our program won’t compile. We have to first implement the both functions findPatternInReference and findPatternInJournalString.

Assignment 2

Type Application

Objective Implement the function findPatternInReference using a brute force pattern search algorithm. Store all found hits in the passed hits variable. Print all occurrences in the end of the main function.

Hints

```cpp
template <typename TString, typename TPattern>
void findPatternInReference(String<int> & hits,
    TString const & reference,
    TPattern const & pattern)
{
    // [A] Check whether pattern fits into the sequence.
    if (length(pattern) > length(reference))
        return;

    // [B] Iterate over all positions at which the pattern might occur.
    for (unsigned pos = 0; pos < length(reference) - length(pattern) + 1; ++pos)
    {
        bool isHit = true;
        // [C] Evaluate all positions of the pattern until you find a mismatch or you have found a hit.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern)
        {
            if (pattern[posPattern] != reference[posPattern + pos])
            {
                isHit = false;
                break;
            }
        }
        // [D] Report begin position at which pattern matches the sequence.
        if (isHit)
            appendValue(hits, pos);
    }
}
```

Solution Here is the solution for this function. Click more... below, to see a complete solution of everything we have done so far.

```cpp
template <typename TString, typename TPattern>
void findPatternInReference(String<int> & hits,
    TString const & reference,
    TPattern const & pattern)
{
    // [A] Check whether pattern fits into the sequence.
    if (length(pattern) > length(reference))
        return;

    // [B] Iterate over all positions at which the pattern might occur.
    for (unsigned pos = 0; pos < length(reference) - length(pattern) + 1; ++pos)
    {
        bool isHit = true;
        // [C] Evaluate all positions of the pattern until you find a mismatch or you have found a hit.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern)
        {
            if (pattern[posPattern] != reference[posPattern + pos])
            {
                isHit = false;
                break;
            }
        }
        // [D] Report begin position at which pattern matches the sequence.
        if (isHit)
            appendValue(hits, pos);
    }
}
```
Include the necessary headers.

```cpp
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>

using namespace seqan;
```

Implementation of the `findPatternInReference` function.

```cpp
template <typename TString, typename TPattern>
void findPatternInReference(String<int> & hits,
                         TString const & reference,
                         TPattern const & pattern)
{
    // [A] Check whether pattern fits into the sequence.
    if (length(pattern) > length(reference))
        return;
    // [B] Iterate over all positions at which the pattern might occur.
    for (unsigned pos = 0; pos < length(reference) - length(pattern) + 1; ++pos)
    {
        bool isHit = true;
        // [C] Evaluate all positions of the pattern until you find a mismatch or you have found a hit.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern)
        {
            if (pattern[posPattern] != reference[posPosPattern + pos])
            {
                isHit = false;
                break;
            }
        }
        // [D] Report begin position at which pattern matches the sequence.
        if (isHit)
            appendValue(hits, pos);
    }
}
```

Implementation of the `searchPattern` function. Note that we haven’t implemented the function `findPatternInJournalString` yet.

```cpp
template <typename TString, typename TPattern>
void searchPattern(StringSet<String<int>> & hitSet,
                   StringSet<TString, Owner<JournaledSet>> const & journalSet,
                   TPattern const & pattern)
{
    typedef typename Host<TString>::Type THost;
    // Check for valid initial state.
    if (empty(globalReference(journalSet)))
    {
        std::cout << "No reference set. Aborted search!" << std::endl;
        return;
    }
    // Reset the hitSet to avoid phantom hits coming from a previous search.
    clear(hitSet);
    resize(hitSet, length(journalSet) + 1);
    // Access the reference sequence.
```
THost & globalRef = globalReference(journalSet);
// Search for pattern in the reference sequence.
findPatternInReference(hitSet[0], globalRef, pattern);

// Search for pattern in the journaled sequences.
for (unsigned i = 0; i < length(journalSet); ++i)
{
    // findPatternInJournalString(hitSet[i+1], journalSet[i], pattern, hitSet[0]);
}

Implementation of the *loadAndJoin* function.

```cpp
template <typename TString, typename TStream, typename TSpec>
inline int loadAndJoin(StringSet<TString, Owner<JournaledSet>> & journalSet, 
                        TStream & stream, JoinConfig<TSpec> const & joinConfig)
{
    typedef typename Host<TString>::Type THost;
    RecordReader<std::ifstream, SinglePass<> > reader(stream);
    clear(journalSet);
    String<char> tempSeqId;
    THost sequence;

    // No sequences in the fasta file!
    if (atEnd(reader))
    {
        std::cerr << "Empty FASTA file." << std::endl;
        return -1;
    }

    // First read sequence for reference sequence.
    if (readRecord(tempSeqId, sequence, reader, Fasta()) != 0)
    {
        std::cerr << "ERROR reading FASTA." << std::endl;
        return 1;
    }

    // We have to create the global reference sequence otherwise we loose the information after
    createGlobalReference(journalSet, sequence);

    // If there are more
    while (!atEnd(reader))
    {
        if (readRecord(tempSeqId, sequence, reader, Fasta()) != 0)
        {
            std::cerr << "ERROR reading FASTA." << std::endl;
            return 1;
        }
        appendValue(journalSet, TString(sequence));
        join(journalSet, length(journalSet) - 1, joinConfig);
    }
    return 0;
}
```

Implementation of the *main* function.
```cpp
int main()
{
    // Definition of the used types.
    typedef String<Dna, Alloc<> > TSequence;
    typedef String<Dna, Journaled<Alloc<>>, SortedArray, Alloc<> > TJournal;
    typedef StringSet< TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/Users/rahn_r/Downloads/sequences.fasta";
    std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
    if(!databaseFile.good())
    {
        std::cerr << "Cannot open file <" << seqDatabasePath << ">!" << std::endl;
    }

    // Reading each sequence and journal them.
    TJournaledSet journalSet;
    JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;
    loadAndJoin(journalSet, databaseFile, joinConfig);
    databaseFile.close();

    // Define a pattern and start search.
    StringSet<String<int> > hitSet;
    TSequence pattern = "GTGGT";
    std::cout << "Search for: " << pattern << ":

    searchPattern(hitSet, journalSet, pattern);

    Printing the hits of the reference sequence.
    if (empty(hitSet[0]))
    {
        std::cout << "No hit in reference " << std::endl;
    }
    else
    {
        std::cout << "Hit in reference " << " at ";
        for (unsigned j = 0; j < length(hitSet[0]); ++j)
        {
            std::cout << hitSet[0][j] << ": " << infix(globalReference(journalSet), hitSet[0][j]) << "
        }
        std::cout << std::endl;
    }

    return 0;
}
```

And here is the result.
```
Search for: GTGGT:
Hit in reference at 311: GTGGT 644: GTGGT
```

We know can search for all occurrences of a pattern in the reference sequence. Now we can try to find all occurrences in the journaled sequences. Therefore we implement the function `findPatternInJournalString`. Our function gets the variable `hitsTarget` which stores the hits found in the JournaledString. It gets the search text and the pattern and finally the hits detected in the reference sequence. Instead of searching each position in the Journaled String, we only search in areas that are new to the search. This involves all inserted parts and all parts where the pattern crosses a border to another node. So instead of iterating over each position we iterate over the nodes of the Journaled String. To do so we have to determine the type of the data structure that is used by the Journaled String to manage the nodes. We can use the metafunction `JournalType` for this task. Afterwards we define an Iterator over the
so called TJournalEntries data structure.

Again we check first whether the pattern fits into our sequence.

```cpp
template <typename TValue, typename THostSpec, typename TJournalSpec, typename TBufferSpec, typename
          TPattern>
void findPatternInJournalString(String<int> & hitTarget,
                              String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const & journal,
                              TPattern const & pattern,
                              String<int> const & refHits)
{
  typedef String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const TJournal;
  typedef typename JournalType<TJournal>::Type TJournalEntries;
  typedef typename Iterator<TJournalEntries>::Type TJournalEntriesIterator;

  if (length(pattern) > length(journal))
    return;

  TJournalEntriesIterator it = begin(journal._journalEntries);
  TJournalEntriesIterator itEnd = findInJournalEntries(journal._journalEntries, length(journal) - length(pattern) + 1) + 1;

  while (it != itEnd)
  {
    if (it->segmentSource == SOURCE_ORIGINAL)
      // Find a possible hit in the current source vertex.
      _findInOriginalNode(hitTarget, it, pattern, refHits);
    else if (it->segmentSource == SOURCE_PATCH)
      // Search for pattern within the patch node.
      _findInPatchNode(hitTarget, it, journal, pattern);
    // Scan the border for a possible match.
    _searchAtBorder(hitTarget, it, journal, pattern);
    ++it;
  }
}
```

We then iterate over all nodes beginning from the first until we have reached the node in which the pattern reaches the end of the Journaled Sequence. The function findInJournalEntries helps us to find the corresponding node. We increment the position of the iterator by one such that it points behind the last element which is included by the search.

```cpp
TJournalEntriesIterator it = begin(journal._journalEntries);
TJournalEntriesIterator itEnd = findInJournalEntries(journal._journalEntries, length(journal) - length(pattern) + 1);
```

Now we search in each node until we have reached the end. For each node we first check the type of the journaled operation. If we are in an “original” node we call the function _findInOriginalNode. If we are in a “patch” node we call the function _findInPatchNode and in the end we call the function _searchAtBorder which is called for each node type and scans all possible hits at the border of a node.

```cpp
while (it != itEnd)
{
  if (it->segmentSource == SOURCE_ORIGINAL)
    // Find a possible hit in the current source vertex.
    _findInOriginalNode(hitTarget, it, pattern, refHits);
  else if (it->segmentSource == SOURCE_PATCH)
    // Search for pattern within the patch node.
    _findInPatchNode(hitTarget, it, journal, pattern);
  // Scan the border for a possible match.
  _searchAtBorder(hitTarget, it, journal, pattern);
  ++it;
}
```

Let us begin with the implementation of the function _findInOriginalNode. In this function we exploit the journaling concept such that we can speed up the search algorithm from \(O(p \cdot n)\) to \(O(\log_2(k))\), where \(p\) is the length of the pattern, \(n\) is the length of the search text, and \(k\) is the number of hits identified in the reference sequence. We need at most \(\log_2(k)\) comparisons to find the first hit which occurred in the reference sequence that also occurs in the current original node.

Assignment 3

Type Transfer

Objective Implement the function _findInOriginalNode, which identifies all shared hits between the current original node and the corresponding part in the reference sequence. Note you do not need to scan all positions again. In the end print all occurrences to the console.
Hints  Use the STL function `std::upper_bound` to conduct a binary search to find the first possible hit from the reference that is also represented by the current node.

```cpp
template <typename TJournalEntriesIterator, typename TPattern>
void _findInOriginalNode(String<int> & hitTarget,
TJournalEntriesIterator & entriesIt,
TPattern const & pattern,
String<int> const & refHits)
{
    // [A] Check if hits exist in the reference.
    // [B] Find upper bound to current physical position in sorted refHits using std::upper_bound.
    // [C] Make sure we do not miss hits that begin at physical position of current node.
    // [D] Store all hits that are found in the region of the reference which is covered by this
    // [E] Store the correct virtual position and check next hit.
}
```

Solution  Here is the solution to this function. Click more... below, to see a complete solution of everything we have done so far.

```cpp
template <typename TJournalEntriesIterator, typename TPattern>
void _findInOriginalNode(String<int> & hitTarget,
TJournalEntriesIterator & entriesIt,
TPattern const & pattern,
String<int> const & refHits)
{
    // Define an Iterator which iterates over the reference hit set.
    typedef typename Iterator<String<int> const, Standard>::Type THitIterator;

    // [A] Check if hits exist in the reference.
    if (!empty(refHits))
    {
        // [B] Find upper bound to current physical position in sorted refHits using std::upper_bound.
        // [C] Make sure we do not miss hits that begin at physical position of current node.
        THitIterator itHit = std::upper_bound(begin(refHits), end(refHits), (int)entriesIt->physicalPosition);
        if (itHit != begin(refHits) && *(itHit - 1) >= (int)entriesIt->physicalPosition)
            --itHit;

        // [D] Store all hits that are found in the region of the reference which is covered by this
        while ((int)*itHit < ((int)entriesIt->physicalPosition + (int)entriesIt->length) && itHit != end(refHits))
        {
            // [E] Store the correct virtual position and check next hit.
            appendValue(hitTarget, entriesIt->virtualPosition + (*itHit - (int)entriesIt->physicalPosition));
            ++itHit;
        }
    }
}
```

Include the necessary headers.

```cpp
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>

using namespace seqan;
```

We know implement the method to search for hits in an original node. We only need to check if the current node covers a region of the reference in which we’ve found a hit. We use the function `std::upper_bound` to find
the first element that is greater than the current physical position. Since, we’ve found an upper bound we have to check additionally if there exists a previous hit that lies directly on the physical begin position of our current node. We then include all hits that fit into this current node until we have found the first position where the pattern would cross the border of this node or we have reached the end of the reference hit set.

```cpp
template <typename TJournalEntriesIterator, typename TPattern>
void _findInOriginalNode(String<int> & hitTarget,
    TJournalEntriesIterator & entriesIt,
    TPattern const & pattern,
    String<int> const & refHits)
{
    // Define an Iterator which iterates over the reference hit set.
    typedef typename Iterator<String<int> const, Standard>::Type THitIterator;

    // [A] Check if hits exist in the reference.
    if(!empty(refHits))
    {
        // [B] Find upper bound to current physical position in sorted refHits using std::upper_bound.
        THitIterator itHit = std::upper_bound(begin(refHits),end(refHits),(
            int(entriesIt->physicalPosition);

        // [C] Make sure we do not miss hits that begin at physical position of current node.
        if(itHit != begin(refHits) && *(itHit - 1) >= (
            int(entriesIt->physicalPosition)
            --itHit;

        // [D] Store all hits that are found in the region of the reference which is covered by this node.
        while((int)*itHit < ((int)entriesIt->physicalPosition + (int)entriesIt->length - (int)length(pattern) + 1) && itHit != end(refHits))
        {
            // [E] Store the correct virtual position and check next hit.
            appendValue(hitTarget, entriesIt->virtualPosition + ((*itHit - (int)entriesIt->physicalPosition + 1)
                ++itHit;
        }
    }
}
```

Implementing the backbone to search for a pattern in the reference string.

```cpp
template <typename TValue, typename THostSpec, typename TJournalSpec, typename TBufferSpec, typename TPattern>
void findPatternInJournalString(String<int> & hitTarget,
    String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const & journal,
    TPattern const & pattern,
    String<int> const & refHits)
{
    typedef String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const TJournal;
    typedef typename JournalType<TJournal>::Type TJournalEntries;
    typedef typename Iterator<TJournalEntries>::Type TJournalEntriesIterator;

    if (length(pattern) > length(journal))
        return;

    TJournalEntriesIterator it = begin(journal._journalEntries);
    TJournalEntriesIterator itEnd = findInJournalEntries(journal._journalEntries, length(journal)) + 1;

    while(it != itEnd)
    {
        if (it->segmentSource == SOURCE_ORIGINAL)
            // Find a possible hit in the current source vertex.
            _findInOriginalNode(hitTarget, it, pattern, refHits);

        if (it->segmentSource == SOURCE_PATCH)
            // Search for pattern within the patch node.
            _findInPatchNode(hitTarget, it, journal, pattern);
```
// Scan the border for a possible match.
// _searchAtBorder(hitTarget, it, journal, pattern);
++it;
}

Implementing the search within the reference sequence.

```cpp
template <typename TString, typename TPattern>
void findPatternInReference(String<int> & hits,
TString const & reference,
TPattern const & pattern)
{
    // Check whether the pattern fits into the sequence.
    if (length(pattern) > length(reference))
        return;

    // for (unsigned pos = 0; pos < length(reference) - length(pattern) + 1; ++pos)
    {
        bool isHit = true;

        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern)
        {
            if (pattern[posPattern] != reference[posPattern + pos])
            {
                isHit = false;
                break;
            }
        }
        // Report the position if found a hit.
        if (isHit)
            appendValue(hits, pos);
    }
}
```

Implementing the backbone of the search.

```cpp
template <typename TString, typename TPattern>
void searchPattern(StringSet<String<int> > & hitSet,
StringSet<TString, Owner<JournaledSet> > const & journalSet,
TPattern const & pattern)
{
    typedef typename Host<TString>::Type THost;

    // Check for valid initial state.
    if (empty(globalReference(journalSet)))
    {
        std::cout << "No reference set. Aborted search!" << std::endl;
        return;
    }

    // Reset the hitSet to avoid phantom hits coming from a previous search.
    clear(hitSet);
    resize(hitSet, length(journalSet) + 1);
    // Access the reference sequence.
    THost & globalRef = globalReference(journalSet);
    // Search for pattern in the reference sequence.
```
findPatternInReference(hitSet[0], globalRef, pattern);

// Search for pattern in the journaled sequences.
for (unsigned i = 0; i < length(journalSet); ++i)
    findPatternInJournalString(hitSet[i+1], journalSet[i], pattern, hitSet[0]);
}

Implement the \textit{loadAndJoin} method.

Implementing the main method.

\begin{verbatim}
int main()
{
    // Definition of the used types.
    typedef String<Dna,Alloc<> > TSequence;
    typedef String<Dna,Journaled<Alloc<>.,SortedArray,Alloc<> > > TJournal;
    typedef StringSet< TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/Users/rahn_r/Downloads/sequences.fasta";
    std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
    if (!databaseFile.good())
    {
        std::cerr << "Cannot open file " << seqDatabasePath << ">
    
    // Reading each sequence and journal them.
    TJournaledSet journalSet;
    JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;
    loadAndJoin(journalSet, databaseFile, joinConfig);
    databaseFile.close();

    // Define a pattern and start search.
    StringSet<String<int> > hitSet;
    TSequence pattern = "GTGGT";
    std::cout << "Search for: " << pattern << 

    searchPattern(hitSet, journalSet, pattern);

    Printing the hits of the reference sequence.
    if (empty(hitSet[0]))
    {
        std::cout << "No hit in reference " << std::endl;
    }
    else
    {
        std::cout << "Hit in reference " << " at ";
        for (unsigned j = 0; j < length(hitSet[0]); ++j)
            std::cout << hitSet[0][j] << ": " << infix(globalReference(journalSet), hitSet[0][j])
        std::cout << std::endl;
    }

    Printing the hits of the journaled sequences.
    for (unsigned i = 1; i < length(hitSet); ++i)
    {
        if (empty(hitSet[i]))
        {

\end{verbatim}

402 Chapter 2. API Documentation
And here is the result.

Search for: GTGGT:
Hit in reference at 311: GTGGT 644: GTGGT
Hit in sequence 0 at 312: GTGGT
Hit in sequence 1 at 308: GTGGT
Hit in sequence 2 at 311: GTGGT
Hit in sequence 3 at 327: GTGGT
Hit in sequence 4 at 317: GTGGT
Hit in sequence 5 at 320: GTGGT

We are almost at the end of our online-search algorithm. Let’s now implement the method _findInPatchNode. We basically had this already implemented when we wrote the search function for the reference. Let’s recall this part together.

First we write the body of our function and define now an Iterator over the Journaled String.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _findInPatchNode(String<int> & hitTarget,
TJournalEntriesIterator & entriesIt,
TJournal const & journal,
TPattern const & pattern)
{
    typedef typename Iterator<TJournal const, Standard>::Type TJournalIterator;
    // Search for pattern in the insertion node.
    TJournalIterator patchIter = iter(journal, entriesIt->virtualPosition);
    TJournalIterator patchEnd = patchIter + _max(0, (int)entriesIt->length - (int)length(pattern) + 1);
    // Move step by step over search region.
    for (; patchIter != patchEnd; ++patchIter)
    {
        TJournalIterator verifyIter = patchIter;
        bool isHit = true;
        // Search for pattern in the insertion node.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern, ++verifyIter)
        {
            // Comparing the pattern value with the current value of the iterator.
            if(pattern[posPattern] != getValue(verifyIter))
            {
```
isHit = false;
break;
}

if (isHit)
    appendString(hitTarget, position(patchIter));
}

To ensure that we are not missing any hits we also have to scan the regions where the pattern is leaving the current node. You can solve this problem in the next assignment.

Assignment 4

Type Review

Objective Implement the last function _searchAtBorder, which identifies all hits that cross the border of the current node to the next node.

Hints The following code snippet provides you with the backbone for this function. Fill in the missing parts [A], [B], [C] and [D].

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _searchAtBorder(String<int> & hitTarget,
                     TJournalEntriesIterator & entriesIt,
                     TJournal const & journal,
                     TPattern const & pattern)
{
    // [A] Determine first position of the at which pattern crosses the border of current node.
    TJournalIterator nodeIter = iter(journal, entriesIt->virtualPosition + _max(0, (int)entriesIt->length - (int)length(pattern) + 1));

    // [B] Determine last position before pattern exits the current node or reaches the end of the sequence.
    TJournalIterator nodeEnd = iter(journal, _min(entriesIt->virtualPosition + entriesIt->length, length(journal) - length(pattern) + 1));

    if (nodeEnd == end(journal))
        return;

    for (; nodeIter != nodeEnd; ++nodeIter)
    {
        // [D] Scan pattern in current window and report possible hits.
    }
}
```

Solution

Here is the solution to this function. Click more... below, to see a complete solution of everything we have done so far.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _searchAtBorder(String<int> & hitTarget,
                     TJournalEntriesIterator & entriesIt,
                     TJournal const & journal,
                     TPattern const & pattern)
{
    typedef typename Iterator<TJournal const, Standard>::Type TJournalIterator;

    TJournalIterator nodeIter = iter(journal, entriesIt->virtualPosition + _max(0, (int)entriesIt->length - (int)length(pattern) + 1));

    TJournalIterator nodeEnd = iter(journal, _min(entriesIt->virtualPosition + entriesIt->length, length(journal) - length(pattern) + 1));

    if (nodeEnd == end(journal))
        return;

    for (; nodeIter != nodeEnd; ++nodeIter)
    {
        // [D] Scan pattern in current window and report possible hits.
    }
}
```
Search at the border the current node for the pattern.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _searchAtBorder(String<int> & hitTarget,
                     TJournalEntriesIterator & entriesIt,
                     TJournal const & journal,
                     TPattern const & pattern)
{
    typedef typename Iterator<TJournal const, Standard>::Type TJournalIterator;

    // [A] Determine first position of the at which pattern crosses the border of current node.
    TJournalIterator nodeIter = iter(journal, entriesIt->virtualPosition + _max(0,
                                                                  (int)entriesIt->length - (int)length(pattern) + 1));

    // [B] Determine last position before pattern exits the current node or reaches the end of the sequence.
    TJournalIterator nodeEnd = iter(journal, _min(entriesIt->virtualPosition + entriesIt->length, length(journal) - length(pattern) + 1));

    if (nodeEnd == end(journal))
        return;

    for (; nodeIter != nodeEnd; ++nodeIter)
    {
        // [D] Scan pattern in current window and report possible hits.
        TJournalIterator verifyIter = nodeIter;
        bool isHit = true;

        // Compare pattern with current search window.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern, ++verifyIter)
        {
            // Comparing the pattern value with the current value of the iterator.
            if (pattern[posPattern] != getValue(verifyIter))
            {
                isHit = false;
                break;
            }
        }

        // Report hit if found.
        if (isHit)
            appendValue(hitTarget, position(nodeIter));
    }
}
```

Include the necessary headers.

```cpp
#include <iostream>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>

using namespace seqan;
```

2.1. Tutorial
Search for the pattern in the insertion region covered by the current node.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _findInPatchNode(String<int> & hitTarget, TJournalEntriesIterator & entriesIt, TJournal const & journal, TPattern const & pattern)
{
    typedef typename Iterator<TJournal const, Standard>::Type TJournalIterator;

    // Search for pattern in the insertion node.
    TJournalIterator patchIter = iter(journal, entriesIt->virtualPosition);
    TJournalIterator patchEnd = patchIter + _max(0, (int)entriesIt->length - (int)length(pattern) + 1);

    // Move step by step over search region.
    for (; patchIter != patchEnd; ++patchIter)
    {
        TJournalIterator verifyIter = patchIter;
        bool isHit = true;

        // Search for pattern in the insertion node.
        for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern, ++verifyIter)
        {
            // Comparing the pattern value with the current value of the iterator.
            if (pattern[posPattern] != getValue(verifyIter))
            {
                isHit = false;
                break;
            }
        }

        if (isHit)
        {
            appendValue(hitTarget, position(patchIter));
        }
    }
}
```

Check if hit was reported for this region in the reference sequence.

```cpp
template <typename TJournalEntriesIterator, typename TPattern>
void _findInOriginalNode(String<int> & hitTarget, TJournalEntriesIterator & entriesIt, TPattern const & pattern, String<int> const & refHits)
{
    // Define an Iterator which iterates over the reference hit set.
    typedef typename Iterator<String<int> const, Standard>::Type THitIterator;

    // Check if hits exist in the reference.
    if(!empty(refHits))
    {
        // Find upper bound to physical position in sorted refHits.
        THitIterator itHit = std::upper_bound(begin(refHits), end(refHits), (int)entriesIt->physicalPosition);

        // Make sure we do not miss hits that begin at physical position of current node.
        if(itHit != begin(refHits) && *(itHit - 1) >= (int)entriesIt->physicalPosition)
        {
            --itHit;
        }
    }
```
// Store all hits that are found in the region of the reference which is covered by this node.
while ((int)*itHit < ((int)entriesIt->physicalPosition + (int)entriesIt->length - (int)length(pattern) + 1) && itHit != end(refHits))
{
    appendValue(hitTarget, entriesIt->virtualPosition + (*itHit - (int)entriesIt->physicalPosition));
    ++itHit;
}
}

Implementing the backbone of the search for the Journaled String.

```cpp
template <typename TValue, typename THostSpec, typename TJournalSpec, typename TBufferSpec, typename TPattern>
void findPatternInJournalString(String<int> & hitTarget,
                                String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const & journal,
                                TPattern const & pattern,
                                String<int> const & refHits)
{
    typedef String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> > const TJournal;
    typedef typename JournalType<TJournal>::Type TJournalEntries;
    typedef typename Iterator<TJournalEntries>::Type TJournalEntriesIterator;

    if (length(pattern) > length(journal))
        return;

    TJournalEntriesIterator it = begin(journal._journalEntries);
    TJournalEntriesIterator itEnd = findInJournalEntries(journal._journalEntries, length(journal) - length(pattern) + 1) + 1;
    while (it != itEnd)
    {
        if (it->segmentSource == SOURCE_ORIGINAL)
        {
            // Find a possible hit in the current source vertex.
            _findInOriginalNode(hitTarget, it, pattern, refHits);
        }
        if (it->segmentSource == SOURCE_PATCH)
        {
            // Search for pattern within the patch node.
            _findInPatchNode(hitTarget, it, journal, pattern);
        }
        // Scan the border for a possible match.
        _searchAtBorder(hitTarget, it, journal, pattern);
        ++it;
    }
}
```

Implementing the search for the reference sequence.

```cpp
template <typename TString, typename TPattern>
void findPatternInReference(String<int> & hits,
                            TString const & reference,
                            TPattern const & pattern)
{
    // Check whether the pattern fits into the sequence.
    if (length(pattern) > length(reference))
        return;

    // for (unsigned pos = 0; pos < length(reference) - length(pattern) + 1; ++pos)
    {
        bool isHit = true;
```
for (unsigned posPattern = 0; posPattern < length(pattern); ++posPattern)
{
    if (pattern[posPattern] != reference[posPattern + pos])
    {
        isHit = false;
        break;
    }
}

// Report the position if found a hit.
if (isHit)
    appendValue(hits, pos);

The backbone of the search method.

\[
\text{template <typename TString, typename TPattern> void searchPattern(StringSet<String<int>> & hitSet, StringSet<TString, Owner<JournaledSet>> const & journalSet, TPattern const & pattern) }
\]

\[
\text{typedef typename Host<TString>::Type THost;}
\]

// Check for valid initial state.
if (empty(globalReference(journalSet)))
{
    std::cout << "No reference set. Aborted search!" << std::endl;
    return;
}

// Reset the hitSet to avoid phantom hits coming from a previous search.
clear(hitSet);
resize(hitSet, length(journalSet) + 1);

// Access the reference sequence.
THost & globalRef = globalReference(journalSet);

// Search for pattern in the reference sequence.
findPatternInReference(hitSet[0], globalRef, pattern);

// Search for pattern in the journaled sequences.
for (unsigned i = 0; i < length(journalSet); ++i)
    findPatternInJournalString(hitSet[i+1], journalSet[i], pattern, hitSet[0]);

Loading and joining the sequences.

int main()
{
    // Definition of the used types.
    typedef String<Dna,Alloc<> > TSequence;
    typedef String<Dna,Journaled<Alloc<>,SortedArray,Alloc<> > > TJournal;
    typedef StringSet< TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/Users/rahn_r/Downloads/sequences.fasta";
    std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
    if (!databaseFile.good())
        // Code for handling the error
{  
  std::cerr << "Cannot open file " << seqDatabasePath << "!" << std::endl;  
}

// Reading each sequence and journal them.  
TJournaledSet journalSet;  
JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;  
loadAndJoin(journalSet, databaseFile, joinConfig);  
databaseFile.close();  

// Define a pattern and start search.  
StringSet<String<int>> hitSet;  
TSequence pattern = "GTGGT";  
std::cout << "Search for: " << pattern << ":\n";  
searchPattern(hitSet, journalSet, pattern);

Reporting the identified hits.
if (empty(hitSet[0]))
  {
    std::cout << "No hit in reference " << std::endl;
  }
else  
  {
    std::cout << "Hit in reference " << " at ";  
    for (unsigned j = 0; j < length(hitSet[0]); ++j)  
      std::cout << hitSet[0][j] << ": " << infix(globalReference(journalSet), hitSet[0][j], hitSet[0][j] + length(pattern)) << "\t";
  }
std::cout << std::endl;  
for (unsigned i = 1; i < length(hitSet); ++i)
  {
    if (empty(hitSet[i]))
      {
        std::cout << "No hit in sequence " << i - 1 << std::endl;
      }
    else  
      {
        std::cout << "Hit in sequence " << i - 1 << " at ";  
        for (unsigned j = 0; j < length(hitSet[i]); ++j)  
          std::cout << hitSet[i][j] << ": " << infix(value(journalSet,i-1), hitSet[i][j],hitSet[i][j] + length(pattern)) << "\t";
      }
  }  
std::cout << std::endl;
}

return 0;  
}

And here is the result.

Search for: GTGGT:
Hit in reference at 311: GTGGT 644: GTGGT
Hit in sequence 0 at 151: GTGGT 312: GTGGT
Hit in sequence 1 at 308: GTGGT
Hit in sequence 2 at 311: GTGGT 507: GTGGT
Hit in sequence 3 at 327: GTGGT
Hit in sequence 4 at 307: GTGGT 312: GTGGT 317: GTGGT
Hit in sequence 5 at 0: GTGGT 320: GTGGT 986: GTGGT
Congratulations! You have just implemented a cool online-search which is speed up by exploiting the parallelism given by the data set. And here is the final result.

Search for: GTGGT:
Hit in reference at 311: GTGGT 644: GTGGT
Hit in sequence 0 at 151: GTGGT 312: GTGGT
Hit in sequence 1 at 308: GTGGT
Hit in sequence 2 at 311: GTGGT 507: GTGGT
Hit in sequence 3 at 308: GTGGT
Hit in sequence 4 at 307: GTGGT 312: GTGGT 317: GTGGT
Hit in sequence 5 at 0: GTGGT 320: GTGGT 986: GTGGT

Assignment 5

Type Transfer

Objective Try to replace the brute force versions using using SeqAn’s Finder and Pattern concept. You can find additional material to this topic in the Pattern Matching Tutorial.

Solution Now we want to replace the brute force methods with some cool pattern matching algorithms. Therefore we include the header `<seqan/finder.h>`.

```cpp
#include <iostream>
#include <seqan/find.h>
#include <seqan/seq_io.h>
#include <seqan/journaled_set.h>
using namespace seqan;
```

Now we can use the Finder interface of SeqAn. One cool thing of the usage of the Finder class is that we don’t have to check for the borders anymore. This will do the Finder for us. We only have to specify the correct infix over which the Finder should iterate to find the pattern. We first compute the positions that enclose the search region. Afterwards, we get an infix for this region and pass it to the Finder’s constructor. We also have to define the Pattern object which gets the pattern we are searching for. Then we can simply call the function `find` until we there is no more match. Be careful when storing the position that the Finder is returning. We have to recompute the correct virtual position since we used an infix of the original search text.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _searchAtBorder(String<int> & hitTarget, TJournalEntriesIterator & entriesIt, TJournal const & journal, TPattern const & needle)
{
    typedef typename Position<TJournal>::Type TPosition;

    // Define region before the border to the next node to search for the pattern.
    TPosition infixBegin = entriesIt->virtualPosition + _max(0,(int)entriesIt->length - (int)length(needle) + 1);
    TPosition infixEnd = _min(length(journal),entriesIt->virtualPosition + entriesIt->length + length(needle) - 1);

    TPattern tmpInsBuffer = infix(journal, infixBegin, infixEnd);
    Finder<TPattern const> finder(tmpInsBuffer);
    Pattern<TPattern, Horspool> pattern(needle);
    while (find(finder, pattern))
    {
        appendValue(hitTarget, infixBegin + beginPosition(finder));
    }
}
```
So the biggest change is done. We simply repeat the changes from above and watch to get the correct virtual position.

```cpp
template <typename TJournalEntriesIterator, typename TJournal, typename TPattern>
void _findInPatchNode(String<int> & hitTarget,
                      TJournalEntriesIterator & entriesIt,
                      TJournal const & journal,
                      TPattern const & needle)
{
    typedef typename Position<TJournal>::Type TPosition;

    TPosition infixBegin = entriesIt->virtualPosition;
    TPosition infixEnd = entriesIt->virtualPosition + entriesIt->length;

    TPattern tmpInsBuffer = infix(journal, infixBegin, infixEnd);
    Finder<TPattern const> finder(tmpInsBuffer);
    Pattern<TPattern, Horspool> pattern(needle);

    while (find(finder, pattern))
    {
        appendValue(hitTarget, entriesIt->virtualPosition + beginPosition(finder));
    }
}
```

Of course we don’t need to change anything for the original nodes.

```cpp
template <typename TJournalEntriesIterator, typename TPattern>
void _findInOriginalNode(String<int> & hitTarget,
                          TJournalEntriesIterator & entriesIt,
                          TPattern const & pattern,
                          String<int> const & refHits)
{
    typedef typename Iterator<String<int> const, Standard>::Type THitIterator;

    if (!empty(refHits))
    {
        // Define an Iterator which iterates over the reference hit set.
        typedef typename TJournal::Type TJournalEntries;

        THitIterator itHit = std::upper_bound(begin(refHits), end(refHits), (int)entriesIt->physicalPosition);
        // Make sure we do not miss hits that begin at physical position of current node.
        if (itHit != begin(refHits) && *(itHit - 1) >= (int)entriesIt->physicalPosition)
            --itHit;
        // Store all hits that are found in the region of the reference which is covered by this node.
        while ((int)*itHit < ((int)entriesIt->physicalPosition + (int)entriesIt->length - (int)length(pattern)) + itHit)
        {
            appendValue(hitTarget, entriesIt->virtualPosition + (*itHit - (int)entriesIt->physicalPosition) + itHit);
            ++itHit;
        }
    }
}
```

Also the function `findPatternInJournalString` remains the same.

```cpp
template <typename TValue, typename THostSpec, typename TJournalSpec, typename TBufferSpec, typename TPattern>
void findPatternInJournalString(String<int> & hitTarget,
                                String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> const & journal,
                                TPattern const & pattern,
                                String<int> const & refHits)
{
    typedef String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> const & journal;
    typedef typename JournalType<TJournal>::Type TJournalEntries;
```
typedef typename Iterator<TJournalEntries>::Type TJournalEntriesIterator;

if (length(pattern) > length(journal))
    return;

TJournalEntriesIterator it = begin(journal._journalEntries);
TJournalEntriesIterator itEnd = findInJournalEntries(journal._journalEntries, length(journal) - length(pattern) + 1) + 1;

while (it != itEnd)
{
    if (it->segmentSource == SOURCE_ORIGINAL)
    {
        // Find a possible hit in the current source vertex.
        _findInOriginalNode(hitTarget, it, pattern, refHits);
    }
    if (it->segmentSource == SOURCE_PATCH)
    {
        // Search for pattern within the patch node.
        _findInPatchNode(hitTarget, it, journal, pattern);
    }
    // Scan the border for a possible match.
    _searchAtBorder(hitTarget, it, journal, pattern);
    ++it;
}

We will switch to the Finder concept for the function findPatternInReference too. This is done quickly, since we have the basis already laid down in the previous functions.

template <typename TString>
void findPatternInReference(String<int> & hits,
    TString const & reference,
    TString const & needle)
{
    // Check whether the pattern fits into the sequence.
    if (length(needle) > length(reference))
        return;

    Finder<TString const> finder(reference);
    Pattern<TString, Horspool> pattern(needle);
    while (find(finder, pattern))
        appendValue(hits, beginPosition(finder));
}

From here on, we don’t have to change anything.

template <typename TString, typename TPattern>
void searchPattern(StringSet<String<int> > & hitSet,
    StringSet<TString, Owner<JournaledSet> > const & journalSet,
    TPattern const & pattern)
{
    typedef typename Host<TString>::Type THost;

    // Check for valid initial state.
    if (empty(globalReference(journalSet)))
    {
        std::cout << "No reference set. Aborted search!" << std::endl;
        return;
    }

    // Reset the hitSet to avoid phantom hits coming from a previous search.
clear(hitSet);
resize(hitSet, length(journalSet) + 1);

// Access the reference sequence.
THost & globalRef = globalReference(journalSet);

// Search for pattern in the reference sequence.
findPatternInReference(hitSet[0], globalRef, pattern);

// Search for pattern in the journaled sequences.
for (unsigned i = 0; i < length(journalSet); ++i)
  findPatternInJournalString(hitSet[i+1], journalSet[i], pattern, hitSet[0]);
}

We write the same main body ...

template <typename TString, typename TStream, typename TSpec>
inline int loadAndJoin(StringSet<TString, Owner<JournaledSet> > & journalSet,
                        TStream & stream,
                        JoinConfig<TSpec> const & joinConfig)
{
  typedef typename Host<TString>::Type THost;

  RecordReader<std::ifstream, SinglePass<> > reader(stream);
  clear(journalSet);
  String<char> seqId;
  THost sequence;

  // No sequences in the fasta file!
  if (atEnd(reader))
  {
    std::cerr << "Empty FASTA file." << std::endl;
    return -1;
  }

  // First read sequence for reference sequence.
  if (readRecord(seqId, sequence, reader, Fasta()) != 0)
  {
    std::cerr << "ERROR reading FASTA." << std::endl;
    return 1;
  }

  // We have to create the global reference sequence otherwise we loose the information after
  createGlobalReference(journalSet, sequence);

  // If there are more
  while (!atEnd(reader))
  {
    if (readRecord(seqId, sequence, reader, Fasta()) != 0)
    {
      std::cerr << "ERROR reading FASTA." << std::endl;
      return 1;
    }
    appendString(journalSet, TString(sequence));
    join(journalSet, length(journalSet) - 1, joinConfig);
  }
  return 0;
}

and finally print the results.
```cpp
int main()
{
    // Definition of the used types.
    typedef String<Dna, Alloc<> > TSequence;
    typedef String<Dna, Journaled<Alloc<>>, SortedArray, Alloc<> > > TJournal;
    typedef StringSet< TJournal, Owner<JournaledSet> > TJournaledSet;

    // Open the stream to the file containing the sequences.
    String<char> seqDatabasePath = "/Users/rahn_r/Downloads/sequences.fasta";
    std::ifstream databaseFile(toCString(seqDatabasePath), std::ios_base::in);
    if(!databaseFile.good())
    {
        std::cerr << "Cannot open file " << seqDatabasePath << "]!" << std::endl;
    }

    // Reading each sequence and journal them.
    TJournaledSet journalSet;
    JoinConfig<GlobalAlign<JournaledCompact> > joinConfig;
    loadAndJoin(journalSet, databaseFile, joinConfig);
    databaseFile.close();

    // Define a pattern and start search.
    StringSet<String<int> > hitSet;
    TSequence pattern = "GTGGT";
    std::cout << "Search for: " << pattern << "]n";
    searchPattern(hitSet, journalSet, pattern);

    if (empty(hitSet[0]))
    {
        std::cout << "No hit in reference " << std::endl;
    }
    else
    {
        std::cout << "Hit in reference " << " at ";
        for (unsigned j = 0; j < length(hitSet[0]); ++j)
            std::cout << hitSet[0][j] << "] " << infix(globalReference(journalSet), hitSet[0][j]);
        std::cout << std::endl;
    }

    for (unsigned i = 1; i < length(hitSet); ++i)
    {
        if (empty(hitSet[i]))
        {
            std::cout << "No hit in sequence " << i - 1 << std::endl;
        }
        else
        {
            std::cout << "Hit in sequence " << i - 1 << " at ";
            for (unsigned j = 0; j < length(hitSet[i]); ++j)
                std::cout << hitSet[i][j] << "] " << infix(value(journalSet,i-1), hitSet[i][j],hitSet[i][j] + length(pattern)) << "]t 
            std::cout << std::endl;
        }
    }

    return 0;
}
```
And here is the result using the Finder and Pattern concept of SeqAn.

Search for: GTGGT:
Hit in reference at 311: GTGGT 644: GTGGT
Hit in sequence 0 at 151: GTGGT 312: GTGGT
Hit in sequence 1 at 308: GTGGT
Hit in sequence 2 at 311: GTGGT 507: GTGGT
Hit in sequence 3 at 327: GTGGT
Hit in sequence 4 at 307: GTGGT 312: GTGGT 317: GTGGT
Hit in sequence 5 at 0: GTGGT 320: GTGGT 986: GTGGT

2.1.39 KNIME Nodes

**Learning Objective**  You will learn how to import new applications written in SeqAn into the KNIME Eclipse plugin. After completing this tutorial, you will be able to use self made applications in KNIME workflows.

**Difficulty**  Basic

**Duration**  1.5 h

**Prerequisites**  *A First Example, Parsing Command Line Arguments*

In this tutorial you will learn how to integrate new apps written in SeqAn into a KNIME workflow. The first part consists of preparing a dummy app such that it can be used in a KNIME workflow and in the second part you are asked to adapt the app such that it becomes a simple quality control tool.

**Important:** The steps described here are necessary if you want to develop and test new SeqAn apps in KNIME. If you only want to use existing SeqAn apps in KNIME follow *Creating Workflows with KNIME*.

**Importing SeqAn apps into KNIME**

For the first part of the tutorial follow the instructions in *Generating SeqAn KNIME Nodes* and import a dummy SeqAn app into KNIME.

**Create a useful KNIME workflow**

In the second part of the tutorial you are asked to modify the app you imported into KNIME such that it becomes a quality control tool.

**Type**  Transfer

**Objective**  Create a simple read mapping workflow in KNIME using ‘razers3’ and map the reads in *reads.fastq* to *ref.fasta*. Configure the node to use a *percent-identity* value of 99 and the output format could be ‘razers’. The third to last and second to last column show the matching position in the reference (begin and end respectively). The last one represents the number of matching characters in percent.

---

**ToC**

**Contents**

- KNIME Nodes
  - Importing SeqAn apps into KNIME
  - Create a useful KNIME workflow
You probably observe that you do not find a lot of matches. The reason for this are incorrectly called bases at the end of the reads.

**Type** Transfer

**Objective** Modify the ‘knime_node’ app such that it becomes a quality trimmer. You might start by just deleting the last bases of the reads (say 4 or 5) and then make the cutting depending on the actual quality values. Include the node into your workflow and inspect if the results change.

**Tip:** KNIME needs to know the input and output ports of a node. Therefore we must specify them using `ArgParseArgument::INPUTFILE` or `ArgParseArgument::OUTPUTFILE` as can be seen in the ‘knime_node’ app. In addition, KNIME needs to know the valid file endings, which you can specify with `setValidValues`, which is also shown in the example.

---

**2.1.40 Basic Techniques**

**Generic Programming**

SeqAn adopts **generic programming**, a paradigm that was proven to be an efficient design strategy in the C++ standard. The standard template library (STL) as part of the C++ standard is a prototypical example for generic programming. Generic programming designs algorithms and data structures in a way that they work on all types that meet a minimal set of requirements. An example for a generic data structure in the STL is the class `vector`: It is a container for storing objects of a type `T` that are assignable, which means that we can assign one instance `s` of `T` to another instance `t` of `T`, i.e. the code `T t = s` is valid.

This kind of requirement to the interface of a type `T` is called a **concept**, and we say that a type `T` **implements** a concept, if it fulfills all requirements stated by that concept; for example the concept assignable is implemented by all built-in types and every class that has both a copy assignment operator and a copy constructor.

Generic programming has two implications:

1. Data structures and algorithms work on **all** types `T` that implement the relevant concept, i.e. relevant is not the type `T` itself but its interface, and
2. this concept is **minimal** in the sense that it contains only those requirements that are essential for the data structure or algorithm to work on `T`.

This way data structures and algorithms can be applied to as many types as possible, and hence generic programming promotes the generality of the library.

Generic data types and algorithms can be implemented in C++ using templates. A class template parameterizes a class with a list of types or constants. For example, a declaration for the class `vector` could be:

```cpp
template<typename T> class vector;
```
where \( T \) stands for the value type, i.e. the type of the values that will be stored in \texttt{vector}. The template is generic, it can be applied to any type \( T \). For example, a vector for storing \texttt{int} values is instantiated by:

\begin{verbatim}
vector<int> my_vector;
\end{verbatim}

That is we use \texttt{int} as template argument for \( T \), and the result of the instantiation is an object \texttt{my_vector} of the complete type \texttt{vector<int>}. The compiler employs the same template, i.e. the same piece of code, for different template argument types. The compilation succeeds if the applied template argument type supports all uses of the parameter \( T \) within the template code, so the C++ template instantiation process implies the minimality of the concepts.

Further reading

For more information about generic programming and the STL we recommend reading:

- Vandervoorde, Josuttis: C++ Templates - The complete guide, Addison-Wesley

\section*{2.1.41 Metafunctions}

Generic algorithms usually have to know certain types that correspond to their arguments. An algorithm on containers may need to know which type of values are stored in the string, or what kind of iterator we need to access it. The usual way in the STL is to define the value type of a class like \texttt{vector} as a member typedef of this class, so it can be retrieved by \texttt{vector::value_type}.

Unfortunately member typedef declarations have the same disadvantages as any members: Since they are specified by the class definition, they cannot be changed or added to the class without changing the code of the class, and it is not possible in C++ to define members for built-in types. What we need therefore is a mechanism that returns an output type (e.g. the value type) given an input type (e.g. the string) and doing so does not rely on members of the input type, but instead uses some kind of global interface.

Such task can be performed by \textbf{metafunctions}, also known as \textbf{type traits}. A metafunction is a construct to map some types or constants to other entities like types, constants, functions, or objects at compile time. The name metafunction comes from fact that they can be regarded as part of a meta-programming language that is evaluated during compilation.

In SeqAn we use class templates to implement metafunctions in C++. Generic algorithms usually have to know certain types that correspond to their arguments: An algorithm on strings may need to know which type of characters are stored in the string, or what kind of iterator can be used to browse it. SeqAn uses Metafunctions (also known as “traits”) for that purpose. For example: Assuming that we define a string of amino acids:

\begin{verbatim}
String<AminoAcid> str = "ARN";
\end{verbatim}

Now lets define a function that exchanges the first two values in a string:
void exchangeFirstValues(String<AminoAcid> & str)
{
    if (length(str) < 2) return;
    AminoAcid temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}

Since this function only works for instances of String<AminoAcid>, we could try to make it more general by making a template out of it.

template <typename T>
void exchangeFirstValues(T & str)
{
    if (length(str) < 2) return;
    AminoAcid temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}

Now the function works for all sequence types T that store AminoAcid objects, but it will fail for other value types as soon as the variable temp cannot store str[0] anymore. To overcome this problem, we must redefine temp in a way that it can store a value of the correct type. The question is: “Given a arbitrary type T, what is the value type of T?”

The metafunction Value answers this question: “The value type of T is given by Value<T>::Type.”

Hence, the final version of our function exchangeFirstValues reads as follows:

template <typename T>
void exchangeFirstValues(T & str)
{
    typename Value<T>::Type temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}

We can view Value as a kind of “function” that takes T as an argument (in angle brackets) and returns the required value type of T. In fact, Value is not implemented as a C++ function, but as a class template. This class template is specialized for each sequence type T in a way that the typedef Type provides the value type of T. Unfortunately, the current C++ language standard does not allow to write simply “Value<T> temp;”, so we must select the return value by appending “::Type”. The leading “typename” becomes necessary since Value<T>::Type is a type that depends on a template parameter of the surrounding function template.

**Type Metafunctions**

The metafunction Value is a type metafunction, i.e. it is used to determine a type. Type metafunctions have the form:

typename TypeMetaFunc<T1, T2, ..., TN>::Type

**TypeMetaFunc** The name of the metafunction

**T1, T2, ..., TN** Arguments (types or constants)

**Type** The resulting type

The keyword typename must be stated if one of the arguments T1, T2, ..., TN is or uses a template parameter. For example the following piece of code uses the metafunction Iterator to determine an iterator type for a string class:
String<char> str = "I am a string";
Iterator<String<char>> ::Type it = begin(str);
while (! atEnd(it, str))
{
    ::std::cout << *it;
    ++it;
}

Value Metafunctions

Metafunctions can also be used to determine constant values at compile time. The general form of value metafunctions is:

\[
\text{VALUE_META_FUNC}\langle T_1, T_2, \ldots, T_N\rangle::\text{VALUE}
\]

**VALUE_META_FUNC** The name of the metafunction

**T1, T2, ..., TN** Arguments (types or constants)

**VALUE** The resulting constant value

For example the following function prints the length of a fixed sized string using the value metafunction `LENGTH`:

```cpp
template <typename T>
void printLenOfFixedSizeString(T const &)
{
    ::std::cout << LENGTH<T>::VALUE;
}
```

String<char, Array<100>> my_str;
printLenOfFixedSizeString(my_str);

SeqAn Metafunctions

If you want to search for metafunctions only you can do so by only selecting the metafunction category to the left of the search window at the online documentation.

Assignment 1

**Objective** Write a generic program that swaps the value ranges \([i, i+k)\) and \([j, j+k)\) of a container `str`. The container should be specified as a template argument `T`.

**Hint** Use the Metafunctions `Value` to access the type of the elements in the container. Use the function `value` to assign the values.

**Solution** We want to have a generic version, similar to the function `ExchangeFirstValues` on the previous page.

Hence we could define the function as follows:

```cpp
template <typename T>
void swap(T& container, int i, int j, int k)
{
    // The function is now quite generic allowing any container of type T. In addition we specify two positions that should be swapped (as integers which is not really generic, but it suffices for the demo) an the length of the swapped region. Now we can define a helper variable `help`, which can be of type T.
}
```
// define helper variable
T help;
resize(help,k);

for (int x=0; x<k; ++x)
    value(help,x) = container[i+x];

and do the swapping

for (int x=0; x<k; ++x)
    value(container,i+x) = value(container,j+x);
for (int x=0; x<k; ++x)
    value(container,j+x) = help[x];

return;
}

Thats it. We can now test our generic swap function using for example a String of characters or a String of integers.

swap(dna,1,4,2);
cout << dna << endl;

swap(numbers,1,7,2);
for ( TIntIterator it=begin(numbers); !atEnd(it); goNext(it)) {
    std::cout << *it;
}cout << endl;

return 0;
}

The whole program taken together looks as follows:

#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h>

using namespace seqan;
using namespace std;

template <typename T> void swap(T& container, int i, int j, int k)
{

// define helper variable
T help;
resize(help,k);

for (int x=0; x<k; ++x)
    value(help,x) = container[i+x];

for (int x=0; x<k; ++x)
    value(container,i+x) = value(container,j+x);
for (int x=0; x<k; ++x)
    value(container,j+x) = help[x];

return;
}
```c++
int main()
{
    typedef String<Dna> TDnaString;
    TDnaString dna = "AAAATTTT";

    typedef String<int> TIntString;
    typedef Iterator<String<int>, Rooted>::Type TIntIterator;

    TIntString numbers;
    appendValue(numbers,1); appendValue(numbers,1); appendValue(numbers,1);
    appendValue(numbers,1); appendValue(numbers,1); appendValue(numbers,1);
    appendValue(numbers,3); appendValue(numbers,3); appendValue(numbers,3);
    appendValue(numbers,3); appendValue(numbers,3); appendValue(numbers,3);

    swap(dna,1,4,2);
    cout << dna << endl;

    swap(numbers,1,7,2);
    for (TIntIterator it=begin(numbers); !atEnd(it); goNext(it)) {
        std::cout << *it;
    }
    cout << endl;
    return 0;
}
```

ToC

2.1.42 Generic Programming

A generic algorithm that is applicable to a type \( T \) needs not to be optimal for that type. The algorithm \texttt{find} in the standard library (ISO/IEC 1998, section 25.3.1.1) for example performs a sequential linear time search and is therefore capable of finding a given value in any standard compliant container. However, the container map was designed to support a faster logarithmic time search, so the algorithm \texttt{find} – though applicable – is not optimal for searching in map. This shows that sometimes a special algorithm could be faster than a generic algorithm. Hence, in order to achieve better performance, SeqAn supports refinements of algorithms. A special version is only useful if it really allows a speedup in some cases, and only in this case it will actually be implemented. Therefore we assume that for a given case always the most special applicable variant is the best, where we have to assure that there is always a definite most special candidate according to the \texttt{C++} function overload resolution rules (ISO/IEC 1998, sections 13.3 and 14.5.8). We can write \texttt{find(obj)} for any container type \texttt{obj}, and this invokes the most suitable implementation of \texttt{find} depending on the type of \texttt{obj}. We call this approach \texttt{template subclassing}. 

2.1. Tutorial
The technique of template subclassing may be summarized as follows:

- The data types are realized as default implementation or specialization of class templates, e.g., `Class`, which have at least one template parameter `TSpec`.
- Refinements of `Class` are specified by using in `TSpec` a tag class, e.g., `Subclass`, that means they are implemented as class template specializations `Class<Subclass>`.
- Whenever further refinements may be possible, we declare the tag classes as class templates with at least one template parameter `TSpec`, in which more tag classes can be used. For example we may implement a class template specialization `Class<Subclass<Subsubclass<<...>>>` . This way, we can reach arbitrary levels of specialization.
- Algorithms can be implemented for each level of specialization. If multiple implementations for different levels of specialization exist, then the C++ function overload resolution selects the most special from all applicable variants.

**Example: Generic q-gram hashing**

In many applications in bioinformatics so called q-grams are used. A short string of length q can be interpreted as a number to the base of the cardinality of the alphabet. So for example for the Dna alphabet `cgta=0*1+3*4+2*16+1*64=108`. q-grams can be gapped or ungapped. In the gapped case they are often called shapes and are simply an ordered list of integers. The number of integers in the list is called the size of a shape whereas the largest element -1 is called the span.

The following code snippet shows a generic algorithm for computing all hash values for a shape. The function `span` applied to the shape `s = s1, ..., sq` returns `sq - 1`.

```cpp
template<typename TShape, typename TString>
void hashAll(TShape & shape, TString & str)
{
    typedef typename Iterator<TString>::Type TIterator;
    TIterator it = begin(str);
    TIterator it_end = end(str) - span(shape);
    while (it != it_end) {
        unsigned int hash_value = hash(shape, it);
        /* do some things with the hash value */
        ++it;
    }
}
```

Each shape has to know the alphabet, so we specify this value type in the first template parameter of `Shape`. The actual specialization is selected in the second template parameter `TSpec`.

```cpp
template<typename TValue, typename TSpec = SimpleShape>
class Shape;
```

the default is `SimpleShape` which is simply an ungapped shape storing merely the length of the shape from which it can deduce its span and size.

```cpp
template<typename TValue>
class Shape<TValue, SimpleShape>
{
    public:
        unsigned int span;
};
```

If we know q at compile time, then we can specify it in a template parameter and define span as a static member:

```cpp
template<unsigned int q = 0>
class UngappedShape;
```

```cpp
template<typename TValue, unsigned int q>
class Shape<TValue, UngappedShape<q> >
{
    public:
        unsigned int span;
};
```
The question is now, whether we can speed up the above `hashAll` functions for specializations of the class `shape` like ungapped shapes. A little thinking yields a positive answer to that question. Indeed, for ungapped shapes, we can incrementally compute the next hash value from a given hashvalue in constant time using the formula

$$\text{hash}(a_{i+1}...a_{i+q}) = \text{hash}(a_{i}...a_{i+q}) + a_{i+q} - a_{i} \cdot \sum |a_{i}| \cdot \sum |\Sigma|^{q} + a_{i+q},$$

that means when shifting the shape along the sequence, we have to subtract the effect of the leftmost letter and add the effect of the rightmost letter, all scaled with the corresponding factor. All digits in between are shifted by multiplying them with the alphabet size. Obviously this allows for a much more efficient implementation of the `hashAll` function. This functionality can be encoded in the following function `hashNext`.

```cpp
#include <vector>

struct UngappedShape {
    unsigned int q;
};

template <typename TValue, unsigned int q, typename TIterator>
inline unsigned int hashNext(Shape< TValue, UngappedShape<q> > const & shape, TIterator it, unsigned int prev) {
    unsigned int val = prev * ValueSize<TValue>::VALUE - *it * shape.fac + *(it + shape.span);
    return val;
}
```

SeqAn aims at not using virtual functions for introducing polymorphism. Instead the concept is called **template subclassing**. Hence we can now define a specialized `hashAll` function for all ungapped shapes as follows.

```cpp
#include <vector>

template <typename TValue, unsigned int q, typename TString>
void hashAll(Shape< TValue, UngappedShape<q> > & shape, TString & str) {
    typedef typename Iterator<TString>::Type TIterator;
    TIterator it = begin(str); TIterator it_end = end(str) - span(shape);
    unsigned int hash_value = hash(shape, it);
    /* do some things with the hash value */
    while (++it != it_end) {
        unsigned int hash_value = hashNext(shape, it, hash_value);
        /* do some things with the hash value */
    }
}
```

That's pretty much it. The C++ resolution mechanisms will ensure that whenever you use an ungapped shape in your code, the more efficient `hashAll` function above will be compiled. Note that this decision is made at **compile time** as opposed to the virtual function mechanism which is invoked at **run time**.

### Further reading

For more information about generic programming and the STL we recommend reading.

- Vandervoorde, Josuttis: C++ Templates - The complete guide, Addison-Wesley

### Template Subclassing Demo

Here is an example of template subclassing.
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
// * Neither the name of Knut Reinert or the FU Berlin nor the names of
its contributors may be used to endorse or promote products derived
from this software without specific prior written permission.

// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.

// ==========================================================================
// Author: Manuel Holtgrewe <manuel.holtgrewe@fu-berlin.de>
// ==========================================================================

// Referenced by TemplateSubclassing.

#include <iostream>

// Setup class hierarchy by using inheritance for the specialization tags.

struct SpecA {};  // Base class -- most generic and thus a fallback.
struct SpecB {};  // Specialization for classes B and D. C automatically inherits everything
struct SpecC : SpecB {};  // from B and does not overwrite anything.
struct SpecD : SpecB {};  

// Template specialization tags

// Base class -- most generic and thus a fallback.
template<typename TSpec>
struct MyClass
{
    int x;

    MyClass() : x(0)
    {
    }
};

// Specialization for classes B and D. C automatically inherits everything
// from B and does not overwrite anything.
template<>
struct MyClass<SpecB>
```cpp
int x;

MyClass() : x(1)
{
};

template <>
struct MyClass<SpecD>
{
    int x;

    MyClass() : x(2)
    {}
};

// Define some functions that demonstrate overloading.

// Most generic case, "base implementation".
template <typename TSpec>
void foo(MyClass<TSpec> const & obj)
{
    std::cout << "foo(MyClass<typename TSpec> const & obj) called! obj.x == " << obj.x << \n"
};

// This function overwrites the generic implementation of foo() for the
// specialization B.
template <>
void foo(MyClass<SpecB> const & obj)
{
    std::cout << "foo(MyClass<SpecB> const & obj) called! obj.x == " << obj.x << \n"
};

// This function overwrites the generic implementation of foo() for the
// specialization C.
template <>
void foo(MyClass<SpecC> const & obj) {
    std::cout << "foo(MyClass<SpecC> const & obj) called! obj.x == " << obj.x << \n"
};

// The main function calls the functions from above.
int main()
{
    std::cout << "Template Subclassing Demo\n"

    MyClass<SpecA> a;
    MyClass<SpecB> b;
    MyClass<SpecC> c;
    MyClass<SpecD> d;

    foo(a); // => foo(MyClass<typename TSpec> const & obj) called! obj.x == 0
    foo(b); // => foo(MyClass<SpecB> const & obj) called! obj.x == 1
    foo(c); // => foo(MyClass<SpecC> const & obj) called! obj.x == 0
    foo(d); // => foo(MyClass<typename TSpec> const & obj) called! obj.x == 2

    return 0;
}
```
2.1.43 Global Function Interface

Example to include STL bitset such that they can be used as a String in SeqAn.

2.1.44 Basics

Alphabets

In SeqAn, alphabets are value types that can take a limited number of values and which hence can be mapped to a range of natural numbers. We can retrieve the number of different values of an alphabet, the alphabet size, by the metafunction `ValueSize`. Another useful metafunction called `BitsPerValue` can be used to determine the number of bits needed to store a value of a given alphabet. The order of a character in the alphabet (i.e., its corresponding natural number) can be retrieved by calling the function `ordValue`. In SeqAn, several standard alphabets are already predefined, for example `Dna Dna5`, `Rna`, `Rna5`, `Iupac`, `AminoAcid`, ....

Let’s start with a simple task. We want to write a program that outputs all letters of the predefined `AminoAcid` alphabet. First we include the corresponding header files and specify that we are using the namespace `seqan`.

```cpp
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>
using namespace seqan;
```

Next, we will define a template function `template<typename TAlphabet> void showAllLettersOfMyAlphabet(TAlphabet const&)` which will iterate over all the characters of the alphabet and outputs them. For this, we need to determine the alphabet size using the metafunction `ValueSize<TAlphabet>::VALUE`. Then we increment a counter from 0 to the alphabet size minus one and output the counter as well as the corresponding letter of the alphabet using a conversion constructor.

```cpp
template <typename TAlphabet>
void showAllLettersOfMyAlphabet(TAlphabet const &)
```

typedef typename Size<TAlphabet>::Type TSize;
TSize alphSize = ValueSize<TAlphabet>::VALUE;
for (TSize i = 0; i < alphSize; ++i)
    std::cout << i << ',' << TAlphabet(i) << " ";
std::cout << std::endl;
}

In the main program we simply call the above function using a number of alphabets that are predefined in SeqAn.

```cpp
int main()
{
    showAllLettersOfMyAlphabet(AminoAcid());
    showAllLettersOfMyAlphabet(Dna());
    showAllLettersOfMyAlphabet(Dna5());
    return 0;
}
```

This program produces the following output:
```
darwin10.0 : ./show_alphabets
0,A 1,R 2,N 3,D 4,C 5,Q 6,E 7,G 8,H 9,I 10,L 11,K 12,M 13,F 14,P 15,S 16,T 17,W 18,Y 19,V 20,B 21,Z 22,X 23, *
0,A 1,C 2,G 3,T
0,A 1,C 2,G 3,T 4,N
```

### Iterators

An iterator is an object that is used to browse through the values of a container. The metafunction `Iterator` can be used to determine an appropriate iterator type given a container. Some containers offer several kinds of iterators, which can be selected by an optional argument of `Iterator`. For example, the tag `Standard` can be used to get an iterator type that resembles the C++ standard random access iterator. The more elaborated `RootedIterator`, i.e., an iterator that knows its container, can be selected by specifying the `Rooted` tag.

Rooted iterators offer some convenience for the user: They offer additional functions like `container` for determining the container on which the iterator works, and they simplify the interface for other functions like `atEnd`. Moreover, rooted iterators may change the container’s length or capacity, which makes it possible to implement a more intuitive variant of a remove algorithm.

While rooted iterators can usually be converted into standard iterators, it is not always possible to convert standard iterators back into rooted iterators, since standard iterators may lack the information about the container they work on. Therefore, many functions that return iterators like `begin` or `end` return rooted iterators instead of standard iterators; this way, they can be used to set both rooted and standard iterator variables. Alternatively it is possible to specify the returned iterator type explicitly by passing the `Rooted` tag as a argument.

The following code piece shows examples for creating Iterators for `Containers`. If no iterator kind is specified, the metafunction `Iterator` assumes `Standard` and the function `begin` assumes `Rooted`. Both `it1` and `it2` are standard iterators, whereas `it3` and `it4` are rooted iterators.

```cpp
String<char> str = "ACME";
Iterator<String<char>> ::Type it1 = begin(str); // a standard iterator
Iterator<String<char>, Standard>::Type it2 = begin(str); // same as above
Iterator<String<char>, Rooted>::Type it3 = begin(str); // a rooted iterator
Iterator<String<char>, Rooted>::Type it4 = begin(str, Rooted()); // same as above
```

### Assignment 1

**Type Transfer**
Objective

Write a program which does the following:

1. Create an amino acid string of the following sequence: “MQDRVKRPMNAFIVWSRDQRRKMALEN”.
2. Iterate through the sequence and replace all ‘R’ with ‘A’.
3. Create a second string where you count the number of occurrences of each amino acid.
4. Iterate through the latter string and output the frequency table.

Hints

After a few hours browsing through the demos you should be able to solve this.

Solution

In this assignment we practice the use of alphabets, iterators and metafunctions in SeqAn. We start by including the seqan basic header and enter the namespace `seqan` to avoid writing it as a prefix (as we do with the namespace `std` in this example). In the `main` function we first define a a type `TAmincoAcidString` which is a `String<AminoAcid>` (Note the SeqAn naming conventions). Then we define a variable `sourceSeq` of this type and initialize it with a string constant.

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h>
using namespace seqan;

int main()
{
    typedef String<AminoAcid> TAminoAcidString;
    TAminoAcidString sourceSeq = "MQDRVKRPMNAFIVWSRDQRRKMALEN";
}
```

Then we define an iterator type using the SeqAn metafunction `Iterator`. Using the correct iterator we iterate over our amino acid string using the SeqAn functions `begin`, `end`, and `goNext`. In the body of the while loop we use the SeqAn function `value` to access the value the iterator is pointing to. Note that this function returns a reference which allows us to replace the occurrence of all `R`’s with `A`’s. So at this point we have solved parts a) and b) of the assignment.

```cpp
typedef Iterator<TAminoAcidString>::Type TIter;
TIter itEnd = end(sourceSeq);
for (TIter it = begin(sourceSeq); it != itEnd; goNext(it))
{
    if (value(it) == 'R') value(it) = 'A';
    std::cout << value(it) << ',';
}
std::cout << std::endl;
```

In the next part of the code we want to count, how often a specific letter of the alphabet occurs in the string. To obtain the size type of the used alphabet we call the SeqAn metafunction `Size` and define a `String` of that type to hold the counters. The `String` has here basically the same functionality as a STL `vector`. Since alphabets are mapped to a contiguous interval of the natural numbers, we can initialize the counter up to the size of the alphabet which we obtain by a call to the SeqAn metafunction `ValueSize`. We then iterate over the amino acid string and increment the counter for the corresponding letter of the alphabet. In order to know the corresponding natural number of an alphabet letter, we use the SeqAn function `ordValue`. Note the use of the `value` function. In this example one could also use the `operator[]` to write `counter[ordValue(value(it))]++`. 

```cpp
int main()
{
    typedef String<int> counter;
    for (int i = 0; i < ValueSize(AminoAcid); i++)
        counter[i] = 0;
    for (TIter it = begin(sourceSeq); it != itEnd; goNext(it))
    {
        int value = ordValue(value(it));
        counter[value] = counter[value] + 1;
    }
    for (int i = 0; i < ValueSize(AminoAcid); i++)
        std::cout << (char)(i) << ':' << counter[i] << ' ';
```cpp
typedef Size<TAminoAcidString>::Type TSize;
typedef String<TSize> TCounterString;
TCounterString counter;
TSize alphSize = ValueSize<AminoAcid>::VALUE;
resize(counter, alphSize, 0);
for (TIter it = begin(sourceSeq); it != itEnd; goNext(it))
    value(counter, ordValue(value(it))) += 1;

Finally we iterate through the counter String and output the i-th aminoacid (by calling a constructor with
the letter’s ordinal value) ad its frequency.

```
typedef Iterator<TCounterString>::Type TCounterIter;
TCounterIter countIt = begin(counter);
TCounterIter countItEnd = end(counter);
for (TSize pos = 0; countIt != countItEnd; ++countIt, ++pos)
    std::cout << AminoAcid(pos) << ':' << value(countIt) << std::endl;

return 0;
```

The result looks like this:

```
  darwin10.0 : basics//strings
  A:7
  R:0
  N:2
  D:2
  C:0
  Q:2
  E:1
  G:0
  H:0
  I:1
  L:1
  K:2
  M:3
  F:1
  P:1
  S:1
  T:0
  W:1
  Y:0
  V:2
  B:0
  Z:0
  X:0
  *:0
```

### Memory Allocation

Controlling memory allocation is one of the big advantages of C++ compared to other programming languages as for
example Java. Depending on the size of objects and the pattern they are allocated during the program execution, certain
memory allocation strategies have advantages compared to others. SeqAn supports a variety of memory allocation
strategies.

The two functions allocate and deallocate are used in SeqAn to allocate and deallocate dynamic memory. Both
functions take an allocator as an argument. An Allocator is an object that is responsible for allocated memory. The
default implementations of allocate and deallocate completely ignore the allocator but simply call the basic operators new and delete. Although in principle every kind of object can be used as allocator, typically the object that stores the pointer to the allocated memory is used as allocator. For example, if memory is allocated for an Alloc String, this string itself acts as allocator. A memory block should be deallocated using the same allocator object as it was allocated for. The following allocators are available in SeqAn and support the clear function. This function deallocates at once all memory blocks that were previously allocated.

**Simple Allocator**
- General purpose allocator.

**Single Pool Allocator**
- Allocator that pools memory blocks of specific size. Blocks of different sizes are not pooled.

**Multi Pool Allocator**
- Allocator that pools memory blocks. Only blocks up to a certain size are pooled. The user can specify the size limit in a template argument.

The function allocate has an optional argument to specify the intended allocator usage for the requested memory. The user can thereby specialize allocate for different allocator applications. For example, the tag TagAllocateTemp specifies that the memory will only be used temporarily, whereas TagAllocateStorage indicates that the memory will be used in the long run for storing values of a container.

SeqAn also offers more complex allocators which support the function clear. The library predefines some allocator specializations for different uses (see above). Most of these allocators are pool allocators. A pool allocator implements its own memory management. It reserves storage for multiple memory blocks at a time and recycles deallocated blocks. This reduces the number of expensive new and delete calls and speeds up the allocation and deallocation.

**Assignment 2**

**Type** Application

**Objective**
Write a program which compares the runtimes of the Simple Allocator and the Multi Pool Allocator for pool sizes (10, 100, 1000) for allocating and deallocating memory.

**Hint**
For timing the allocation you can use sysTime.

**Solution**

We start in this assignment by including the basic.h SeqAn header and defining two different allocators, one Multi Pool Allocator and one Simple Allocator.

```cpp
#include <iostream>
#include <seqan/basic.h>

using namespace seqan;

int main()
{
    Allocator<MultiPool< > > mpa;
    Allocator<SimpleAlloc< > > sa;
    // store block sizes in an array
    int bs[3] = {10, 100, 1000};
    int runs = 100000;
}```
Given these fixed allocators we allocate now various size blocks, namely of size 10, 100, and 1000. We
repeat the allocation a number of times and then clear the allocated memory. For each of the block sizes
we output the system time needed to allocate and clear the memory.

```cpp
// loop through the different block sizes
for (int i=0; i<3; ++i)
{
    startTime = sysTime();
    for (int j=0; j<runs; ++j)
        allocate(mpa,buf,bs[i],TagAllocateTemp());
    clear(mpa);

    std::cout << "Allocating and clearing " << runs << " times blocks of size ";
    std::cout << bs[i] << " with MultiPool Allocator took " << sysTime() - startTime << std::endl;

    startTime = sysTime();
    for (int j=0; j<runs; ++j)
        allocate(sa,buf,bs[i],TagAllocateTemp());
    clear(sa);

    std::cout << "Allocating and clearing " << runs << " times blocks of size ";
    std::cout << bs[i] << " with Standard Allocator took " << sysTime() - startTime << std::endl;
}

return 0;
```

Running this program results in the following output which shows the advantage of the Multi Pool Allo-
cator:

```
$ darwin10.0 : cd ~/seqan/projects/library/demos/tutorial
$ darwin10.0 : ./basics/allocator
Allocating and clearing 100000 times blocks of size 10 with MultiPool Allocator took 0.00200295
Allocating and clearing 100000 times blocks of size 10 with Standard Allocator took 0.0451179
Allocating and clearing 100000 times blocks of size 100 with MultiPool Allocator took 0.0599239
Allocating and clearing 100000 times blocks of size 100 with Standard Allocator took 0.127033
Allocating and clearing 100000 times blocks of size 1000 with MultiPool Allocator took 0.368732
Allocating and clearing 100000 times blocks of size 1000 with Standard Allocator took 0.560434
```
2.1.45 Writing Tests

This tutorial introduces the SeqAn test system. Currently, there are two kinds of tests in SeqAn: “Normal” tests written in C++ that test C++ classes and functions and “app tests” that test applications. More information is also available in Writing Tests and Writing App Tests.

Library Tests

Creating Tests Skelletons

Use the skel.py script to create a new test, in this case for the module my_module in the sandbox my_sandbox.

```
$ ./util/bin/skel.py test my_module sandbox/my_sandbox
Creating test in ./sandbox/my_sandbox/tests/my_module
Target path is: ./sandbox/my_sandbox/tests/my_module

mkdir('./sandbox/my_sandbox/tests/my_module')

Configuring file.
    Source: ./util/skel/test_template/test.cpp
    Target: ./sandbox/my_sandbox/tests/my_module/test_my_module.cpp

Configuring file.
    Source: ./util/skel/test_template/test.h
    Target: ./sandbox/my_sandbox/tests/my_module/test_my_module.h

Configuring file.
    Source: ./util/skel/test_template/CMakeLists.txt
    Target: ./sandbox/my_sandbox/tests/my_module/CMakeLists.txt
```

The generated files are:
There always is just one .cpp file with the test suite for calling the tests. Usually, there is one test header for each library header under test. There is one CMakeLists.txt that has to be adjusted if new test headers are added.

**Test Suites**

Test suites are collection of tests. They are defined using SEQAN_BEGIN_TESTSUITE and SEQAN_END_TESTSUITE as shown below.

Essentially, these macros when used like SEQAN_BEGIN_TESTSUITE(test_suite_name) { /*...*/ } SEQAN_END_TESTSUITE will expand to a main function: int main() { /*initialization boilerplate*/ /*...*/ /*finalization boilerplate*/; return res; }.

```cpp
#include <seqan/basic.h>
#include <seqan/file.h>
#include "test_my_module.h"

SEQAN_BEGIN_TESTSUITE(test_my_module)
{
    // Call tests.
    SEQAN_CALL_TEST(test_my_module_strings_example1);
}
SEQAN_END_TESTSUITE
```

The generated test .cpp file includes test_my_module which contains the actual tests (see below). The generated file only contains the test test_my_module_strings_example1 which is called using the SEQAN_CALL_TEST macro.

The macro SEQAN_CALL_TEST expands to a function call that calls the test function generated by SEQAN_DEFINE_TEST (see below). The function call will be wrapped in code that will detect if an assertion fails in the test (or the functions called by the test). Note that only SeqAn assertions can be caught, not standard C assertions using assert() from the <cassert> header.

**Tests**

Tests are usually grouped and each group is put into one header file. Tests are declared using SEQAN_DEFINE_TEST.

The generated file looks as follows:

```cpp
#ifndef SANDBOX_MY_SANDBOX_TESTS_MY_MODULE_TEST_MY_MODULE_H_
#define SANDBOX_MY_SANDBOX_TESTS_MY_MODULE_TEST_MY_MODULE_H_
#include <seqan/basic.h>
#include <seqan/sequence.h>

// A test for strings.
SEQAN_DEFINE_TEST(test_my_module_strings_example1)
{
```

my_sandbox/tests/my_module/test_my_module.cpp
Contains the testsuite (the main program that calls the tests).

my_sandbox/tests/my_module/test_my_module.h
A header with tests.

my_sandbox/tests/my_module/CMakeLists.txt
The CMake file for building the test.
using namespace seqan;

// Define some constant test data for comparison...
CharString const STRING1 = "test 1";
CharString const STRING2 = "test 2";

// Append to a string and make equality assertion on the result.
CharString myStr = "test ";
append(myStr, "1");
SEQAN_ASSERT_EQ(STRING1, myStr);

// Demonstration of other assertions.
SEQAN_ASSERT_GT(STRING2, myStr);
SEQAN_ASSERT_GEQ(STRING2, myStr);
SEQAN_ASSERT_LT(myStr, STRING2);
SEQAN_ASSERT_LEQ(STRING2, STRING2);
}
#endif // SANDBOX_MY_SANDBOX_TESTS_MY_MODULE_TEST_MY_MODULE_H_

The macro SEQAN_DEFINE_TEST expands to a function definition. The test functions can be called using the SEQAN_CALL_TEST macro in your test suite (see above).

Note that we do not use a global using namespace seqan but keep this local to the test functions.

CMakeLists.txt File

The relevant lines from the file my_sandbox/tests/my_module/CMakeLists.txt are as follows:

cmake_minimum_required (VERSION 2.8.2)
project (seqan_core_tests_align)
message (STATUS "Configuring my_sandbox/tests/my_module")

# Search SeqAn and select dependencies.
set (SEQAN_FIND_DEPENDENCIES NONE)
find_package (SeqAn REQUIRED)

# Add include directories.
include_directories (${SEQAN_INCLUDE_DIRS})

# Add definitions set by find_package (SeqAn).
add_definitions (${SEQAN_DEFINITIONS})

# Update the list of file names below if you add source files to your test.
add_executable (test_my_module test_my_module.cpp test_my_module.h)

# Add dependencies found by find_package (SeqAn).
target_link_libraries (test_my_module ${SEQAN_LIBRARIES})

# Add CXX flags found by find_package (SeqAn).
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${SEQAN_CXX_FLAGS}")

# Register with CTest
add_test (NAME test_test_align COMMAND $<TARGET_FILE:test_my_modulen>)

When we add a new test header test_my_module_header.h, we would change the line with add_executable() to read:
add_executable (test_my_module test_my_module.cpp test_my_module.h test_my_module_header.h)

When next building the target `test_my_module` via `make` or an IDE, CMake will run automatically. Now, the build system is aware of the new header file `test_my_module_header.h` and will rebuild the test binary if the file changes.

**Building And Running Tests**

Now, go to your build directory and re-run CMake so the Makefiles (or your IDE’s project files) include the target for the test.

```bash
$ cmake .
...
$ make test_my_module
...
$ ./sandbox/holtgrew/tests/my_module/test_my_module
TEST SUITE test_my_module
SEQAN_ENABLE_DEBUG == 1
SEQAN_ENABLE_TESTING == 1
SEQAN_ENABLE_CHECKPOINTS == 0
SEQAN_CXX_FLAGS == "SEQAN_CXX_FLAGS_NOT_SET"
test_my_module_strings_example1 OK
**************************************
Total Check Points : 0
Found Check Points : 0
Lost Check Points : 0
--------------------------------------
Total Tests: 1
Skipped: 0
Errors: 0
**************************************
```

**Tests in apps folders**

It is also possible to write normal tests inside apps. One example is the `extras/apps/param_chooser/param_chooser` app.

To add a test to your app, you have to (1) add a test `.cpp` file and (2) register it in the `CMakeLists.txt` file.

**Writing the test program**

The test program looks the same as above, but defines the precompiler symbol `SEQAN_ENABLE_TESTING` as 1. This has to happen at the top of the file (say the name of the file is `test_my_app_funcs.cpp`).

```cpp
# undef SEQAN_ENABLE_TESTING
# define SEQAN_ENABLE_TESTING 1

#include <seqan/basic.h>
#include <seqan/file.h>

SEQAN_DEFINE_TEST(test_my_app_funcs_hello)
{
    SEQAN_FAIL("Hello, tester!");
}

SEQAN_BEGIN_TESTSUITE(test_my_app_funcs)
```
SEQAN_CALL_TEST(test_my_app_funcs_hello);
SEQAN_END_TESTSUITES

Registering in CMakeLists.txt

The following lines have to be added to CMakeLists.txt.

```cmake
add_executable (test_my_app_funcs test_my_app_funcs.cpp)
target_link_libraries (test_my_app_funcs ${SEQAN_LIBRARIES})
add_test (NAME test_test_my_app_funcs COMMAND <TARGET_FILE:test_my_app_funcs>)
```

Next Steps

The best way to learn about the testing system is to look at the existing test suites. Good examples are test_score and test_random. Note that other test suites might not be very clean since they were ported from the old test system and are not completely cleaned up yet.

Have a look at the Writing Tests for more examples.

Assertions

You can make assertions on the called functions using the SEQAN_ASSERT* macros, e.g. SEQAN_ASSERT, SEQAN_ASSERT_EQ. For each assertion macro, there is one macro that has to be passed a message and optionally parameters, e.g. SEQAN_ASSERT_MSG, SEQAN_ASSERT_EQ_MSG.

Assertions can also be used in your library and application code. When compiled in Debug mode, the assertions are enabled. They are disabled in Release or RelWithDebInfo mode.

Checks

There also is a macro called SEQAN_CHECK that creates an assertion that is enabled regardless of whether debugging is enabled. It only makes sense to use the SEQAN_CHECK macro in library or application code, not in tests. There are no variants of SEQAN_CHECK for comparisons. Also see the macro SEQAN_FAIL.

App Tests

For writing app tests, see Writing App Tests.

The SeqAn tutorials are the best way to get started with learning how to develop using SeqAn. In contrast, the API Documentation gives more comprehensive but less verbose documentation about the library while the How-Tos are strictly task driven and narrower in scope.

The main audience of the tutorials are graduate students and professionals who want to learn how to use SeqAn. Previous programming knowledge is required, knowledge of C++ is recommended.
Introduction

These tutorials show you how to get started with SeqAn, including the installation. Then, you can learn about the background and motivation of SeqAn. You should then definitely start your engines and read the A First Example tutorial to see an example highlighting many important concepts in the SeqAn library.

**Getting Started**  This tutorial will walk you through the installation of SeqAn and its dependencies. Then, you will create your first minimal SeqAn application!

**A First Example**  This tutorial gives practical examples and applications of the most important basic techniques. You should read this tutorial if you are starting out with SeqAn.

**Background and Motivation**  This tutorial gives an overview over the design aims and principles of SeqAn and a motivation for the employed mechanisms.

We highly recommend you to follow the Getting Started instructions if you are starting out with SeqAn. Note that it is also possible to use SeqAn strictly as a library with your own build system. The article *Integration with your own Build System* contains detailed information about this.

A Stroll Through SeqAn

**Sequences**

**Sequences**  This tutorial introduces you to the basics of fundamental concept of sequences, namely Strings and Segments.

**Alphabets**  This tutorial introduces you to SeqAn’s alphabets, or in other words, the contained types of sequences.

**String Sets**  StringSets  This tutorial introduces you to SeqAn’s StringSet, an efficient data structure to store a set of sequences.

**Sequences In-Depth**  In this tutorial you will learn how to optimize the work with sequences, using different specializations of Strings and different overflow strategies for capacity changes.

**Iterators**

**Iterators**  This tutorial explains how to use iterators in SeqAn, illustrated on containers.

**Alignments**

**Alignment Representation**  This section of the tutorial introduces you to the data structures that are used to represent alignments in SeqAn.

**Pairwise Sequence Alignment**  In this part of the tutorial we demonstrate how to compute pairwise sequence alignments in SeqAn. It shows the use of different scoring schemes, and which parameters can be used to customize the alignment algorithms.

**Multiple Sequence Alignment**  In the last section of this tutorial we show how to compute multiple sequence alignments in SeqAn using a scoring matrix.

**Indices**

**Indices**  This tutorial introduces you to the various indices in SeqAn like extended suffix arrays or k-mer indices.
Index Iterators  This tutorial introduces you to the various index iterators with which you can use indices as if traversing search trees or tries.

Q-gram Index  This tutorial introduces you to SeqAn’s q-gram index.

Pattern Matching

Pattern Matching  This section of the tutorial introduces you to the algorithms in SeqAn for exact and approximate pattern matching.

Graphs

Graphs  This section of the tutorial introduces you to the graph type in SeqAn. We will discuss the various graph specializations and show you how to create directed and undirected graphs as well as HMMs, how to store additional information for edges and vertices and last but not least how to apply standard algorithms to the graphs.

I/O Basics

Basic Sequence I/O  Basic Sequence I/O This tutorial explains how to use the high-level API for reading and writing sequence files.

Indexed FASTA I/O  Indexed FASTA I/O This tutorial explains how to use FASTA index files for quick random access within FASTA files: read contigs or just sections without having to read through whole FASTA file.

Basic SAM and BAM I/O  Basic SAM and BAM I/O This tutorial explains how to use the high-level API for reading and writing SAM and BAM files.

VCF I/O  VCF I/O This tutorial explains how to use the high-level API for reading and writing VCF files.

BED I/O  BED I/O This tutorial explains how to use the high-level API for reading and writing BED files.

GFF and GTF I/O  GFF and GTF I/O This tutorial explains how to use the high-level API for reading and writing GFF and GTF files.

Modifiers

Modifiers  Modifiers Modifiers can be used to change the elements of a container without touching them. Here you will see, what modifiers are available in SeqAn.

Randomness

Randomness  This chapter shows module random that provides pseudo random number generation functionality.

Seed-And-Extend

Seed-and-Extend  In this part of the tutorial we will introduce SeqAn’s seed class, demonstrate seed extension and banded alignment with seeds, and finally show the usage of seed chaining algorithms.
Parsing Command Line Arguments

Parsing Command Line Arguments  Parsing Command Line Arguments In this tutorial, you will learn how to use the ArgumentParser class for parsing command line arguments.

Genome Annotations

Genome Annotations  You will learn how to work with annotations in SeqAn and analyzing them, using the annotationStore which is part of SeqAn’s FragmentStore.

More I/O

These tutorials explain how to use the I/O functionality in SeqAn beyond the basic sequence, SAM/BAM and indexed FASTA I/O from above. The tutorials are targeted at developers that either want to use the lower level I/O routines in SeqAn or write their own parsers. We recommended to start out reading the I/O Overview and then jump to the chapter that interests you most. In this Section we introduce the three main techniques of programming in SeqAn, namely the ‘‘global function interface’’, the use of ‘‘Metafunctions’’, and the concept of ‘‘Template subclassing’’.

I/O Overview  This article gives an overview of the I/O functionality in SeqAn.

After reading, you will have a better understanding of the different bits in this section of the library. The following tutorials introduce the lower level I/O routines for specific file formats.

Sequence File I/O  This tutorial explains the RecordReader- and Stream-based interface for reading sequence files.

SAM and BAM I/O  This tutorial explains the lower level API for reading and writing SAM and BAM files.

Read the following tutorials to learn how to write your own I/O routines.

File I/O  This chapter shows how to use the file I/O facilities of SeqAn, including streams, compressed streams and memory mapped files.

Lexical Casting  This tutorial explains the lexicalCast and lexicalCast2 functions that allow to convert strings representing numbers into their numeric values.

Parsing  In this part of the tutorial, you will be introduced to the parsing and tokenizing functionality using the RecordReader class. You will get the necessary information to write your own file parsers.

Advanced Tutorials

Fragment Store  This tutorial shows how to use the fragment store which is a database for read mapping, sequence assembly or gene annotation. It supports to read/write multiple read alignments in SAM or AMOS format and access and modify them. It supports to read/write gene annotations in GFF/GTF and UCSC format, to create custom annotation types, and to traverse and modify the annotation tree.

Simple RNA-Seq  In this tutorial you will learn how to implement a simple RNA-Seq based gene quantification tool, that computes RPKM expression levels based on a given genome annotation and RNA-Seq read alignments.

Simple Read Mapping  This tutorial shows how to implement a simple read mapping program based on the SWIFT filter and online Hamming finder for verification.

Mini Bowtie  Mini-Bowtie is a very basic read aligner that is inspired by the well known Bowtie program [LTP+09]. It serves as an example to show that you can write sophisticated programs with SeqAn using few lines of code.

Journaled Set  In this tutorial we demonstrate how you can handle multiple large sequence in main memory while the data structures themself support a certain parallel sequence analysis.
**KNIME Nodes**  Here you can learn how to use SeqAn apps in KNIME.

**Developer's Corner**

First, congratulations on becoming an official SeqAn developer! After you went through the tutorials and before you actually start to develop your own application with SeqAn you might want to learn Writing Tests and read about the API documentation. In addition, we follow a SeqAn specific SeqAn Style Guides. Information like this can be found on the section site. There are plenty of information completing your knowledge about SeqAn so have a look!

**Frequently used Software Techniques**

We assume that the user is acquainted with the basic data types of SeqAn, the introductory example and the demo programs. Also you should be acquainted with the STL and template programming. In this Section we introduce the three main techniques of programming in SeqAn, namely the global function interface, the use of Metafunctions, and the concept of Template subclassing.

**Basic Techniques**  Here we remind you of the basics of template programming and the use of the STL.

**Metafunctions**  In this section you find an introductory explanation how Metafunctions are used in SeqAn to obtain information about data types used which will only be instantiated at compile time.

**Generic Programming**  In this section you find a short example that illustrates the power of template subclassing.

**Global Function Interface**  In this section you find a useful piece of code that shows you the flexibility of the global function interface.

### 2.2 How Tos

After reading the Tutorial we recommend to read some of the following HowTos:

<table>
<thead>
<tr>
<th>ToC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
</tr>
<tr>
<td>• Accessing Index Fibres</td>
</tr>
<tr>
<td>– Overview</td>
</tr>
<tr>
<td>– Creation</td>
</tr>
<tr>
<td>– Access</td>
</tr>
</tbody>
</table>

#### 2.2.1 Accessing Index Fibres

**Overview**

Basically each index consists of a set of tables, called fibres. The set of available fibres of an index `Index<TText, TSpec>` depends on the index specialization TSpec.
<table>
<thead>
<tr>
<th>Fibres</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EsaText</td>
<td>The original text the index should be based on. Can be either a sequence or a StringSet.</td>
</tr>
<tr>
<td>EsaSA</td>
<td>The suffix array stores the begin positions of all suffixes in lexicographical order.</td>
</tr>
<tr>
<td>EsaLcp</td>
<td>The lcp table stores at position i the length of the longest common prefix between suffix with rank i and rank i+1.</td>
</tr>
<tr>
<td>EsaChildtab</td>
<td>See [AKO04]</td>
</tr>
<tr>
<td>EsaBwt</td>
<td>The Burrows-Wheeler table stores at position i the character left of the suffix with rank i.</td>
</tr>
<tr>
<td>EsaRawText</td>
<td>Virtually concatenates all strings of the EsaText fibre.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WOTDIndexFibres</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WotdText</td>
<td>The original text the index should be based on.</td>
</tr>
<tr>
<td>WotdSA</td>
<td>The suffix array stores the begin positions of all suffixes in lexicographical order.</td>
</tr>
<tr>
<td>WotdDir</td>
<td>See [GKS03].</td>
</tr>
<tr>
<td>WotdRawText</td>
<td>Virtually concatenates all strings of the WotdText fibre.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DfiIndexFibres</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DfiText</td>
<td>The original text the index should be based on.</td>
<td>First template argument of the Index. Can be either a sequence or a StringSet.</td>
</tr>
<tr>
<td>DfiSA</td>
<td>The suffix array stores the begin positions of all suffixes in lexicographical order.</td>
<td>String over the SAValue type of the index.</td>
</tr>
<tr>
<td>DfiDir</td>
<td>See [GKS03].</td>
<td>String over the Size type of the index.</td>
</tr>
<tr>
<td>DfiRawText</td>
<td>Virtually concatenates all strings of the DfiText fibre.</td>
<td>ContainerConcept over the alphabet of the text.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QGramIndexFibres</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>QGramText</td>
<td>The original text the index should be based on.</td>
<td>First template argument of the Index. Can be either a sequence or a StringSet.</td>
</tr>
<tr>
<td>QGramShape</td>
<td>The q-gram Shape.</td>
<td>Specified by the first template argument of IndexQGram.</td>
</tr>
<tr>
<td>QGramSA</td>
<td>The suffix array stores the begin positions of all suffixes in lexicographical order.</td>
<td>String over the SAValue type of the index.</td>
</tr>
<tr>
<td>QGramDir</td>
<td>The directory maps q-gram hash values to start indices in the QGramSA fibre.</td>
<td>String over the Size type of the index.</td>
</tr>
<tr>
<td>QGramCounts</td>
<td>Stores numbers of occurrences per sequence of each q-gram in pairs (seqNo,count), count&gt;0.</td>
<td>String over Pair of the Size type of the index.</td>
</tr>
<tr>
<td>QGramCountsDir</td>
<td>The counts directory maps q-gram hash values to start indices in the QGramCounts fibre.</td>
<td>String over the Size type of the index.</td>
</tr>
<tr>
<td>QGramBucketMap</td>
<td>Used by the OpenAddressingQGramIndex index to store the hash value occupancy in the QGramDir fibre.</td>
<td>String over the Value type of the shape.</td>
</tr>
<tr>
<td>QGramRawText</td>
<td>Virtually concatenates all strings of the QGramText fibre.</td>
<td>ContainerConcept over the alphabet of the text.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pizza &amp; Chili Index Fibres</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PizzaChiliText</td>
<td>The original text the index should be based on.</td>
<td>First template argument of the Index. Must be a sequence (no support for StringSet).</td>
</tr>
<tr>
<td>PizzaChiliCompressed</td>
<td>Specialization dependent data structure to store the compressed index.</td>
<td>Depends on the compressed index.</td>
</tr>
</tbody>
</table>

The first column in each table above contains the tags to select the corresponding fibre. Most of these tags are aliases for the same tag, e.g. EsaSA, QGramSA, ... are aliases for FibreSA. If you write an algorithm that is generic in the type of index can use FibreText to specify the fibre that stores the index text.

2.2. How Tos
Creation

In most cases you don’t need to create the fibres of an index by hand. Most algorithms and data structures create them automatically, e.g. Finder or VSTreeIterator. If you want to specify a certain index construction algorithm, have to recreate a fibre or manually access a fibre you can recreate or create on-demand a fibre by indexCreate and indexRequire. If your algorithm should behave differently depending on the presence or absence of a fibre (and the fibre should then not be created), you can test for presence by indexSupplied.

Access

The type of each fibre can be determined by the metafunction Fibre. To access a fibre you can use the function getFibre whose return type is the result of Fibre. The second argument of both functions is a tag to select a specific fibre. See the first column in the tables above. One fibre in every index is the text to be indexed itself. This fibre can be assigned during the construction. For the ease of use, there exist shortcuts to access frequently used fibres:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Expands To ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexBucketMap(index)</td>
<td>getFibre(index, FibreBucketMap())</td>
</tr>
<tr>
<td>indexBwt(index)</td>
<td>getFibre(index, FibreBwt())</td>
</tr>
<tr>
<td>indexChildtab(index)</td>
<td>getFibre(index, FibreChildtab())</td>
</tr>
<tr>
<td>indexCounts(index)</td>
<td>getFibre(index, FibreCounts())</td>
</tr>
<tr>
<td>indexCountsDir(index)</td>
<td>getFibre(index, FibreCountsDir())</td>
</tr>
<tr>
<td>indexLcp(index)</td>
<td>getFibre(index, FibreLcp())</td>
</tr>
<tr>
<td>indexRawText(index)</td>
<td>getFibre(index, FibreRawText())</td>
</tr>
<tr>
<td>indexSA(index)</td>
<td>getFibre(index, FibreSA())</td>
</tr>
<tr>
<td>indexShape(index)</td>
<td>getFibre(index, FibreShape())</td>
</tr>
<tr>
<td>indexText(index)</td>
<td>getFibre(index, FibreText())</td>
</tr>
</tbody>
</table>

and to access a single values:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Expands To ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwtAt(pos, index)</td>
<td>indexBwt(index)[pos]</td>
</tr>
<tr>
<td>childAt(pos, index)</td>
<td>indexChildtab(index)[pos]</td>
</tr>
<tr>
<td>dirAt(pos, index)</td>
<td>indexDir(index)[pos]</td>
</tr>
<tr>
<td>lcpAt(pos, index)</td>
<td>indexLcp(index)[pos]</td>
</tr>
<tr>
<td>rawtextAt(pos, index)</td>
<td>indexRawText(index)[pos]</td>
</tr>
<tr>
<td>saAt(pos, index)</td>
<td>indexSA(index)[pos]</td>
</tr>
<tr>
<td>textAt(pos, index)</td>
<td>indexText(index)[pos]</td>
</tr>
</tbody>
</table>

Please note that textAt can also be used if the index text is a StringSet. pos can then be a SAValue.

ToC

Contents

- Automating Tests With CTest
  - For Linux and Mac OS X
  - For Windows
  - Sparse Checkouts

2.2.2 Automating Tests With CTest

The dashboard lives at the SeqAn CDash site.
For Linux and Mac OS X

Create ~/Nightly where everything will take place and check out the trunk:

```bash
cd ~
mkdir Nightly
cd Nightly
svn co http://svn.seqan.de/seqan/trunk seqan-trunk
```

Now, get the build scripts:

```bash
cp seqan-trunk/misc/ctest/run_nightly.sh .
cp seqan-trunk/misc/ctest/Seqan_Nightly.cmake.example Seqan_Nightly.cmake
cp seqan-trunk/util/cmake/CTestConfig.cmake seqan-trunk/
```

Adjust the build name and site name in Seqan_Nightly.cmake. Now, test the setup by running:

```bash
chmod u+x run_nightly.sh
./run_nightly.sh
```

Add run_nightly.sh to your nightly cron jobs:

```bash
crontab -e
```

Example crontab file:

```bash
# min hour mday month wday command
05 1 * * * sh -l ${HOME}/Nightly/run_nightly.sh > /dev/null
```

For Windows

Create Nightly in your home directory where everything will take place and check out the trunk:

```bash
cd /D %HOME%
mkdir Nightly
cd Nightly
svn co http://svn.seqan.de/seqan/trunk seqan-trunk
```

Now, get the build scripts:

```bash
copy seqan-trunk\misc\ctest\run_nightly.sh .
copy seqan-trunk\misc\ctest\Seqan_Nightly.cmake.example Seqan_Nightly.cmake
copy seqan-trunk\util\cmake\CTestConfig.cmake seqan-trunk\
```

Adjust the build name and site name in Seqan_Nightly.cmake. Now, test the setup by running:

```bash
run_nightly.bat
```

Add run_nightly.bat to nightly Scheduled Tasks of Windows (analogously to the CTest Tutorial):

1. Open “Scheduled Tasks” from Control Panel.
2. Select Add Scheduled Task“
3. Select Next to select command.“
4. Click Browse... and select \`\`run_nightly.bat.
5. Click Next and select name and repetition date. Repetition date for Nightly dashboards should be Daily.“
6. Click Next and select time to start the dashboard.“
7. Click Next and select Open advanced properties... to fine tune the scheduled task."
8. Select Next and type password of the user."
9. Task is created. The Advanced Properties dialog should open."
10. In advanced properties, specify full command name. This is very important that you use double quotes in case you have space in your path."
11. Select ‘Ok, which will ask for password again.’
12. The new task should be created."

**Sparse Checkouts**

This is only necessary/interesting if you are a developer with read permissions to more than core, extras, and your own sandbox. You can checkout only a subset of the directories in the repository using a Subversion feature called sparse directories.

Also consult the Subversion reference on sparse checkouts.

```bash
#sh
svn co --depth immediates https://svn.seqan.de/seqan/trunk seqan-trunk-sparse
cd seqan-trunk-sparse
svn update --set-depth infinity build core docs extras misc util
svn update --set-depth files sandbox
```

2.2.3 Choosing The JRE Version

In order to change the JRE to be used by KNIME go to Eclipse Preferences and select the Java menu.

Afterwards you can add the right JRE. Under MacOS you choose the entry MacOS X VM

then press next and select the right path, which should be /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home

as shown here:

Press Finish and the right JRE will be added.

Afterwards you have to set the compiler options. In order to do so go to Project and select Properties.

No select Java Compiler and select the correct JRE at the Compiler compliance level:

If you run the project now KNIME should start without problems.
2.2.4 Computing Positions In Clipped Alignments

This page describes how to compute view and source positions in an unclipped and clipped Align.

Position Computation Overview

There are four coordinate systems related to each gap object. One can consider the positions with and without gaps, both with and without clipping. The following picture and list show the easiest transformations between the coordinate systems.

1. Translate between view (gapped clipped) position and source (ungaped unclipped) position using the functions toSourcePosition and toViewPosition.
2. Translate between clipped and unclipped gapped position by adding/subtracting clippedBeginPosition of the gaps object.
3. Translate between clipped ungapped and unclipped ungapped position by adding/subtracting beginPosition of the gaps object.

All other transformations are most easily done following one of the paths from the picture above.
Properties for com.genericworkfllownodes.knime.tests

Java Compiler

- Enable project specific settings
- Configure Workspace Settings...

JDK Compliance

- Use compliance from execution environment 'JavaSE-1.6' on the Java Build Path
- Compiler compliance level: 1.6
- Use default compliance settings
- Generated .class files compatibility: 1.6
- Source compatibility: 1.6
- Disallow identifiers called 'assert': Error
- Disallow identifiers called 'enum': Error

Classfile Generation

- Add variable attributes to generated class files (used by the debugger)
- Add line number attributes to generated class files (used by the debugger)
- Add source file name to generated class file (used by the debugger)
- Preserve unused (never read) local variables
- Inline finally blocks (larger class files, but improved performance)

Restore Defaults  Apply  Cancel  OK
An Example

The following extensive example shows how to practically translate between the coordinate systems.

// Demo program for clipping with Gaps objects.

#include <iostream>
#include <seqan/sequence.h>
#include <seqan/align.h>

int main()
{
    typedef seqan::Position<seqan::Gaps<seqan::CharString> >::Type TPos;

    // Create sequence variable and gaps based on sequence.
    seqan::CharString seq("ABCDEFGHIJ");
    seqan::Gaps<seqan::CharString> gaps(seq);

    // Insert gaps, the positions are in (clipped) view space.
    insertGaps(gaps, 2, 2);
    insertGap(gaps, 6);
    insertGap(gaps, 10);

    // Print to stdout.
    std::cout << "gaps\t" << gaps << "\n"
               << "seq \t" << seq << "\n\n";

    // Print the begin and end positions in sequence space and the clipped
    // begin and end positions in gap space. We have no clipping, so no
    // surprises here.
    std::cout << "beginPosition(gaps) == " << beginPosition(gaps) << "\n"
               << "endPosition(gaps) == " << endPosition(gaps) << "\n"
               << "clippedBeginPosition(gaps) == " << clippedBeginPosition(gaps) << "\n"
               << "clippedEndPosition(gaps) == " << clippedEndPosition(gaps) << "\n\n";

    // Now, clip the alignment and again print the gaps, sequence and begin/end
    // positions. Note that the clipping positions are relative to the unclipped
    // view.
setClippedBeginPosition(gaps, 3);
setClippedEndPosition(gaps, 10);

std::cout << "gaps\t" << gaps << "\n"
<< "seq \t" << infix(seq, beginPosition(gaps), endPosition(gaps)) << "\n"

// We can translate between the (clipped) gapped position (aka view) and
// the unclipped ungapped positions (aka) source using toSourcePosition()
// and toViewPosition(). Note that because of projection to the right of
// gaps, these operations are not symmetric.
std::cout << "4 view position => " << toSourcePosition(gaps, 4) << " source position\n"
<< "2 source position => " << toViewPosition(gaps, 2) << " view position\n"

// Translating between clipped gapped and unclipped gapped position can
// be done by adding/subtracting clippedBeginPosition(gaps).
std::cout << "3 clipped gapped => " << 3 + clippedBeginPosition(gaps) << " unclipped gapped\n"
<< "6 unclipped gapped => " << 5 - clippedBeginPosition(gaps) << " clipped gapped\n"

// Translating between clipped ungapped and unclipped ungapped position can
// be done by adding/subtracting beginPosition(gaps). Since there are no
// gaps, this operation is symmetric.
std::cout << "3 clipped ungapped => " << 3 + beginPosition(gaps) << " unclipped ungapped\n"
<< "5 unclipped ungapped => " << 5 - beginPosition(gaps) << " clipped ungapped\n"

// Translating between gapped clipped position and ungapped clipped
// position and between gapped unclipped and ungapped unclipped positions
// has to be done using the translations above.
std::cout << "3 clipped gapped => " << toSourcePosition(gaps, 3) - beginPosition(gaps) << " clipped gapped\n"
<< "4 unclipped ungapped => " << toViewPosition(gaps, 4) + clippedBeginPosition(gaps) << " unclipped ungapped

return 0;
}
2.2.5 Getting Started with CUDA

Requirements

In order to follow this HowTo, you need:

- git to download the sources
- cmake to build the projects
- the CUDA toolkit >= v5.0 to compile the CUDA demos
- a CUDA-capable GPU with SM architecture >= 2.0 to run the CUDA demos

Refer to Getting Started for detailed SeqAn installation instructions.

Getting the source code

CUDA acceleration resides in the develop branch of SeqAn, hosted on GitHub. Execute the following command to get the last sources:

$ git clone -b develop https://github.com/seqan/seqan.git SeqAn

Compiling the demos

Hello CUDA

Let us first set up the build system:

$ mkdir SeqAn-Builds && cd SeqAn-Builds
$ cmake ../SeqAn -DCMAKE_BUILD_TYPE=Release

Now we can compile and execute our CUDA hello world:

$ make demo_cuda_hello
$ bin/demo_cuda_hello
Hello CUDA!

Important: Some users experienced compilation problems on Mac OS X. If the compilation fails with: **clang: error: unsupported option '-dumpspecs**, then you need to manually create links to gcc 4.4 in the nvcc directory. If you for example installed gcc-44 via MacPorts in /opt/local/bin you can create these links as follows:

$ ln -s /opt/local/bin/gcc-mp-4.4 /usr/local/cuda/bin/gcc
$ ln -s /opt/local/bin/g++-mp-4.4 /usr/local/cuda/bin/g++

MMap String

Now let's try the MMap String demo. This demo maps a text file in memory, copies it on the device and uses Thrust to count the number of occurrences of a given character into the file.

$ make demo_cuda_mmap
$ echo "THIS IS A TEST" > test.txt
$ bin/demo_cuda_mmap test.txt T
3
**FM-index counting**

The **FM-index counting demo** builds an FM-index over a static text. Given a set of patterns, the program counts - both on the host and on the device - the total number of occurrences of all patterns in the text.

```
$ make demo_cuda_count
$ bin/demo_cuda_count ACGTACGTACGTACGT ACGT GTA
CPU Occurrences: 7
GPU Occurrences: 7
```

### 2.2.6 Setup Your Editor

This page describes how to adjust your editor to the *SeqAn C++ Style*.

**Eclipse CDT**

- Open preference page of Eclipse:
  - [Windows/Linux] Menu -> Windows -> Preferences
  - [Mac OS] Menu -> Eclipse -> Preferences (Cmd-Key + ,)
- Within the *Preferences* select ‘C/C++’ and next select ‘Code Style’.
- Note: If you have not prepared your own code style scheme, than select on of the existing built-in schemes and extend it according to the new SeqAn style guides.
- Push ‘Edit’ and select the tab ‘Indention’ in the appeared *Profile* window.
- Under the group ‘Tab policy’ select ‘Spaces only’ in the drop down menu.
- Set ‘Indentation size’ to 4.
- Set ‘Tab size’ to 4.
- Rename the profile to your preferred name, e.g. *K&R - SeqAn*, if the changes you have made rely on the *K&R [built-in]* profile.
- Press ‘OK’

The new profile appears in the drop down menu and is selected automatically to be active.
Visual Studio

- Open ‘Tools’ in the Visual Studio Menu and click on ‘Options’
- Click on ‘Text Editor’, then select ‘C/C++’ and open the ‘Tabs’ settings
- Set both ‘Tab size’ and ‘Indent size’ to 4 and make sure ‘Insert spaces’ is selected below.

XCode

**Xcode 3**

- Open Xcode->Preferences... in the Xcode menu.
- Open the indentation settings.
- Choose Indentation in the top scroll panel (scroll right).
- Choose to insert spaces instead of tabs.
- Uncheck “Tab key inserts tag, not spaces”.
- Set tab and indent width to 4.

**Xcode 4**

- Open Xcode->Preferences... in the Xcode menu.
- Open the indentation settings.
- Choose Text Editing in the top panel.
2.2. How Tos
Choose to insert spaces instead of tabs.
Uncheck “Tab key inserts tag, not spaces”.
Under “Prefer indent using;” choose “Spaces”.
Set tab and indent width to 4.

**Vim**

Append the following to your ~/.vimrc.

```vim
set nosmartindent " smartindent (filetype indenting instead)
\set autoindent       " autoindent (should be overwrote by cindent or filetype indent)
set cindent         " do c-style indenting
set softtabstop=4  " unify
set shiftwidth=4    " unify
set tabstop=4       " unify
set copyindent     " but above all -- follow the conventions laid before us
set expandtab      " we do not want to type tabs
```

---

### 2.2.7 Efficiently Importing Millions Of Sequences

#### Memory Mapped Files

The fastest way to import tons of sequences in Fasta/Fastq/GSeq/... file format is to avoid slow C++ I/O streams and instead map the file directly into memory. This can be done via the MMapString which uses memory mapping of the operating system or via the ExternalString which emulates memory mapping by doing the paging by-hand. Most commonly used file formats concatenate sequences separated by a delimiter, e.g. >, @, line-break, that marks the begin of each sequence. In SeqAn there is also a data structure that represents multiple sequences using one concatenation string and the begin positions of each sequence, the ConcatDirectStringSet. We therefore defined the type MultiSeqFile as an alias for a ConcatDirectStringSet using a single MMapString.

In the next example we are going to open a sequence file, recognize its format, split the file into sequence fractions and import each sequence, its quality values and id.

```cpp
#define SEQAN_PROFILE // enable time measurements
#include <seqan/file.h>
#include <iostream>
using namespace seqan;
```

First we associate our sequence file with the memory mapped string underlying the ConcatDirectStringSet using open.
```c
int main (int argc, char const * argv[]) {
    SEQAN_PROTIMESTART(loadTime);

    MultiSeqFile multiSeqFile;
    if (argc < 2 || !open(multiSeqFile.concat, argv[1], OPEN_RDONLY))
        return 1;

    Next we guess the file format of the single concatenation string and store the result in a AutoSeqFormat object, which is used subsequently to select the right import function. split expects a ConcatDirectStringSet and divides the underlying string into sequence fragments separated by a file format specific delimiter.

    AutoSeqFormat format;
    guessFormat(multiSeqFile.concat, format);
    split(multiSeqFile, format);

    After calling split the multiSeqFile StringSet represents the sequence fragments and can be used to reserve memory for the StringSets that store sequences and ids.

    unsigned seqCount = length(multiSeqFile);
    StringSet<String<Dna5Q> > seqs;
    StringSet<CharString> seqIDs;

    reserve(seqs, seqCount, Exact());
    reserve(seqIDs, seqCount, Exact());

    The main loop iterates over each sequence fragment and uses the functions assignSeq, assignQual and assignSeqId to extract sequence data, qualities and id. The quality values are encoded in ASCII and have to be converted into integer values between 0 and 62 before assigning it to a Dna5Q character via assignQualityValue.

    String<Dna5Q> seq;
    CharString qual;
    CharString id;

    for (unsigned i = 0; i < seqCount; ++i) {
        assignSeq(seq, multiSeqFile[i], format);  // read sequence
        assignQual(qual, multiSeqFile[i], format);  // read ascii quality values
        assignSeqId(id, multiSeqFile[i], format);  // read sequence id

        // convert ascii to values from 0..62
        // store dna and quality together in Dna5Q
        for (unsigned j = 0; j < length(qual) && j < length(seq); ++j)
            assignQualityValue(seq[j], (int)(ordValue(qual[j]) - 33));

        // we use reserve and append, as assign is not supported
        // by StringSet<..., Owner<ConcatDirect<> > >
        appendValue(seqs, seq, Generous());
        appendValue(seqIDs, id, Generous());
    }

    Finally we output the number of imported sequences, the overall runtime and the first 10 sequences in Fasta format.

    std::cout << "Loading " << seqCount << " sequences took " << SEQAN_PROTIMEDIFF(loadTime);
    std::cout << " seconds." << std::endl << std::endl;
    for (unsigned i = 0; i < seqCount && i < 10; ++i) {
        std::cout << '>' << seqIDs[i] << std::endl;
        std::cout << seqs[i] << std::endl;
    }
}
```

2.2. How Tos 457
Program Output

```bash
$ cd build/Release
$ make efficiently_import_sequences
[...]
$ ./core/demos/howto/efficiently_import_sequences reads.fq
Loading 1000000 sequences took 4.82109 seconds

>HWI-EAS299_3_30MAPAAXX:6:1:1561:1481/1
GTTTATTTCACCTCCTTTACTTGTAGTCCAGGCGGTA
>HWI-EAS299_3_30MAPAAXX:6:1:1561:1481/2
AAAGAATTTAAATATTTCCTTAATAAGGCACGCCGTT
>HWI-EAS299_3_30MAPAAXX:6:1:1703:1976/1
GTTTTGATGTACAACGCCGTTACAGGTATAGTGAGAG
TTCATAAAATAAAACCTCCAGAATAAGGAACATAAGAG
>HWI-EAS299_3_30MAPAAXX:6:1:1638:1932/1
GAAATTGGTAGTTATTCGCTCTTGCAACACTTTTT
CACCCATACTATTAAAGCAAGCATCGGGAAAAGTAAAT
>HWI-EAS299_3_30MAPAAXX:6:1:1726:1928/1
GCATAATGCAAAGGGTTAGTATATGATTTTTAGTATG
GAGACGACAACTCCCTCGGGAACACTAAAGGCTCGGTAT
>HWI-EAS299_3_30MAPAAXX:6:1:720:1208/1
GCATATTCTATAAATGCTAAGCATAAAAATAATTTTC
>HWI-EAS299_3_30MAPAAXX:6:1:720:1208/2
TGCGTGGTTACCATTTAGACAGGGTCACAATTTTCA
```

Remarks

- We intentionally use `appendValue` to fill the StringSets as for some applications it is more memory efficient to use a `ConcatDirectStringSet` to store imported sequences and ids. The `ConcatDirectStringSet` consists of only one `String` concatenating all sequences and a `String` containing the begin positions which induce less overhead compared to storing millions of single `Strings` separately on heap with their own begin, end and capacity information.

- Although not visible in the example, the import functions can of course also import large sequences spanning multiple lines in various formats.

Fragment Store

The whole program above is condensed into the function `loadReads` working on a `FragmentStore`. An example for this function is given in `Filtering Similar Sequences`.

458 Chapter 2. API Documentation
2.2.8 Filtering Similar Sequences

Using Swift

In the next example we are going to use the Swift filter to efficiently find pairs of similar reads. The Swift algorithms searches for so-called epsilon matches, local alignments, of two sequences with an error rate below a certain epsilon threshold.

The Swift implementation in SeqAn provides a find interface and requires the Finder and Pattern to be specialized with Swift<..>. Millions of sequences can be searched simultaneously with one Swift Pattern in a Swift Finder of a single haystack sequence. The error rate of a local alignment is the number of errors divided by the length of the needle sequence part of the match. There are currently two version of the Swift algorithm implemented in SeqAn, SwiftSemiGlobal and SwiftLocal. Both can be used to search epsilon-matches of a certain minimum length.

**Hint:** SwiftSemiGlobal should only be used for short needles (sequenced reads) as it always returns potential epsilon matches spanning a whole needle sequence. SwiftLocal should be preferred for large needles as it returns needle sequences potentially having an intersection with an epsilon match.

The following program searches for semi-global alignments between pairs of reads with a maximal error rate of 10%.

```cpp
#include <seqan/file.h>
#include <seqan/index.h>
#include <seqan/store.h>
#include <iostream>
using namespace seqan;

int main (int argc, char const * argv[]) {
    FragmentStore<> fragStore;
    if (argc < 2 || !loadReads(fragStore, argv[1]))
        return 1;

    Swift uses a q-gram index of the needle sequences. Thus, we have to specialize the Swift Semi Global Pattern with a IndexQGram index of the needle StringSet in the first template argument, create the index over the readSeqStore and pass the index to the Pattern constructor. Swift Semi Global Finder and Swift Semi Global Pattern classes have to be specialized with SwiftSemiGlobal in the second template argument.

**Note:** Note, to use the local swift filter you simply switch the specialization tag to SwiftLocal: Swift Local Finder and Swift Local Pattern.

The main loop iterates over all potential matches which can be further processed, e.g. by a semi-global or overlap aligner.
typedef FragmentStore<>::TReadSeqStore TReadSeqStore;
typedef GetValue<TReadSeqStore>::Type TReadSeq;
typedef Index<TReadSeqStore, IndexQGram<Shape<Dna, UngappedShape<11> >, OpenAddressing> > TIndex;
typedef Pattern<TIndex, Swift<SwiftSemiGlobal> > TPattern;
typedef Finder<TReadSeq, Swift<SwiftSemiGlobal> > TFinder;

TIndex index(fragStore.readSeqStore);
TPattern pattern(index);
for (unsigned i = 0; i < length(fragStore.readSeqStore); ++i)
{
    if ((i % 1000) == 0) std::cout << '.' << std::flush;
    TFinder finder(fragStore.readSeqStore[i]);
    while (find(finder, pattern, 0.1))
    {
        if (i == position(pattern).i1) continue;
        // do further alignment here
        /*
        std::cout << "Found possible overlap of " << std::endl;
        std::cout << "\t" << infix(finder) << std::endl;
        std::cout << "\t" << seqs[position(pattern).i1] << std::endl;
        */
    }

    return 0;
}
Preview with Universal Indent GUI

When started, the program will present you with a window like the following.

First, we set the indenter to Uncrustify.

Then, we load SeqAn’s uncrustify.cfg which is located in $CHECKOUT/misc. We can do so by selecting Indenter > Load Indenter Config File in the program menu.

Then, we load a file from the SeqAn repository, for example core/apps/sak/sak.cpp.

Now, we can toy around with the reformatter by checking Live Indent Preview.

The settings on the left panel allow us to tweak the style to our liking. Any changes can be stored by selecting Indenter > Load Indenter Config File in the program menu. The source can also be stored, using File > Save Source File and File > Save Source File As...

Using The Command Line

Uncrustify can also be used via the command line. This is best done after a rough visual verification that the uncrustify.cfg yields works for your sources using the Universal Indenter UI.

Work on single file:

2.2. How Tos
```cpp
#include <a.h>
#include <a.h>
#include <a.h>

using namespace a.b.c a;

PREPROCESSOR()

BEGIN MESSAGE MAP()
    ON_COMMAND()
    END_MESSAGE_MAP()

extern struct x y;

static const class Example : 
    Int1, Int2, Int3
{
    public:
        Example::~Example() :
            S1(), 
            S2(), 
            S3()
        { 
            // if statements with empty braces
            if(x) {}
            else if(x) {}
            else {}
            // if statements with exactly one braced statement
            if(x) {
```
```cpp
#include <a.h>
#include <a.h>
#include <a.h>

using namespace a.b.c a;

PREPROCESSOR()

BEGIN MESSAGE MAP()
  ON_COMMAND()
END_MESSAGE_MAP()

extern struct x y;

static const class Example:
  Int1, Int2, Int3
{
  public:
    Example::Example() :
      S1(),
      S2(),
      S3()
    {
      // if statements with empty braces
      if( x ) {}
      else if( x ) {}
    }
    else {}

    // if statements with exactly one braced statement
    if( x ) {
```
```c
int optionInfStart = 6;
int optionInfEnd = -1;
bool optionRevComp = false;
const char *optionOutput = NULL;
bool optionFastQ = false;
bool optionSeqNameSet = false;
CharString optionSeqName = "";
int optionMaxLength = -1;
unsigned optionLineLength = 200;

typedef DNA5 Alphabet;
typedef String<Alphabet> TSeqString;

// Load multi-Fasta sequences

// Template to load multi-Fasta sequences

template<
    typename TSeqSet,
    typename TQualls,
    typename TIDs
>
bool loadSeqs(TSeqSet &seqs, TQualls &qualls, TIDs &ids, const...

Multifasta multifasta;
if (open(multifasta.concat, fileName, OPEN_RDONLY))
    AutoSeqFormat format;
    guessFormat(multifasta.concat, format);
    split(multifasta, format);

int seqCount = length(multifasta);
```
```cpp
int optionInfStart = 8;
int optionInfEnd = -1;
bool optionRevComp = false;
const char * optionOutput = NULL;
bool optionFastQ = false;
bool optionSeqNameSet = false;
CharString optionSeqName = "";
int optionMaxLength = -1;
unsigned optionLineLength = 200;
typedef Dna5 TAlphabet;
typedef String<TAlphabet> TSeqString;

// Load multi-Fasta sequences

template <typename TSeqSet, typename TQals, typename TIDs
bool loadSeqs(TSeqSet & seqs, TQals & qals, TIDs & ids,
    MultiFasta multiFasta;
    if (open(multiFasta.concat, fileName, OPEN_RDONLY))
        return false;

    AutoSeqFormat format;
    guessFormat(multiFasta.concat, format);
    split(multiFasta, format);
```
# uncrustify -c \$\{(CHECKOUT)\}/misc/uncrustify.cfg --replace -f path/to/file.cpp

Batch work:

# find path/to -name '*.cpp' > list.txt
# uncrustify -c \$\{(CHECKOUT)\}/misc/uncrustify.cfg --replace -F list.txt

Automatically fix whitespaces in Xcode

Uncrustify can also be used directly from Xcode. With Xcode 4 Apple introduced so called “Behaviors” that can be executed using for instance keyboard shortcuts. To use uncrustify you can add a new behavior in the Xcode Preferences (tab Behaviors) and select “Run”. Here you add the attached ruby script.

![Behaviors](image)

Note: The script does not uncrustify the currently opened source file but all source files that were changed in your current svn checkout. Xcode does not provide the information which source file is currently opened.
2.2.10 Generating KNIME Nodes for External Tools

Prerequisites

KNIME SDK

You can download it from the KNIME Download Site. We will use Version 2.8. (We assume that you have installed it to $HOME/eclipse_knime_2.8.0).

git

For Downloading the latest samtools and GenericKnimeNodes.

Apache Ant

The Generic KNIME Plugins project uses Apache Ant as the build system. On Linux and Mac, you should be able to install it through your package manager. For Windows, see the Apache Ant Downloads (note that samtools does not work on Windows so you will not be able to follow through with this tutorial on Windows).

Running Example

We will adapt some functions from the samtools package to KNIME:

BamToSam

This tool will execute `samtools view -o ${OUT} ${IN}`.

SamToBam

This tool will execute `samtools view -Sb -o ${OUT} ${IN}`.

SortBam

This tool will execute `samtools sort -f -o ${OUT} ${IN}`.
**Hint:** The `-f` flag is required for the integration of samtools without a wrapper, since it would append `.bam` to `${OUT}` for getting the output name. However, only the current trunk version from the samtools GitHub project supports this flag.

**Preparation: Building samtools and Downloading GenericKnimeNodes**

As mentioned above, we have to build the current trunk version of samtools for the sort_bam tool to work. The following shell commands download the current samtools trunk from GitHub and build samtools. We will work in a new directory `knime_samtools` (we will assume that the directory is directly in your $HOME for the rest of the tutorial.

```
knime_samtools # git clone https://github.com/samtools/samtools
... knime_samtools # cd samtools
samtools # make
... samtools # ls -l samtools
-rwxr-xr-x 1 user group 1952339 May 7 16:36 samtools
samtools # cd ..
knime_samtools #
```

Then, we need to download GenericKnimeNodes:

```
knime_samtools # git clone git://github.com/genericworkflownodes/GenericKnimeNodes.git
```

**Preparation: Installing KNIME File Handling**

We need to install support for file handling nodes in KNIME. For this, open the window for installing Eclipse plugins; in the program’s main menu: Help > Install New Software....

Here, enter `http://www.knime.org/update/2.8/` into the Work with: field, enter file into the search box, and finally select KNIME File Handling Nodes in the list. Then, click Next and follow through with the installation of the plugin. When done, Eclipse must be restarted.

**Overview**

KNIME nodes are shipped as Eclipse plugins. The GenericKnimeNodes (GWN) package provides the infrastructure to automatically generate such nodes from the description of their command line. The description of the command line is kept in XML files called Common Tool Descriptor (CTD) files. The input of the GWN package is a directory tree with the following structure:

```
plugin_dir
    |__ plugin.properties
    |__ descriptors (place your ctd files and mime.types here)
    |__ payload (place your binaries here)
    |__ icons (the icons to be used must be here)
    |__ DESCRIPTION (A short description of the project)
    |__ LICENSE (Licensing information of the project)
    |__ COPYRIGHT (Copyright information of the project)
```
plugin.properties
File with the plugin configuration.

descriptors
Directory with the CTD files and a mime.types file. This mime.types file contains a mapping between MIME types and file extensions. There is one CTD file called ${app_name}.ctd.

payload
ZIP archives with the binaries are located here. This directory has to be present even if the directory is empty. Also, you need a file binaries.ini in this directory which can be empty or contain environment variable definitions as name=value lines.

icons
Some icons: A file category.png (15x15 px) for categories in the KNIME tool tree. A file splash.png (50x50 px) with an icon to display in the KNIME splash screen. One for each app, called ${app_name}.png.

DESCRIPTION
A text file with your project’s description.

LICENSE
A file with the license of the project.

COPYRIGHT
A file with copyright information for the project.

Obtaining the Demo Workflow Plugin Directory

Please download the file workflow_plugin_dir.zip and look around in the archive. Also have a look into binaries_*_*.zip files in payload. The structure of this ZIP file is explained below in Anatomy of a Plugin Directory.

Creating an Eclipse Plugin from the Plugin Directory

The next step is to use GKN to create an Eclipse plugin from the workflow plugin directory. For this, change to the directory GenericKnimeNodes that we cloned using git earlier. We then execute ant and pass the variables knime.sdk with the path to the KNIME SDK that you downloaded earlier and plugin.dir with the path of our plugin directory.

knime_samtools # cd GenericKnimeNodes
GenericKnimeNodes # ant -Dknime.sdk=${HOME}/eclipse_knime_2.8.0 \ 
-Dplugin.dir=${HOME}/knime_samtools/workflow_plugin_dir

This generates an Eclipse plugin with wrapper classes for our nodes. The generated files are within the generated_plugin directory of the directory GenericKnimeNodes.
Importing the Generated Projects into Eclipse

In the main menu File > Import.... In the Import window, select General > Existing Project Into Workspace

In the next dialog, click Browse... next to Select root directory.

Then, select the directory of your “GenericWorkflowNodes” checkout. The final dialog should then look as follows. Clicking finish will import (1) the GKN classes themselves and (2) your generated plugin’s classes.

Now, the packages of the GKN classes and your plugin show up in the left Package Explorer pane of Eclipse.

Hint: Information: Synchronizing ant build result with Eclipse.

Since the code generation happens outside of Eclipse, there are often problems caused by Eclipse not recognizing updates in generated .java files. After each call to ant, you should clean all built files in all projects by selecting the menu entries Project > Clean..., selecting Clean all projects, and then clicking OK.

Then, select all projects in the Package Explorer, right-click and select Refresh.
Import Projects

Select a directory to search for existing Eclipse projects.

- Select root directory:
- Select archive file:

Projects:

- Copy projects into workspace

Working sets
- Add project to working sets

Working sets:
Import Projects

Select a directory to search for existing Eclipse projects.

- Select root directory: /home/holtgrew/Development/knime_nodes/Gen
- Select archive file:

Projects:

- com.genericworkflownodes.knime (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkflownodes.knime.feature (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkflownodes.knime.node_generator (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkflownodes.knime.tests (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools.feature (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools.linux.x86_64 (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools.macosx.x86_64 (/home/holtgrew/Development/knime_nodes/Gen)

- Copy projects into workspace

Working sets

- Add project to working sets

Working sets: Select...
Launching Eclipse with our Nodes

Finally, we have to launch KNIME with our plugin. We have to create a run configuration for this. Select Run > Run Configurations....

In the Run Configurations window, select Eclipse Application on the left, then create the small New launch configuration icon on the top left (both marked in the following screenshot). Now, set the Name field to “KNIME”, select Run an application and select org.knime.product.KNIME_APPLICATION in the drop down menu. Finally, click Run.

Your tool will show up in the tool selector in community/SAM and BAM.

Here is an example KNIME workflow with the nodes that we just created.

Anatomy of a Plugin Directory

You can download a ZIP archive of the resulting project from the attached file workflow_plugin_dir.zip. We will ignore the contents of icons, DESCRIPTION, LICENSE, and COPYRIGHT here. You can see all relevant details by inspecting the ZIP archive.

The file plugin.properties

The content of the file plugin.properties is as follows:

```properties
# the package of the plugin
pluginPackage=net.sf.samtools

# the name of the plugin
pluginName=SamTools
```
When creating your own plugin directory, you only have to update the first three properties:

**pluginPackage**
A Java package path to use for the Eclipse package.

**pluginName**
A CamelCase name of the plugin.

**pluginVersion**
Version of the Eclipse plugin.

**The file descriptors/mime.types**

The contents of the file is as shown below. Each line contains the definition of a MIME type. The name of the mime type is followed (separated by a space) by the file extensions associated with the file type. There may be no ambiguous mappings, i.e. giving the extension for both `application/x-fasta` and `application/x-fastq`.

---

2.2. How Tos 475
application/x-fasta fa fasta
application/x-fastq fq fastq
application/x-sam sam
application/x-bam bam

The file descriptors/samtools_sort_bam.ctd

This file describes the SortBam tool for sorting BAM files. We do not describe the files descriptors/samtools_sam_to_bam.ctd and descriptors/samtools_bam_to_sam.ctd in the same detail as you can interpolate from here.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tool name="SortBam" version="0.1.17" category="SAM and BAM"
    docurl="http://samtools.sourceforge.net/samtools.shtml">
  <executableName>Samtools</executableName>
  <description><![CDATA[Samtools BAM Sorting.]]></description>
  <manual><![CDATA[samtools sort]]></manual>
  <docurl>Direct links in docs</docurl>
  <cli>
    <clielement optionIdentifier="sort" isList="false" />
    <clielement optionIdentifier="-f" isList="false" />

    <!-- Following clielements are arguments. You should consider providing a help text to ease understanding. -->
    <clielement optionIdentifier="" isList="false">
      <mapping referenceName="bam_to_sam.argument-0" />
    </clielement>
    <clielement optionIdentifier="" isList="false">
      <mapping referenceName="bam_to_sam.argument-1" />
    </clielement>
    <clielement optionIdentifier="" isList="false">
      <mapping referenceName="bam_to_sam.argument-2" />
    </clielement>
  </cli>
  <PARAMETERS version="1.4"
    xsi:noNamespaceSchemaLocation="http://open-ms.sourceforge.net/schemas/Param_1_4.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <NODE name="bam_to_sam" description="Samtools BAM to SAM conversion">
```
Here is a description of the tags and the attributes:

/tool
   The root tag.
/tool@name
   The CamelCase name of the tool as shown in KNIME and part of the class name.
/tool@version
   The version of the tool.
/tool@category
   The path to the tool’s category.
/tool/ executableName
   The name of the executable in the payload ZIP’s bin dir.
/tool/ description
   Description of the tool.
/tool/ manual
   Long description for the tool.
/tool/ docurl
   URL to the tool’s documentation.
/tool/ cli
   Container for the <clielement> tags. These tags describe the command line options and arguments of the tool. The command line options and arguments can be mapped to parameters which are configurable through the UI. The parameters are stored in tool/PARAMETERS
/tool/ cli/ clielement
   There is one entry for each command line argument and option.
/tool/ cli/ clielement@optionIdentifier
   The identifier of the option on the command line. For example, for the -l option of ls, this is -l.
/tool/ cli/ clielement@isList
   Whether or not the parameter is a list and multiple values are possible. One of true and false.
/tool/ cli/ clielement/mapping
   Provides the mapping between a CLI element and a PARAMETER.
/tool/ cli/ clielement/mapping@referenceName
   The path of the parameter. The parameters <ITEM> s in tool/PARAMETERS are stored in nested <NODE> tags and this gives the path to the specific parameter.
/tool/ PARAMETERS
   Container for the <NODE> and <ITEM> tags. The <PARAMETERS> tag is in a diferent namespace and provides its own XSI.
/tool/ PARAMETERS@version
   Format version of the <PARAMETERS> section.
A node in the parameter tree. You can use such nodes to organize the parameters in a hierarchical fashion.

(tool/PARAMETERS/.../NODE@advanced
  Boolean that marks an option as advanced.

(tool/PARAMETERS/.../NODE@name
  Name of the parameter section.

(tool/PARAMETERS/.../NODE@description
  Documentation of the parameter section.

(tool/PARAMETERS/.../ITEM
  Description of one command line option or argument.

(tool/PARAMETERS/.../ITEM@name
  Name of the option.

(tool/PARAMETERS/.../ITEM@value
  Default value of the option. When a default value is given, it is passed to the program, regardless of whether the user touched the default value or not.

(tool/PARAMETERS/.../ITEM@type
  Type of the parameter. Can be one of string, int, double, input-file, output-path, input-prefix, or output-prefix. Booleans are encoded as string with the restrictions attribute set to "true,false".

(tool/PARAMETERS/.../ITEM@required
  Boolean that states whether the parameter is required or not.

(tool/PARAMETERS/.../ITEM@description
  Documentation for the user.

(tool/PARAMETERS/.../ITEM@supported_formats
  A list of supported file formats. Example: ":*.bam,*.sam".

(tool/PARAMETERS/.../ITEM@restrictions
  In case of int or double types, the restrictions have the form min:, :max, min:max and give the smallest and/or largest number a value can have. In the case of string types, restrictions gives the list of allowed values, e.g. one,two,three. If the type is string and the restriction field equals "true,false", then the parameter is a boolean and set in case true is selected in the GUI. A good example for this would be the -l flag of the ls program.

**Hint:** If a <clielement> does provides an empty <tt>optionIdentifier</tt> then it is a positional argument without a flag (examples for parameters with flags are -n 1,--number 1).

If a <clielement> does not provide a <mapping> then it is passed regardless of whether has been configured or not.

The samtools_sort_bam tool from above does not provide any configurable options but only two arguments. These are by convention called argument-0 and argument-1 but could have any name.

Also, we always call the program with view -f as the first two command line arguments since we do not provide a mapping for these arguments.

The directory payload

The directory payload contains ZIP files with the executable tool binaries. There is one ZIP file for each platform (Linux, Windows, and Mac Os X) and each architecture (32 bit and 64 bit). The names of the files are binaries_${plat}_${arch}.zip where ${plat} is one of lnx, win, or mac, and ${arch} is one of 32 and 64.
Each ZIP file contains a directory /bin which is used as the search path for the binary given by <executableName>. Also, it provides an INI file /binaries.ini which can be used to define environment variables to set before executing any tools.

The ZIP file can also provide other files in directories such as /share.

**Generating KNIME Nodes for SeqAn Apps**

You can generate a workflow plugin directory for the SeqAn apps using the `prepare_workflow_plugin` target. Then, you can generate the Knime Nodes/Eclipse plugins as described above using ant.

```
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/release
~ # seqan-trunk-build/release
release # cmake ../../seqan-trunk
release # make prepare_workflow_plugin
release # cd ~/knime_samtools/GenericKnimeNodes
GenericKnimeNodes # ant -Dknime.sdk=${HOME}/eclipse_knime_2.8.0
          -Dplugin.dir=$HOME/seqan-trunk-build/release/workflow_plugin_dir
```

---

### 2.2.11 Generating SeqAn KNIME Nodes

**Learning Objective**  You will learn how to import applications written in SeqAn into the KNIME Eclipse plugin. After completing this tutorial, you will be able to use self made applications in KNIME workflows.

**Difficulty**  Very basic

**Duration**  1 h

**Prerequisites**

- **KNIME SDK**  You can download it from the [KNIME Download Site](at the end of the page). We will use Version 2.8. (We assume that you have installed it to $HOME/eclipse_knime_2.8. but it could be anywhere).
- **git**  For Downloading the latest GenericKnimeNodes.
Apache Ant  The Generic KNIME Plugins project uses Apache Ant as the build system. On Linux and Mac, you should be able to install it through your package manager. For Windows, see the Apache Ant Downloads.

We will generate a simple SeqAn KNIME node from a SeqAn app that reads a fastq file from disk and just writes it back. We start by installing the necessary software. Afterwards, we explain which steps are required in order to prepare a SeqAn app to be used in KNIME, and finally, we show how to import the app into KNIME. The following section provides some more information on the plugin structure and where the necessary information is stored. Note that this tutorial is mainly written for MacOS and Linux users, but Windows users should also be able to follow through.

Preparation: Downloading GenericKnimeNodes

We will work in a new directory knime_node (we will assume that the directory is directly in your $HOME for the rest of the tutorial).

knime_node # git clone git://github.com/genericworkflownodes/GenericKnimeNodes.git

Preparation: Installing KNIME File Handling

We need to install support for file handling nodes in KNIME. For this, open the window for installing Eclipse plugins; in the program’s main menu: Help > Install New Software....

Here, enter http://www.knime.org/update/2.8/ into the Work with: field, enter file into the search box, and finally select KNIME File Handling Nodes in the list. Then, click Next and follow through with the installation of the plugin. When done, Eclipse must be restarted.

Generating KNIME Nodes for SeqAn Apps

You can generate a workflow plugin directory for the SeqAn apps using the prepare_workflow_plugin target.

In order for your application to turn into a KNIME node, you have to add the line:

```
set (SEQAN_CTD_EXECUTABLES ${SEQAN_CTD_EXECUTABLES} <my_app> CACHE INTERNAL "")
```

to the end of the CMakeList.txt file of your application.

The following example will demonstrate the creation of a SeqAn app and its registration as a KNIME node.

```
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # cd seqan-trunk
~ # ./util/bin/skel.py app knime_node sandbox/my_sandbox
```

Now open the file seqan-trunk/sandbox/my_sandbox/apps/knime_node/knime_node.cpp and replace its content with the one found in seqan-trunk/core/demos/knime_node.cpp. The code implements the reading of a read file and its storage somewhere on the disk.

In order to register the app knime_node, you simply add the line

```
set (SEQAN_CTD_EXECUTABLES ${SEQAN_CTD_EXECUTABLES} knime_node CACHE INTERNAL "")
```

to seqan-trunk/sandbox/my_sandbox/apps/knime_node/CMakeList.txt.

Then, you can generate the Knime Nodes/Eclipse plugin. First, change to the directory GenericKnimeNodes that we cloned using git earlier. We then execute ant and pass the variables knime.sdk with the path to the KNIME SDK that you downloaded earlier and plugin.dir with the path of our plugin directory.
~ # mkdir -p seqan-trunk-build/release
~ # seqan-trunk-build/release
~ # cd seqan-trunk-build/release
release # cmake ../../seqan-trunk
release # make prepare_workflow_plugin
release # cd ~/knime_node/GenericKnimeNodes
GenericKnimeNodes # ant -Dknime.sdk=${HOME}/eclipse_knime_2.8.0 \
    -Dplugin.dir=${HOME}/seqan-trunk-build/release/workflow_plugin_dir

The generated files are within the generated_plugin directory of the directory GenericKnimeNodes.

If you ran into problems, you may copy the file my_sandbox.zip, which contains a fully functional sandbox with the knime_node app and the adjusted CMakeList.txt file. You still have to call ant though.

**Importing the Generated Projects into Eclipse**

In the main menu, go to File > Import.... In the Import window, select General > Existing Project Into Workspace.
In the next dialog, click Browse... next to Select root directory.

Then, select the directory of your “GenericWorkflowNodes” checkout. The final dialog should then look as follows. Clicking finish will import (1) the GKN classes themselves and (2) your generated plugin’s classes.

Now, the packages of the GKN classes and your plugin show up in the left Package Explorer pane of Eclipse.

**Tip:** Synchronizing ant build result with Eclipse.

Since the code generation happens outside of Eclipse, there are often problems caused by Eclipse not recognizing updates in generated ‘.java’ files. After each call to ant, you should clean all built files in all projects by selecting the menu entries Project > Clean..., selecting Clean all projects, and then clicking OK.
Import Projects

Select a directory to search for existing Eclipse projects.

Select root directory: /home/holtgrew/Development/knime_nodes/Gen

Projects:
- com.genericworkfloknodes.knime (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkfloknodes.knime.feature (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkfloknodes.knime.node_generator (/home/holtgrew/Development/knime_nodes/Gen)
- com.genericworkfloknodes.knime.tests (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools.feature (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools-linux.x86_64 (/home/holtgrew/Development/knime_nodes/Gen)
- net.sf.samtools-macosx-x86_64 (/home/holtgrew/Development/knime_nodes/Gen)

Copy projects into workspace

Add project to working sets

Finish
Then, select all projects in the Package Explorer, right-click and select Refresh.

Tip: You might get a warning with in one of the KNIME files. In order to remove it you need to download the KNIME’s test environment, but you can just ignore the error in our case.

Launching Eclipse with our Nodes

Finally, we have to launch KNIME with our plugin. We have to create a run configuration for this. Select Run -> Run Configurations....

In the Run Configurations window, select Eclipse Application on the left, then click the small New launch configuration icon on the top left (both marked in the following screenshot). Now, set the Name field to “KNIME”, select Run an application and select org.knime.product.KNIME_APPLICATION in the drop down menu. Finally, click Run.

Your tool will show up in the tool selector in Community Nodes.

Important: Sometimes KNIME complains about the Java version you are using. In that case, you can use Java 1.6. as shown in Choosing The JRE Version.

Important: If you are running a MacOS you might need to add -Xms40m -Xmx512M -XX:MaxPermSize=256m -Xdock:icon=./Resources/Eclipse.icns -XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts -server to the VM argument box of your Run Configuration.

You should now be able to use the created node in a KNIME workflow. The following sections provide additional information about the structure of the plugin and where the crucial information is stored.
Plugin Overview

KNIME nodes are shipped as Eclipse plugins. The **GenericKnimeNodes** (GWN) package provides the infrastructure to automatically generate such nodes from the description of their command line. The description of the command line is kept in XML files called Common Tool Descriptor (CTD) files. The input of the GWN package is a directory tree with the following structure.

```
plugin_dir
 |-- plugin.properties
 |-- descriptors (place your ctd files and mime.types here)
  |  |-- payload (place your binaries here)
  |  |-- icons (the icons to be used must be here)
  |  |-- DESCRIPTION (A short description of the project)
  |  |-- LICENSE (Licensing information of the project)
  |  |-- COPYRIGHT (Copyright information of the project)
```

**plugin.properties** File with the plugin configuration.

**descriptors** Directory with the CTD files and a mime.types file. This mime.types file contains a mapping between MIME types and file extensions. There is one CTD file called `app_name`.ctd.
payload  ZIP archives with the binaries are located here. This directory has to be present even if the directory is empty. Also, you need a file `binaries.ini` in this directory which can be empty or contain environment variable definitions as `name=value` lines.

icons  Some icons: A file `category.png` (15x15 px) for categories in the KNIME tool tree. A file ‘splash.png’ (50x50 px) with an icon to display in the KNIME splash screen. One for each app, called `${app_name}.png`.

DESCRIPTION  A text file with your project’s description.

LICENSE  A file with the license of the project.

COPYRIGHT  A file with copyright information for the project.

The GWN project provides tools to convert such a plugin directory into an Eclipse plugin. This plugin can then be launched together with KNIME. The following picture illustrates the process.

![Anatomy of a Plugin Directory](image)

Anatomy of a Plugin Directory

You can download a ZIP archive of the resulting project from the attached file `workflow_plugin_dir.zip`. We will ignore the contents of `icons`, `DESCRIPTION`, `LICENSE`, and `COPYRIGHT` here. You can see all relevant details by inspecting the ZIP archive.

The file `plugin.properties`

The content of the file `plugin.properties` is as follows:

```plaintext
# the package of the plugin
pluginPackage=de.seqan

# the name of the plugin
pluginName=SeqAn

# the version of the plugin
pluginVersion=1.5.0.201309051220

# the path (starting from KNIMEs Community Nodes node)
nodeRepositoyRoot=community

executor=com.genericworkflownodes.knime.execution.impl.LocalToolExecutor
commandGenerator=com.genericworkflownodes.knime.execution.impl.CLICommandGenerator
```

When creating your own plugin directory, you only have to update the first three properties:

`pluginPackage`  A Java package path to use for the Eclipse package.

`pluginName`  A CamelCase name of the plugin.

`pluginVersion`  Version of the Eclipse plugin.
The file descriptors/mime.types

The contents of the file is as shown below. Each line contains the definition of a MIME type. The name of the mime type is followed (separated by a space) by the file extensions associated with the file type. There may be no ambiguous mappings, i.e. giving the extension for both application/x-fasta and application/x-fastq.

application/x-fasta fa fasta
application/x-fastq fq fastq
application/x-sam sam
application/x-bam bam

The file descriptors/samtools_sort_bam.ctd

This file descripes the SortBam tool for sorting BAM files. We do not describe the files descriptors/samtools_sam_to_bam.ctd and descriptors/samtools_bam_to_sam.ctd in the same detail as you can interpolate from here.

<?xml version="1.0" encoding="UTF-8"?>
<tool name="KnimeNode" version="0.1" docurl="http://www.seqan.de" category=""

<executableName>knime_node</executableName>
<description>This is a very simple KNIME node providing an input and output port.</description>
<manual>This is a very simple KNIME node providing an input and output port. The code should be modified such that the node does something useful</manual>
</tool>

<PARAMETERS version="1.6.2" xsi:noNamespaceSchemaLocation="http://open-ms.sourceforge.net/schemas/Param_1_6_2.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<NODE name="knime_node" description="This is a very simple KNIME node providing an input and output port.">

<ITEM name="write-ctd-file-ext" value="" type="string" description="Override file extension for --write-ctd" />
<ITEM name="arg-1-file-ext" value="" type="string" description="Override file extension for argument 1" restrictions="fastq,fq" />
<ITEM name="outputFile" value="result.fastq" type="output-file" description="Name of the multi-FASTA output." supported_formats="*.fastq,*.fq" />
<ITEM name="outputFile-file-ext" value="" type="string" description="Override file extension for --outputFile" restrictions="fastq,fq" />
<ITEM name="quiet" value="false" type="string" description="Set verbosity to a minimum." restrictions="true,false" />
<ITEM name="verbose" value="" type="string" description="Set verbosity to medium." />
<ITEM name="very-verbose" value="" type="string" description="Set verbosity to high." />
</NODE>
</PARAMETERS>
Here is a description of the tags and the attributes:

/tool The root tag.

/tool@name The CamelCase name of the tool as shown in KNIME and part of the class name.

/tool@version The version of the tool.

/tool@category The path to the tool's category.

/tool/executableName The name of the executable in the payload ZIP's bin dir.

/tool/description Description of the tool.

/tool/manual Long description for the tool.

/tool/docurl URL to the tool's documentation.

/tool/cli Container for the <clielement> tags. These tags describe the command line options and arguments of the tool. The command line options and arguments can be mapped to parameters which are configurable through the UI. The parameters are stored in /tool/PARAMETERS

/tool/cli/clielement There is one entry for each command line argument and option.

/tool/cli/clielement?optionIdentifier The identifier of the option on the command line. For example, for the -l option of ls, this is -l.

/tool/cli/clielement@isList Whether or not the parameter is a list and multiple values are possible. One of true and false.

/tool/cli/clielement/mapping Provides the mapping between a CLI element and a PARAMETER.

/tool/cli/clielement/mapping@referenceName The path of the parameter. The parameters <ITEM>s in /tool/PARAMETERS are stored in nested <NODE> tags and this gives the path to the specific parameter.

/tool/PARAMETERS Container for the <NODE> and <ITEM> tags. The <PARAMETERS> tag is in a different namespace and provides its own XSI.

/tool/PARAMETERS@version Format version of the <PARAMETERS> section.

/tool/PARAMETERS/.../NODE A node in the parameter tree. You can use such nodes to organize the parameters in a hierarchical fashion.

/tool/PARAMETERS/.../NODE@advanced Boolean that marks an option as advanced.

/tool/PARAMETERS/.../NODE@name Name of the parameter section.

/tool/PARAMETERS/.../NODE@description Documentation of the parameter section.

/tool/PARAMETERS/.../ITEM Description of one command line option or argument.

/tool/PARAMETERS/.../ITEM@name Name of the option.

/tool/PARAMETERS/.../ITEM@value Default value of the option. When a default value is given, it is passed to the program, regardless of whether the user touched the default value or not.

/tool/PARAMETERS/.../ITEM@type Type of the parameter. Can be one of string, int, double, input-file, output-path, input-prefix, or output-prefix. Booleans are encoded as string with the restrictions attribute set to "true,false".
/tool/PARAMETERS/.../ITEM@required Boolean that states whether the parameter is required or not.

/tool/PARAMETERS/.../ITEM@description Documentation for the user.

/tool/PARAMETERS/.../ITEM@supported_formats A list of supported file formats. Example:
"*.bam,*.sam".

/tool/PARAMETERS/.../ITEM@restrictions In case of int or double types, the restrictions have the form min.:max, min:max and give the smallest and/or largest number a value can have. In the case of string types, restrictions gives the list of allowed values, e.g. one,two,three. If the type is string and the restriction field equals "true,false", then the parameter is a boolean and set in case true is selected in the GUI. A good example for this would be the -i flag of the ls program.

**Tip:** If a `<clielement>` does provide an empty optionIdentifer then it is a positional argument without a flag (examples for parameters with flags are -n 1, --number 1).

If a `<clielement>` does not provide a `<mapping>` then it is passed regardless of whether has been configured or not.

The `samtools_sort_bam` tool from above does not provide any configurable options but only two arguments. These are by convention called argument-0 and argument-1 but could have any name.

Also, we always call the program with `view -f` as the first two command line arguments since we do not provide a mapping for these arguments.

**The directory payload**

The directory payload contains ZIP files with the executable tool binaries. There is one ZIP file for each platform (Linux, Windows, and Mac Os X) and each architecture (32 bit and 64 bit). The names of the files are `binaries_${plat}_${arch}.zip` where `{plat}` is one of lnx, win, or mac, and `{arch}` is one of 32 and 64.

Each ZIP file contains a directory `/bin` which is used as the search path for the binary given by `<executableName>`. Also, it provides an INI file `/binaries.ini` which can be used to define environment variables to set before executing any tools.

The ZIP file can also provide other files in directories such as `/share`.

### ToC

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Installing Contribs On Windows</td>
</tr>
<tr>
<td>– Download Contribs</td>
</tr>
<tr>
<td>– Extract Contribs</td>
</tr>
<tr>
<td>– Re-run CMake</td>
</tr>
</tbody>
</table>

#### 2.2.12 Installing Contribs On Windows

**Download Contribs**

The downloadable contribs contain precompiled library binaries (zlib, libbz2) for Windows by the supported compilers. The contribs come in 32 bit and 64 bit variants.
• Download contribs for 32 bit builds.
• Download contribs for 64 bit builds.

You can install both variants in parallel if you want to do both 32 bit and 64 bit builds.

**Extract Contribs**

Now, extract the downloaded ZIP file either to C:\Program Files or C:.

After downloading the 64 bit variant, you should now have a folder named C:\Program Files\seqan-contrib-D20130710-x64 or a folder named C:\seqan-contrib-D20130710-x64.

After downloading the 32 bit variant, you should now have a folder named C:\Program Files\seqan-contrib-D20130710-x86 or a folder named C:\seqan-contrib-D20130710-x86.

**Re-run CMake**

You now have to re-run CMake to find the libraries. You also have to remove the CMake Cache so it finds the new libraries. You might also need to update your SeqAn Checkout.

The following assumes that your checkout is in c:\seqan-trunk and your build directory is c:\seqan-build\vs10.

```bash
> cd c:\seqan-trunk
> svn update .
> cd c:\seqan-build\vs10
> del CMakeCache.txt
> cmake c:\seqan-trunk -G "Visual Studio 2010"
```

## 2.2.13 Profiling Programs

**Linux Perf Tools (Linux)**

• https://perf.wiki.kernel.org/

• Requires echo ’-1’ > /proc/sys/kernel/perf_event_paranoid as root.

Useful commands:

• perf top - display top-like display but on function granularity
• perf record PROGRAM - execute PROGRAM with profiling
• perf report PROGRAM - display report for PROGRAM
Google Perftools (Linux, Mac Os X)

- Download and install http://code.google.com/p/gperftools/ (also available through Ubuntu/Debian packages)
- Compile your program with debug symbols (you probably want to enable optimization as well).

```sh
# Tell the profiler where to write its output.
export CPUPROFILE=\$\{OUT\}
LD_PRELOAD="/usr/lib/libprofiler.so.0" \$\{PROGRAM\} \$\{COMMAND_LINE\}
google-pprof \$\{PROGRAM\} \$\{OUT\}
```

Interesting commands:

- `gv/web` - display weighted call graph in gv or in your browser
- `top/topX` - display top 10/X hitters
- `disasm NAME` - disassemble functions matching NAME

Instruments (Mac Os X)

2.2.14 Setup Nightly Builds

Subversion Repository

There is a Subversion repository for the nightly build scripts at http://svn.mi.fu-berlin.de/seqan-nightly/trunk/.

Caveats

32bit: bzlib and zlib

On Debian, you need the packages `libz-dev` and `libbz2-dev`. If you want to do 32 bit builds, you have to install `lib32bz2-dev` and `lib32z1-dev`.

2.2.15 Using Parallel Build Directories

Motivation

Why would you need more than one build directory or more than one IDE project file? This is very useful

- if you want use the same set of source files from multiple version of the same IDE (e.g. two Visual Studio versions),
• if you want to have both debug builds (for debugging) and release builds (for performance tests) in parallel,
• if you have your source files stored on a shared network location and want to have build files on two computer
and/or operating systems, or
• if you want to build the sources with two different compilers or compiler versions at the same time (e.g. to see
whether you can figure out compiler errors better from the messages by another compiler).

This How-To also serves as a collection of CMake command lines for copy-and-paste.

The Overall Idea

The overall idea is very simple: you create one build directory for each variant and call CMake in each of it using
different settings.

If you want to have different IDE project files then you use different CMake generators. In most IDEs, there is an
option to select debug or release builds. For the CMake Makefile generator, however, we can select the build types
using a command line option. Also, the compiler program (and version) can be switched using a command line option.

Generating Parallel IDE Project Files (Visual Studio, Xcode etc.)

You will only be able to generate files for Xcode when on Mac Os X, for Visual Studio when on Windows and so on.
The following section assumes that you have a subdirectory of your SeqAn checkout called build. We will create
subdirectories for each IDE project we create.

Creating Directories

For example, if we have installed Visual studio 8, 9, and 10 and want to create project files for each, we might use the
following commands to create the directories:

```
> cd seqan-trunk\build
> mkdir vs8
> mkdir vs9
> mkdir vs10
```

For XCode on Mac, we could do the following:

```
$ cd seqan-trunk/build
$ mkdir xcode
```

Note that you can choose any directory name. You have to take care that no such directory exists before. Previously
generated project files can break the generation process!

Generating Project Files

We can now use CMake to generate the project files specifying a generator with the command line parameter -G.

Let us generate the Visual Studio projects in the directories we mentioned above:

```
> cd vs8
> cmake -G "Visual Studio 8 2005" ..\..\..
> cd ..\vs9
> cmake -G "Visual Studio 9 2008" ..\..\..
> cd ..\vs10
> cmake -G "Visual Studio 10" ..\..\..
```

Click more... to see the commands for 64 bit builds.
On Mac Os X, we can generate XCode build files as follows:

```
# cd xcode
# cmake -G "Xcode" ../../..
```

### How To Make Release/Debug builds

When using the Makefile generator, it is useful to have multiple build types. CMake supports the following build types:

- **Debug**  No optimization, with debug symbols.
- **Release** Optimization, without debug symbols.
- **RelWithDebInfo** Optimization, with debug symbols. Useful for profiling.
- **MinSizeRel** Size-optimized release binary without debug symbols.

You can select the build type with a command line parameter to `cmake`, e.g. `-DCMAKE_BUILD_TYPE=Debug` or `-DCMAKE_BUILD_TYPE=Release`.

### Picking A Compiler

You can pick a C++ compiler using the command line parameter to `cmake`, e.g. `-DCMAKE_CXX_COMPILER=g++-4.1` or `-DCMAKE_CXX_COMPILER=clang++`.

### Creating Directories

Let’s create a build directory with the system’s default compiler both in debug and release mode. Also, we create one directory for the Clang compiler in debug mode.

```
# cd seqan-trunk/build
# mkdir Debug
# mkdir Release
# mkdir Debug-clang
```

Note that you should use fresh directories. Previously generated Makefiles can break the generation process!

### Generating Project Files

```
# cd Debug
# cmake ..../..
# cd ../Release
# cmake -DCMAKE_BUILD_TYPE=Release ..../..
# cd ../Debug-clang
# cmake -DCMAKE_CXX_COMPILER=clang++
```

Note that when using clang, you actually have to use clang++ and not clang (although clang++ usually only is a symlink to clang). If you use clang then all C++ features will be disabled and you will get configuration errors.
2.2.16 Creating Workflows with KNIME

KNIME is a well established data analysis framework which supports the generation of workflows for data analysis. In the following, we describe how to use SeqAn applications in KNIME.

Install SeqAn in KNIME

The Installation of the SeqAn NGS Toolbox in KNIME is very easy. Download the latest KNIME release from the KNIME website. In KNIME click on Help > Install new Software.

In the opening dialog choose Add....

In the opening dialog fill in the following Information:

Name  KNIME Nightly Unstable
Location  http://tech.knime.org/update/community-contributions/nightly/
After pressing OK, KNIME will show you all the contents of the added Update Site, containing also the SeqAn nodes.

Select the SeqAn NGS Toolbox and click Next. Follow the instructions. After a restart of KNIME the SeqAn nodes will be available under Community Nodes.

Add your own application to KNIME

Using the CTD and a node generator program, all SeqAn applications that use the ArgumentParser can be made available to run in KNIME. This is done automatically and nightly for all applications in the master branch on github that are listed in the CMAKE variable SEQAN_CTD_EXECUTABLES. The auto-generated KNIME nodes of these apps are then uploaded to the KNIME community node server and can easily be used by all KNIME users.

The following two steps are required to make your application KNIME-ready.
Adapt your applications to use the argument parser

Follow the Parsing Command Line Arguments tutorial and adapt your application to use only the ArgumentParser to parse command line arguments. Especially, take care to:

1. Declare your input and output file names as such via ArgParseArgument::INPUTFILE and ArgParseArgument::OUTPUTFILE.

2. Detect the file format from the file extension (and not from a dedicated file format option). This can be done, for example, with guessFormatFromFilename() on an AutoSeqFormat object to detect a particular sequence format (e.g. FASTA) in a predefined set of formats.

3. For input/output files define a list of possible extensions via setValidValues() (e.g. “fa fasta”). This list of possible extensions can be generated with getFileFormatExtensions() for a TagSelector of predefined file formats (e.g. AutoSeqFormat).

4. Avoid mutual exclusive options or other constraints that cannot be not represented by the ArgumentParser, simply ignore one of them (depending on a behavioral option). See the ArgumentParser tutorial if you need to define a numerical interval of possible values or a finite set of argument options.

5. Give default values.

Register your application to be considered by the node generator

Add the following section to the CMakeLists.txt file in your application folder (replace razers by your executable name):

```cmake
# Setup Common Tool Description for Generic Workflow Nodes
# Include executable razers in CTD structure.
set (SEQAN_CTD_EXECUTABLES $<SEQAN_CTD_EXECUTABLES> razers CACHE INTERNAL "")
```

Use existing and contribute new workflows

With the steps described above you will be able to set up your own workflows in KNIME. If you want to contribute a workflow to the SeqAn community or use workflows from others you can do that on https://github.com/seqan/knime_seqan_workflows

To contribute your own workflow, simply clone the workflow git repository into your own github repository and add a new folder WORKFLOWNAME_workflow. In KNIME export your workflow without the data files as a .zip file into that folder. Provide a README, a screenshot and some examples as well. Just have a look into existing workflow folders to get a notion.

After everything is ready, add and commit the new folder into your github repository and make a github pull request to the original workflow repository (https://github.com/seqan/knime_seqan_workflows) and - voila - it will be shared with the community.
2.2.17 Using The Code Generator

SeqAn comes with a code generator to create new sandboxes, library modules, apps, tests, and demos. You must have Python (>= 2.5) installed to use this generator.

The generator can be used to create:

- library modules,
- tests for library modules,
- application (app) directories,
- demo programs,
- headers for applications, and
- library headers.

For more information, see the articles SeqAn Repository Structure and The CMake-Based Build System.

The top level help screen of the skel.py program looks as follows:

Usage: skel.py [options] repository NAME
        skel.py [options] [module|test|app|demo|header|lheader] NAME LOCATION

The SeqAn code generator. The first version ("repository") is to be be called to create your new entries below the directory sandbox. The second version is to be called to create new library modules, tests, apps, and demos inside a sandbox.

Options:
  -h, --help show this help message and exit
  -s SKEL_ROOT, --skel-root=SKEL_ROOT
    Set path to the directory where the skeletons live in. Taken from environment variable SEQAN_SKELS if available.
  -a AUTHOR, --author=AUTHOR
    Set author to use. Should have the format USER <EMAIL>. Taken from environment variable SEQAN_AUTHOR if it exists.
Creating A Sandbox

Creating a sandbox is easy. The following command will create a new sandbox called `my_sandbox` within the directory `sandbox` of the repository.

```
seqan # ./util/bin/skel.py repository sandbox/my_sandbox
...
```

```
seqan # $ tree sandbox/my_sandbox/
sandbox/my_sandbox/
    -- CMakeLists.txt
    -- apps
        |  -- CMakeLists.txt
    -- demos
        |  -- CMakeLists.txt
    -- include
        |  -- seqan
    -- tests
        -- CMakeLists.txt
```

Creating An App

Inside the sandboxes, you can create new apps using the `skel.py` tool, too. We create a new app in our sandbox called “my_app”.

```
seqan # ./util/bin/skel.py app my_app sandbox/my_sandbox
...
```

```
seqan # tree sandbox/my_sandbox/apps
sandbox/my_sandbox/apps
    -- CMakeLists.txt
    -- my_app
        -- CMakeLists.txt
        -- INFO
        -- my_app.cpp
```

Creating A Library Module

Library modules can be created likewise.

```
seqan # ./util/bin/skel.py module my_module sandbox/my_sandbox
...
```

```
seqan # tree sandbox/my_sandbox/include/seqan
sandbox/my_sandbox/include/seqan
    -- my_module
        |  -- INFO
        |  -- my_module_base.h
    -- my_module.h
```
Creating A Test

Now, we can also create a test for our module.

```bash
seqan # ./util/bin/skel.py test my_module sandbox/my_sandbox
...  
sandbox/my_sandbox/tests/my_module/
   -- CMakeLists.txt
   -- test_my_module.cpp
   -- test_my_module.h
```

Create A Demo

Demos can be created in a similar way:

```bash
seqan # ./util/bin/skel.py demo my_demo sandbox/my_sandbox
...  
seqan # tree ./sandbox/my_sandbox/demos
./sandbox/my_sandbox/demos
   -- CMakeLists.txt
   -- my_demo.cpp
```

Create A Header

To create a header in your application, use the following command:

```bash
seqan # ./util/bin/skel.py header header_name.h sandbox/my_sandbox/apps/my_app
```

Create A Library Header

To create a library header (one with a namespace `seqan { ... }` construct), use the following:

```bash
seqan # ./util/bin/skel.py lheader lheader_name.h sandbox/my_sandbox/include/seqan/my_module
```

Setting The Author String

By default, SeqAn uses `Your Name <your.email@example.net>` as the author string in the generated files. There are two ways to change this:

First, you can set the environment variable `SEQAN_AUTHOR`:

```bash
seqan # export SEQAN_AUTHOR='Me <me@example.com>'
seqan # ./util/bin/skel.py demo my_demo sandbox/my_sandbox
```

Second, you can use the `--author` parameter when calling `skel.py`. This will override the environment variable `SEQAN_AUTHOR` if set.

```bash
seqan # ./util/bin/skel.py --author 'Me <me@example.com>' demo my_demo sandbox/my_sandbox
```

Creating App Tests

See the article *Writing App Tests* for this.
Using Your Own Templates

You might want to use your own templates, e.g. to replace the license comment header at each top of the file. To do this, you have to create a copy of /util/skel, e.g. to ~/.seqan/templates. Then, you can edit the files, but you have to keep the file names intact. You can specify the location of your template by specifying --skel-root or having set the environment variable SEQAN_SKELS when calling skel.py.

In your templates, the following placeholders will be replaced. Note the trailing s, leaving this out is a common source of error.

%(AUTHOR)s will be replaced by the author’s name, either given on command line or taken from environment variable SEQAN_AUTHOR.

%(NAME)s will be replaced by the name of the generated code.
%(TITLE)s will be replaced by the name of the generated, but centered in 74 characters, to be used in the file header comment.

%(YEAR)d will be replaced by the current year.
%(DATE)s will be replaced by the current date.
%(TIME)s will be replaced by the current time.

%(HEADER_GUARD)s will be replaced by the UPPER_CASE_PATH_H_ to the file.

%(CMAKE_PROJECT_PATH)s will be replaced by lower_case_path to the target directory.

2.2.18 Working With Custom Score Matrices

This How To describes how to create new scoring matrices for Amino Acids and DNA alphabets and how to load score matrices from files.

Creating A New Built-In Score Matrix

The following program demonstrate how to implement a new built-in score matrix.

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/file.h> // For printing strings.
#include <seqan/score.h> // The module score.

using namespace seqan;
```

Then, we perform the necessary definitions for the matrix. This consists of three steps:
• defining a tag struct
• specializing the class ScoringMatrixData_ with your tag

Note how we use enum values to compute the matrix size which itself is retrieved from the ValueSize metafunction.

// Extend SeqAn by a user-define scoring matrix.
namespace seqan {

// We have to create a new specialization of the ScoringMatrix_ class
// for amino acids. For this, we first create a new tag.
struct UserDefinedMatrix {};

// We also do this for the DNA alphabet.
struct AnotherUserDefinedMatrix {};

// Then, we specialize the class ScoringMatrix_.

template<>
struct ScoringMatrixData_<int, AminoAcid, UserDefinedMatrix> {

enum {

VALUE_SIZE = ValueSize<AminoAcid>::VALUE,
TAB_SIZE = VALUE_SIZE * VALUE_SIZE

};

static inline int const * getData() {

// The user defined data table. In this case, we use the data from BLOSUM-30.

static int const _data[TAB_SIZE] = {

4, -1, 0, 0, -3, 1, 0, 0, -2, 0, -1, 0, 1, -2, -1, 1, 1, -5, -4, 1, 0, 0,
-1, 8, -2, -1, -2, 3, -1, -2, -1, -3, -2, 1, 0, -1, -1, -1, -3, 0, 0, -1, -2, 0, -1,
0, -2, 8, 1, -1, -1, 1, 0, -1, 0, -2, 0, 0, -1, -3, 0, 1, -7, -4, -2, 4, -1,
0, -1, 1, 9, -3, -1, 1, -1, -2, -1, 0, -3, -5, -1, 0, -1, -4, -1, -2, 5, 0, -1,
-3, -2, -1, -5, 17, -2, 1, -4, -5, -2, 0, -3, -2, -3, -2, -2, -2, -6, -2, -2, 0,
1, 3, -1, -1, -2, 8, 2, -2, 0, -2, -2, 0, -1, -3, 0, 0, -1, -1, -1, -3, -1, 4,
0, -1, -1, 1, 2, 6, -2, 0, -3, -1, 2, -1, -4, 1, 0, -2, -1, -2, -3, 0, 5, -3,
0, -2, 0, -1, -4, -2, -2, 8, -3, -1, -2, -1, -2, -3, -1, 0, -2, 1, -3, -3, 0, -2,
-2, -1, -1, -2, -5, 0, 0, -3, 14, -2, -1, -2, -2, -3, -1, 0, -2, -5, 0, -3, -2, 0, -2,
0, -3, 0, -4, -2, -2, -2, -1, 6, 2, -2, 1, 0, -3, -1, 0, -3, -1, 4, -2, -3,
-2, -2, -1, 0, -2, -1, -2, -1, 2, 4, -2, 2, 2, -2, -2, 0, -2, 3, 1, -1, -1,
0, 1, 0, 0, -3, 0, 2, -1, -2, -2, -2, 4, -2, -1, 1, 0, -1, -2, -1, 0, 1,
1, 0, 0, -3, -2, -1, -1, -2, 2, 1, 2, 2, 6, -2, -4, -2, 0, -3, -1, 0, -2, -1,
-2, -1, -1, -5, -3, -3, -4, -3, -3, 0, 2, -1, -2, 10, -4, -1, -2, 1, 3, 1, -3, -4,
-1, -1, -3, -1, 3, 0, 1, -1, 1, -3, -3, 1, -4, -4, 11, -1, 0, -3, -2, -4, -2, 0,
0, -1, 1, 0, 0, -2, -1, 0, 0, -1, -2, -1, 2, 0, -3, -2, -1, 0, -1,
1, -3, 1, -1, -2, 2, 0, -2, -2, -2, 0, 0, -1, 0, -2, 0, 2, 5, -5, -1, 1, 0, 1,
-5, 0, -7, -4, -2, -1, -1, 1, -5, -3, -2, -2, -3, -1, -3, -5, 20, 5, -3, -5, -1,
-4, 0, -4, -1, -6, -1, -2, -3, 0, -1, 3, -1, -1, 3, -2, -2, -1, 5, 9, 1, -3, -2,
1, -1, -2, -2, -2, -3, -3, -3, 4, 1, -2, 0, 1, -4, -1, -1, -3, 1, 5, -2, 3,
0, -2, 4, 5, -2, -1, 0, 0, -2, -2, -1, 0, -2, -3, -2, 0, 0, -5, -3, -2, 5, 0, -1,
0, 0, -1, 0, 0, 4, 5, -2, 0, -3, -1, 1, -1, -4, 0, -1, -1, -2, -3, 0, 4,
0, -1, 0, -1, -2, 0, -1, -1, 0, 0, 0, 0, -1, -1, 0, 0, -2, -1, 0, -1, 0, -1,
-7, -7, -7, -7, -7, -1, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7, -7,

};

return _data;

};

// And we do this for the Dna5 alphabet.

template<>
struct ScoringMatrixData_<int, Dna5, AnotherUserDefinedMatrix> {

enum {

VALUE_SIZE = ValueSize<Dna5>::VALUE,

};

static inline int const * getData() {

};

};
We define a function `showScoringMatrix` for displaying a matrix.

```cpp
// Print a scoring scheme matrix to stdout.
template <typename TScoreValue, typename TSequenceValue, typename TSpec>
void showScoringMatrix(Score<TScoreValue, ScoreMatrix<TSequenceValue, TSpec>> const & scoringScheme) {
    // Print top row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i) {
        std::cout << TSequenceValue(i) << std::endl;
    }
    // Print each row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i) {
        for (unsigned j = 0; j < ValueSize<TSequenceValue>::VALUE; ++j) {
            std::cout << score(scoringScheme, TSequenceValue(i), TSequenceValue(j)) << std::endl;
        }
    }
}
```

Finally, the function `main` function demostrates some of the things you can do with scores:

- Construct empty score matrix object (2.)
- Programatically fill the matrix with a built-in matrix values (3.1)
- Programatically fill the score matrix in a loop (3.2)
- Programatically fill the matrix with the user-defined matrix values (3.3)
- Directly create a score matrix with the user-defined matrix values (4)

```cpp
int main() {
    // 1. Define type and constants.
    // Define types for the score value and the scoring scheme.
    typedef int TValue;
    typedef Score<TValue, ScoreMatrix<AminoAcid, Default>> TScoringScheme;
    // Define our gap scores in some constants.
    int const gapOpenScore = -1;
    int const gapExtendScore = -1;
    // 2. Construct scoring scheme with default/empty matrix.
}
// Construct new scoring scheme, alternatively only give one score
// that is used for both opening and extension.
TScoringScheme scoringScheme(gapExtendScore, gapOpenScore);

// 3. Fill the now-existing ScoreMatrix
//
// The scoring scheme now already has a matrix of the size
// ValueSize<AminoAcid>::VALUE x ValueSize<AminoAcid>::VALUE which
// we can now fill.

// 3.1 First, fill it with BLOSUM30.
std::cout << "BLOSUM 30" << std::endl;
setDefaultScoreMatrix(scoringScheme, Blosum30_());
showScoringMatrix(scoringScheme);

// 3.2 Now, we fill it with the product of the coordinates.
std::cout << "Coordinate Products" << std::endl;
for (unsigned i = 0; i < ValueSize<AminoAcid>::VALUE; ++i) {
    for (unsigned j = 0; j < ValueSize<AminoAcid>::VALUE; ++j) {
        setScore(scoringScheme, AminoAcid(i), AminoAcid(j), i * j);
    }
}
showScoringMatrix(scoringScheme);

// 3.3 Now, we fill it with the user defined matrix above.
std::cout << "User defined matrix (also BLOSUM 30)...." << std::endl;
setDefaultScoreMatrix(scoringScheme, UserDefinedMatrix());
showScoringMatrix(scoringScheme);

// 4. Create ScoreMatrix object with user-defined matrix.
std::cout << "User scoring scheme...." << std::endl;
Score<TValue, ScoreMatrix<AminoAcid, UserDefinedMatrix> > userScoringScheme;
showScoringMatrix(userScoringScheme);

// 5. Show our Dna5 scoring matrix.
std::cout << "User DNA scoring scheme...." << std::endl;
Score<TValue, ScoreMatrix<Dna5, AnotherUserDefinedMatrix> > userScoringSchemeDna;
showScoringMatrix(userScoringSchemeDna);

return 0;

Here is the output of the program:

$ make tutorial_init_score
$ ./demos/tutorial_init_score
BLOSUM 30
 A R N D C Q E G H I L K M F P S T W Y V B Z X *
A 4 -3 0 0 -2 0 -1 1 -2 -1 1 1 -5 -4 1 0 0 0 -7
R -1 8 -2 -1 -2 3 -1 -2 -1 -1 -3 -2 1 0 -1 -1 -1 -3 0 0 -1 2 0 -1 -7
N 0 -2 8 1 -1 -1 -1 0 -1 0 -2 0 0 -1 -3 0 1 -7 -4 -2 4 -1 0 -7
D 0 -1 1 9 -3 -1 1 -1 -2 -4 -1 0 -3 -5 -1 0 -1 -4 -1 -2 5 0 -1 -7
C -3 -2 -1 -1 17 -2 1 -4 -5 -2 0 -3 -2 -3 -3 -2 -2 -2 -6 -2 -2 0 -2 -7
Q 1 3 -1 -1 -2 8 2 -2 0 -2 -2 0 -1 -3 0 -1 0 -1 -1 -3 -1 4 0 -7
E 0 -1 -1 1 1 2 6 -2 -2 0 -3 -1 2 -1 -4 1 0 -2 -1 -2 -3 0 5 -1 -7
G 0 -2 0 -1 -4 -2 -2 8 -3 -1 -2 -1 -2 -3 -1 0 -2 1 -3 -3 0 -2 -1 -7
H -2 -1 -1 -2 -5 0 0 -3 14 -2 -1 -2 2 -3 1 -1 -2 -5 0 -3 -2 0 -1 -7
### Coordinate Products

| A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V | B | Z | X | * |
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R | 0 | 1 | 0 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 | 48 | 51 | 54 | 57 | 60 | 63 | 66 | 69 |
| N | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 |
| D | 0 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 | 42 | 45 | 48 | 51 | 54 | 57 | 60 | 63 | 66 | 69 | 72 | 75 |
| C | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 | 56 | 60 | 64 | 68 | 72 | 76 | 80 | 84 | 88 | 92 | 96 | 100 |
| Q | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 | 85 | 90 | 95 | 100 | 105 | 110 | 115 | 120 |
| E | 0 | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60 | 66 | 72 | 78 | 84 | 90 | 96 | 102 | 108 | 114 | 120 | 126 | 132 | 138 | 144 |
| G | 0 | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70 | 77 | 84 | 91 | 98 | 105 | 112 | 119 | 126 | 133 | 140 | 147 | 154 | 161 | 168 |
| H | 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80 | 88 | 96 | 104 | 112 | 120 | 128 | 136 | 144 | 152 | 160 | 168 | 176 | 184 | 192 |
| I | 0 | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90 | 99 | 108 | 117 | 126 | 135 | 144 | 153 | 162 | 171 | 180 | 189 | 198 | 207 | 216 |
| L | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 | 160 | 170 | 180 | 190 | 200 | 210 | 220 | 230 | 240 |
| K | 0 | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 | 110 | 121 | 132 | 143 | 154 | 165 | 176 | 187 | 198 | 209 | 220 | 231 | 242 | 253 | 264 |
| M | 0 | 12 | 24 | 36 | 48 | 60 | 72 | 84 | 96 | 108 | 120 | 132 | 144 | 156 | 168 | 180 | 192 | 204 | 216 | 228 | 240 | 252 | 264 | 276 | 288 |
| F | 0 | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 | 143 | 156 | 169 | 182 | 195 | 208 | 221 | 234 | 247 | 260 | 273 | 286 | 299 | 312 |
| P | 0 | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 | 140 | 154 | 168 | 182 | 196 | 210 | 224 | 238 | 252 | 266 | 280 | 294 | 308 | 322 | 336 |
| S | 0 | 15 | 30 | 45 | 60 | 75 | 90 | 105 | 120 | 135 | 150 | 165 | 180 | 195 | 210 | 225 | 240 | 255 | 270 | 285 | 300 | 315 | 330 | 345 | 360 |
| T | 0 | 16 | 32 | 48 | 64 | 80 | 96 | 112 | 128 | 144 | 160 | 176 | 192 | 208 | 224 | 240 | 256 | 272 | 288 | 304 | 320 | 336 | 352 | 368 | 384 |
| W | 0 | 17 | 34 | 51 | 68 | 85 | 102 | 119 | 136 | 153 | 170 | 187 | 204 | 221 | 238 | 255 | 272 | 289 | 306 | 323 | 340 | 357 | 374 | 391 | 408 |
| Y | 0 | 18 | 36 | 54 | 72 | 90 | 108 | 126 | 144 | 162 | 180 | 198 | 216 | 234 | 252 | 270 | 288 | 306 | 324 | 342 | 360 | 378 | 396 | 414 | 432 |
| V | 0 | 19 | 38 | 57 | 76 | 95 | 114 | 133 | 152 | 171 | 190 | 209 | 228 | 247 | 266 | 285 | 304 | 323 | 342 | 361 | 380 | 399 | 418 | 437 | 456 |
| B | 0 | 20 | 40 | 60 | 80 | 100 | 120 | 140 | 160 | 180 | 200 | 220 | 240 | 260 | 280 | 300 | 320 | 340 | 360 | 380 | 400 | 420 | 440 | 460 | 480 |
| Z | 0 | 21 | 42 | 63 | 84 | 105 | 126 | 147 | 168 | 189 | 210 | 231 | 252 | 273 | 294 | 315 | 336 | 357 | 378 | 399 | 420 | 441 | 462 | 483 | 504 |
| X | 0 | 22 | 44 | 66 | 88 | 110 | 132 | 154 | 176 | 198 | 220 | 242 | 264 | 286 | 308 | 330 | 352 | 374 | 396 | 418 | 440 | 462 | 484 | 506 | 528 |
| * | 0 | 23 | 46 | 69 | 92 | 115 | 138 | 161 | 184 | 207 | 230 | 253 | 276 | 299 | 322 | 345 | 368 | 391 | 414 | 437 | 460 | 483 | 506 | 529 | 552 |

User defined matrix (also BLOSUM 30)...
User DNA scoring scheme...

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Loading Score Matrices From File

This small demo program shows how to load a score matrix from a file. Examples for score file are core/demos/howto/scores/dna_example.txt for DNA alphabets and core/tests/score/PAM250 for amino acids.

Include the necessary headers.

```cpp
#include <iostream>
#include <seqan/basic.h>
```

508 Chapter 2. API Documentation
```cpp
#include <seqan/file.h>  // For printing strings.
#include <seqan/score.h> // The module score.

using namespace seqan;

We define a function that can show a scoring matrix.

// Print a scoring scheme matrix to stdout.
template <typename TScoreValue, typename TSequenceValue, typename TSpec>
void showScoringMatrix(Score<TScoreValue, ScoreMatrix<TSequenceValue, TSpec> > const & scoringScheme)
{
    // Print top row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i)
        std::cout << TSequenceValue(i);
    std::cout << std::endl;
    // Print each row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i) {
        std::cout << TSequenceValue(i);
        for (unsigned j = 0; j < ValueSize<TSequenceValue>::VALUE; ++j) {
            std::cout << TSequenceValue(i);  
            std::cout << score(scoringScheme, TSequenceValue(i), TSequenceValue(j));
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }
}

Finally, the main program loads the scoring matrix from the file given on the command line and then shows it.

int main(int argc, char **argv)
{
    if (argc != 2) {
        std::cout << "Invalid argument count!" << std::endl;
        return 1;
    }

    typedef int TScoreValue;

    Score<TScoreValue, ScoreMatrix<Dna, Default> > scoreMatrix;
    loadScoreMatrix(scoreMatrix, argv[1]);
    showScoringMatrix(scoreMatrix);
    return 0;
}

Here’s the program output.

$ make tutorial_load_score
$ ./demos/tutorial_load_score ../../demos/howto/scores/dna_example.txt
   A   C   G   T
A -1  -1  -1  -1
C -1   1  -1  -1
G -1  -1  1  -1
T -1  -1  -1  1
```
2.2.19 Writing App Tests

This HowTo describes the basics of writing your own app tests.

Overview

SeqAn application tests allow the simple testing of apps: the application is run several times, each with a different set of parameters and/or input files. The output to STDOUT and STDERR is captured. The captured output and all files written by the application are then compared against “gold standard” output files that were previously generated.

The general data flow for app tests is as follows (a simple working example are the app tests for the app `pair_align`).
There are two major steps involved: (A) Test data generation and (B) actually running the tests.

**Test Data Generation**

This is done once when preparing the test.

The program is run with different parameters (in the case of pair_align with different algorithms, scores, alphabets etc) and input files (for pair_align, e.g. for proteins and DNA sequences). The output files and the output to stdout (logs) are collected. The output and logs are then stored as files in the SVN repository and serve as the reference for future comparisons.

The expected output files are mostly generated by running a “sane” version of the program and then being checked for problems. Since there are a lot, they are usually generated by shell files, the generate_outputs.sh files. These files also serve as documentation which settings were used for which output files. Generally, they mirror the structure of the app test Python script (described below).

To reiterate, the shell script is usually only executed once when the tests are created. The output needs to be regenerated only when the program output changes and this change is deliberate. They are not regenerated on each test run.

Note that the path to the binary that is used to generate the output should be stored in a shell variable at the top of
the file. This allows for easily changing this. These shell scripts should also be well-documented. See the `generate_outputs.sh` file of `pair_align` or `tree_recon` for simple and mostly clean examples.

**Running Tests**

The app tests are then run in the nightly CMake builds and their results are submitted to CDash. There are two steps involved here: (1) Executing the programs and (2) comparing their result with the expected ones. There is a Python test driver program (called `run_tests.py` by convention) for each collection of app tests.

These programs use the Python module `seqan.app_tests` for running and usually mirror the corresponding `generate_outputs.sh` file.

**Creating App Tests**

We will create app tests for a small app that converts its argument to upper case and prints it to stdout.

**Setup App “upcase”**

First, create the app using `skel.py`.

```
$ ./util/bin/skel.py app upcase sandbox/$USER
```

Then, edit `upcase.cpp` to look as follows:

```cpp
#include <iostream>
#include <cstring>

using namespace seqan;

int main(int argc, char const ** argv)
{
    if (argc <= 1)
    {
        std::cerr << "No arguments given!" << std::endl;
        return 1;
    }

    for (int i = 1; i < argc; ++i)
    {
        for (char const * ptr = &argv[i][0]; *ptr != '\0'; ++ptr)
            std::cout << static_cast<char>(toupper(*ptr));
        std::cout << std::endl;
    }

    return 0;
}
```

Then, go to your build directory (here: `build/Debug`), build the app, and make sure it works correctly.

```
$ cd build/Debug
$ cmake .
$ cd sandbox/$USER/apps/upcase
$ make
$ ./upcase This is a test
THIS IS
```

512 Chapter 2. API Documentation
Creating App Tests

You can use the skel.py program to create the app tests.

```
$ cd ../../../../../..
$ ./util/bin/skel.py app_tests sandbox/$USER/apps/upcase/
```

As suggested by the output of skel.py, add the following to your sandbox/$USER/upcase/CMakeLists.txt:

```
# Add app tests if Python interpreter could be found.
if(PYTHONINTERP_FOUND)
  add_test(NAME app_test_upcase COMMAND ${PYTHON_EXECUTABLE}
            ${CMAKE_CURRENT_SOURCE_DIR}/tests/run_tests.py ${CMAKE_SOURCE_DIR}
            ${CMAKE_BINARY_DIR})
endif(PYTHONINTERP_FOUND)
```

Now, open the file sandbox/$USER/upcase/tests/generate_outputs.sh and modify it as follows.

```bash
#!/bin/sh
#
# Output generation script for upcase
UPCASE=../../../../../build/Debug/sandbox/holtgrew/apps/upcase/upcase
# ==============================================================
# Generate Output
# ==============================================================
${UPCASE} simple example > simple.stdout
${UPCASE} 'another()/' 'example!' > other.stdout
```

We now run the program two times with different arguments and stored the output in files simple.stdout and other.stdout. These files are kept in the directory sandbox/$USER/apps/upcase/tests and can now go into version control.

```
$ cd sandbox/$USER/apps/upcase/tests
$ ./generate_outputs.sh
$ head -1000 simple.stdout other.stdout
===> simple.stdout <===
SIMPLE
===> other.stdout <===
ANOTHER()/
EXAMPLE!
```

Now, we have the expected test output files. We now have to modify the test driver script run_tests.py. Open the file sandbox/$USER/apps/upcase/tests/run_tests.py. This file is a Python script that runs the programs, collects their output and compares the expected output prepared above with the actual one. It should look like the following:

```python
#!/usr/bin/env python
"""Execute the tests for upcase.

The golden test outputs are generated by the script generate_outputs.sh.
```

2.2. How Tos
You have to give the root paths to the source and the binaries as arguments to the program. These are the paths to the directory that contains the 'projects' directory.

Usage: run_tests.py SOURCE_ROOT_PATH BINARY_ROOT_PATH

```python
import logging
import os.path
import sys

# Automagically add util/py_lib to PYTHONPATH environment variable.
path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..', '..','..','..','..','..','util','py_lib'))
sys.path.insert(0, path)

import seqan.app_tests as app_tests

def main(source_base, binary_base):
    """Main entry point of the script."""

    print 'Executing test for upcase'
    print '========================='

    ph = app_tests.TestPathHelper(    
        source_base, binary_base,    
        'sandbox/holtgrew/apps/upcase/tests') # tests dir

    # Auto-detect the binary path.
    #===============================================================================
    path_to_program = app_tests.autolocateBinary(        
        binary_base, 'sandbox/holtgrew/apps/upcase', 'upcase')

    #===============================================================================
    # Built TestConf list.
    #===============================================================================

    # Build list with TestConf objects, analoguely to how the output was generated in generate_outputs.sh.
    conf_list = []

    #===============================================================================
    # First Section.
    #===============================================================================

    # App TestConf objects to conf_list, just like this for each test you want to run.
    conf = app_tests.TestConf(
        program=path_to_program,
        redir_stdout=ph.outFile('STDOUT_FILE'),
        args=['ARGS', 'MUST', 'BE', 'STRINGS', str(1), str(99),
            ph.inFile('INPUT_FILE1'),
            ph.inFile('INPUT_FILE2')],
        to_diff=[(ph.inFile('STDOUT_FILE'),
            ph.outFile('STDOUT_FILE')),
            (ph.inFile('STDOUT_FILE')),
            ph.outFile('STDOUT_FILE')]),
ph.outFile('INPUT_FILE1'))}
conf_list.append(conf)

# ==============================================================
# Execute the tests.
# ==============================================================
failures = 0
for conf in conf_list:
    res = app_tests.runTest(conf)
    # Output to the user.
    print ' '.join(['upcase'] + conf.args),
    if res:
        print 'OK'
    else:
        failures += 1
        print 'FAILED'

print '================================='
print ' total tests: %d' % len(conf_list)
print ' failed tests: %d' % failures
print 'successful tests: %d' % (len(conf_list) - failures)
print '================================='

# Compute and return return code.
return failures != 0

if __name__ == '__main__':
sys.exit(app_tests.main(main))

Here, we now mirror the generate_outputs.sh file by replacing the section First Section with the following:

# ==============================================================
# Run Program upcase.
# ==============================================================

# Simple Example.
conf = app_tests.TestConf(
    program=path_to_program,
    redir_stdout=ph.outFile('simple.stdout'),
    args=['simple', 'example'],
    to_diff=[[ph.inFile('simple.stdout'),
              ph.outFile('simple.stdout')]])
conf_list.append(conf)

# Another Example.
conf = app_tests.TestConf(
    program=path_to_program,
    redir_stdout=ph.outFile('other.stdout'),
    args=['another()/', 'example!'],
    to_diff=[[ph.inFile('other.stdout'),
              ph.outFile('other.stdout')]])
conf_list.append(conf)

Finally, we can run the program using ctest.

$ cd ../../..
$ cd build/Debug/sandbox/holtgrew/apps/upcase
$ ctest .
If everything goes well, the output will be as follows:

```
$ ctest .
Test project $(PATH_TO_CHECKOUT)/build/Debug/sandbox/holtgrew/apps/upcase
  Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ................. Passed 0.04 sec

100% tests passed, 0 tests failed out of 1
Total Test time (real) = 0.05 sec
```

In the case of failures, the output could be as follows.

```
$ ctest .
Test project /home/holtgrew/Development/seqan-trunk/build/Debug/sandbox/holtgrew/apps/upcase
  Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ..................***Failed 0.02 sec

0% tests passed, 1 tests failed out of 1
Total Test time (real) = 0.03 sec
```

The following tests FAILED:
  1 - app_test_upcase (Failed)

Errors while running CTest

The `ctest` command has many options. A useful one for debugging is `--output-on-failure` which will print the test result if the test does not succeed. For example, the output could be as follows:

```
$ ctest . --output-on-failure
Test project /home/holtgrew/Development/seqan-trunk/build/Debug/sandbox/holtgrew/apps/upcase
  Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ..................***Failed 0.02 sec
Traceback (most recent call last):
  File "/home/holtgrew/Development/seqan-trunk/sandbox/holtgrew/apps/upcase/tests/run_tests.py", line
    import seqan.app_tests as app_tests
  ImportError: No module named seqan.app_tests
```

0% tests passed, 1 tests failed out of 1
Total Test time (real) = 0.03 sec

The following tests FAILED:
  1 - app_test_upcase (Failed)

Errors while running CTest

This is a common error that tells us that we have to appropriately set the environment variable `PYTHONPATH` so the module `seqan.app_tests` is available from within Python.

You have to add `util/py_lib` to your PYTHONPATH. On Linux/Mac OS X, you can do this as follows (on Windows you have to set the environment variable, e.g. following this guide).

```
$ export PYTHONPATH=${PYTHONPATH}:PATH_TO_SEQAN/util/py_lib
```

Now, you test should run through:

```
$ ctest . --output-on-failure
...
2.2.20 Writing Commit Messages

Format

On every commit to our revision control system (currently SVN) please provide a commit message of the following form:

```
[CLASS1,CLASS2,...] Short description
Optional long description
```

- The first line starts with an arbitrary number of tags in square brackets, e.g. `[CLASS1]` or `[CLASS1,CLASS2]`. See below for a possible list of classes.
- These tags are followed by a short description, try to keep the first line below 120 characters, 80 if possible.
- You can add an optional long description after an empty line.

Possible Classes

NOP

Only whitespace changes.

- e.g. removed trailing whitespace, replaced tabs by spaces, changed indentation

DOC

Changes in the user documentation.

- This includes changes to the DDDoc documentation, README files etc.

COMMENT

Changes in the source documentation.

- These changes are not visible to the users.
  - This includes `TODO($name))` : statements.

API

Changes in the API.

- These changes classically break backward compatibility.
  - e.g. renaming of function names, changing of function parameter order.

INTERNAL
Changes in the implementation.
These changes do not influence the public the API.
e.g. renaming of variable names, simplification of code

**FEATURE**
A user-visible feature.
e.g. extension of an interface, measurable performance improvement
*If the change is also API breaking the classes FEATURE and API must be used.*

**FIX**
Bug removal.
If one or more bugs from the ticket tracker are removed then this should be written as \[\text{FIXED-#7,#35}\]
where #7 and #35 are ticket numbers.

**TEST**
Addition or changes of tests.
All code changes that are accompanied with tests must provide the original and the TEST class.
Don’t consider this as a coercion but as a privilege to use both classes!

**CLI**
Change to the command line interface of a program.
e.g. change to the arguments a program accepts, change to the messages printed to the user
*Output that is meant for debugging or detailed introspection is handled by the LOG class.*

**LOG**
Change of output for developers or very advanced users.
This is the output that is meant for debugging or detailed introspection that is excluded from CLI.
Such output is usually printed to stderr.

**Examples**

**Example: API Changes**

API change with tests and detailed description of changes.

\[\text{API,TEST}\] Large changes of align module’s API.

This is a large patch that mostly updates the align module:

* The Anchor Gaps specialization is moved from the store module to the align module.
* The Array Gaps class is rewritten.
* Both Anchor and Array gaps have mostly the same API now, differences are documented.
* Greatly unified the interface of the `''globalAlignment()''` and `''localAlignment()''` interface.
* The `''LocalAlignmentFinder''` is now called `''LocalAlignmentEnumerator''`.
* Threw out unused DP algorithms (DP algorithm will be cleaned up in the future, see below).
* Clipping of gaps works consistently in both Anchor and Array Gaps data structures.
* Wrote a large number of tests for all changes.
* Adjusted SeqAn library and apps to new API.

All in all, this should make the module more usable, albeit breaking the interface in some cases.
There will be a future change by Rene that strongly unifies the DP algorithms.
This will not inflict another API change, though, and further simplify the align module.
Example: Bug Fixes

A fix that solves two tickets:


Quite involved fix that allows iteration of "ModifiedString" objects.

A fix that does not have a ticket:


There was a bug when reading the operation "F", which was translated to FLABBERGASTED. Fixed this to the documented behaviour.

Example: Internal Changes

An internal change, reordering of code without changing the public API.

[INTERNAL] Reordering code in module sequence so no more generated forwards are needed.

An internal change might include test and improved comments.


Restructured the whole internal structure of the module, adding a large number of tests and improving the source-level documentation. The user level documentation is still lacking and should be the target of a future change.

Example: Changes To Command Line Interface And Logging

Changes to the command line interface:

[CLI] Changed output of STELLAR such to unify scientific notation floats.

Changes to logging in an app:

[LOG] Improved logging in RazerS 5.

Much more detailed logging allows easier debugging. Part of this should probably be commented out before the next stable release once the dust has settled and most bugs have been removed.
2.2.21 Writing Nice Unix Programs

In bioinformatics, many programs are of “academic” quality, i.e. they are written to present a new method. The implementation is often “only” used by other academics who, since they do not pay for the program, are willing to take some glitches for granted.

This page tries to help academic software developers in writing better Unix programs. The points raised here come from our own experience with using academic software. The focus is on C and C++ programming in a Unix (e.g. Linux, Mac Os X) environment. The hints should also be applicable to other languages such as Java, and in some way also Windows.
Program Return Codes

Rationale

The `main()` method of a program should be 0 if there were no errors and a value different from 0 otherwise.

Explanation & Reasoning

The `main()` function should return an integer indicating whether the program completed running successfully or not. A value of 0 indicates that no error occurred, a value not equal to 0 indicates that an error occurred. You might consider returning different values for different kinds of errors, e.g. 2 for I/O errors, 3 for logic errors and 1 as a catch-all for any all errors.

This makes it easy for a calling script/program to check whether the program executed successfully.

Example

The following program returns 1 if it receives any argument and 0 if this is not the case.

```c
#include <cstdio>

int main(int argc, char ** argv)
{
    if (argc > 1) {
        fprintf(stderr, "I do not like arguments!\n");
        return 1;
    }

    return 0; // Everything went smoothly!
}
```

The following bash script calls programs and reacts to the return code.

```bash
#!/bin/sh

# 1. Only success case.
program arg1 arg2 && echo "success!"

# 2. Only failure case.
{| ! echo "failure" |}

# 3. Handle success/failure case
program arg1 arg2
if [ "$?" ]; then
    echo "success"
else
    echo "failure"
fi

# 4. Use case for separating cases
# TODO
```
Assume Few Things About Paths

Rationale

Do not assume anything on paths for (1) the program to reside in (2) temporary files or (3) the working directory. Fixing the program install directory at configure time is OK.

Explanation

Most Unix programs are configured with a $PREFIX variable (e.g. setting --prefix= in the ./configure script) and assume that all paths are relative to the given one. For example, the Apache 2 web server reads its configuration from the directory ${PREFIX}/apache2. This is a reasonable assumption. Another reasonable assumption is that the current working directory is writeable. However, temporary files should be stored in ${TMPDIR} or /tmp (see the related section).

Non-reasonable assumptions are:

- The program is executed in the directory the binary resides in. For example, program prog at path /path/to/prog should not assume that the working directory is /path/to when it is executed. Especially, do not assume that the directory the binary resides in is writeable. If your program is installed in /usr/bin, this path is non-writeable for normal users on Unix.

- A program must be in a given specific path fixed at code writing time. While it is reasonable for the user to give an install path at configure-time, the user should be able to install the program in any directory, including /opt, his ${HOME} directory or /some-weird-path/the/sys/admin/gave.

Best practice is:

- Use ${TMPDIR} if available, fall back to /tmp, for intermediate/temporary files.

- Use reasonable defaults for result files, e.g. the path the input file resides in.

- Allow the user to set an output directory.

- Consider asking the user before overwriting result files when using defaults.

Example

Some programs create the result files in the current working directory. This is not good practice, since the current working directory is context dependent. While it is possible to use pushd and popd to use one directory per call to the program, it is much less error prone and more comfortable for the caller to specify the file on the command line.

Provide Good Defaults

Rationale

Require as few parameters as possible, provide defaults or guess as many as possible.
Explanation

The more parameters are required in a program, the hard it gets to use. For many parameters, default values can be given by the program’s author. Other parameters can be guessed depending on the input.

It should still be possible to override such value by command line parameters.

Example

The quality type of a FASTQ file can be guessed from the file contents very reliably by looking at the quality entries. Nevertheless, the user should be able to override this by a command line parameter.

Positional vs. Named Arguments

TODO

Provide all-in-one-go Variants of your program

Rationale

While many program’s steps might add to flexibility, a tool is easier to use if there is only one call to it.

Explanation

Some bioinformatics programs consist of many steps, e.g. (1) building an index (e.g. k-mer or suffix array) (2) perform a search, and (3) combine multiple search results to one. While this might enable the flexible usage of the program it complicates its usage. Please also provide a way to call your program that creates an output from the input files with one program call.

Example

For paired-end read mapping, the program \textit{bwa} consists of multiple calls.

1. Call bwa to build an index on your genome.
2. Map the left-end reads, yielding a position file.
3. Map the right-end reads, yielding a position file.
4. Combine the two position files previously created.

While it is OK to first create an index file (this file can be used for many reads files), the last three steps could be combine into one umbrella command. This would reduce the number of intermediate files and be much more comfortable for users.

Use stdout and stderr correctly

Rationale

The standard stream \texttt{stdout} is for the program’s output while \texttt{stderr} is for logging and error messages. It should be possible to redired \texttt{stdout} to an output file and \texttt{stderr} to a log file. Use \texttt{-} as shortcuts for \texttt{stdout} and \texttt{stderr}. 

2.2. How Tos
Explanation

In C/Unix programming `stdout` is for output to the user, `stderr` is for error messages and logging. For example, when running daemons (e.g. web servers), the output to `stderr` ends up in log files.

If your program has only one input and one output file, it could accept the input from `stdin` by default and write to `stderr`. An example is the `grep` tool on Unix. You can specify different programs on the command line, however.

If you have program arguments for input and output files then you should use `--` for shortcuts to `stdin` and `stderr`. An example is a call to `grep --in-file - --out-file -` would read from `stdin` and write to `stdout`.

Example

- When the program is called with wrong parameters, the return code should not be 0 and the help should be printed to `stderr`.
- When the program is called with a `--help` parameter, the return code should return 0 and the help should be printed to `stdout`.

Allow specifying all file names through the command line

TODO

Do Not Require A Specific Working Directory

Rationale

Do not require that the current working directory is in any relation to the directory containing the binary.

Explanation

Some programs must be called with `.program`, e.g. the current working directory. This makes it harder to use the program when installed centrally and when multiple instances are called at the same time on the same file system. This makes it harder to use in complex software pipelines. Here, additional working directories and either symbolic links or copies of the program binary have to be created for each called instance.

Use `$TMPDIR` For Temporary Files, Fall Back to `/tmp`

Rationale

Use the value of the environment variable `${TMPDIR}` for temporary files. If it is not set, use `/tmp` or `/var/tmp`.

Explanation

On Unix, the canonical place to store temporary file is the value of the environment variable `${TMPDIR}`. If it is not set, then use `/tmp` or `/var/tmp`. `/tmp` might be cleared during system reboots while `/var/tmp` is not cleared during system reboots but possibly rather depending on the file age.
2.2.22 Writing Tests

This page describes how to write tests for the SeqAn library. Each test program defines a Test Suite, a collection of related Tests.

Test Suite Skeleton / Example

A skeleton and example for a test suite program look as follows:

```cpp
#include <seqan/basic.h>

SEQAN_DEFINE_TEST(test_suite_name_test_name)
{
    int ii = 1;
    for (int jj = 0; jj < 10; ++jj)
    {
        ii *= 2;
    }
    SEQAN_ASSERT_EQ(ii, 1024);
}

SEQAN_BEGIN_TESTSUITES(test_suite_name)
{
    SEQAN_CALL_TEST(test_suite_name_test_name);
}
SEQAN_END_TESTSUITES
```

Be Consistent
Tests Should Compile Without Warnings
Break Your Tests Down
Use Helper Functions For Setup/TearDown
Comment Your Tests
SEQAN_BEGIN_TESTSUITE(...) and SEQAN_END_TESTSUITE are macros that expand to book-keeping code for running a test suite. SEQAN_DEFINE_TEST(...) expands to the definition of a function that runs a test.

Getting Started With Our Test Template

To make creating tests easier, the code generator skel.py has a command to generate test skelletons for you. As parameters, you give it the name of the module you want to test and the path to the repository (core, extras or a sandbox). For example, use skel.py tests my_module sandbox/my_sandbox to create tests for the module my_module in the sandbox sandbox/my_sandbox:

```bash
sean $ ./util/bin/skel.py test my_module sandbox/my_sandbox ...
```

sandbox/my_sandbox/tests/my_module/
-- CMakeLists.txt
-- test_my_module.cpp
-- test_my_module.h

Afterwards, you can compile and run the tests:

```bash
$ mkdir -p build/Debug
$ cd build/Debug
$ cmake ../..
$ make test_my_module
$ ./core/tests/my_module/test_my_module ...
```

Note that when adding new tests then you have to add them to the dependencies of the test target in sandbox/my_sandbox/tests/my_module/CMakeLists.txt.

Test Macros

Inside your tests, you can use the SEQAN_ASSERT*, and SEQAN_ASSERT_*_MSG macros to check for assertions. Other useful macros are SEQAN_PATH_TO_ROOT and SEQAN_TEMP_FILENAME.

The macros themselves are documented in the dox:SeqAn API documentation AssertMacros.

Assertion Caveats

When using one of the LT/GT/LEQ/GEQ/EQ/NEQ macros, the values have to provide a stream operator (operator<<) to write them to an output stream. If this is not implemented, then the assertion will not compile and something like the following will be printed by the compiler (in this case the GCC).

```bash
In file included from seqan/basic.h:55:0,
  from tests/sequence/test_sequence.cpp:4:
seqan/basic/basic_testing.h: In function ‘bool ClassTest::testEqual(const char*, int, const T1&, const char*, const T2&, ... [with T1 = Iter<String<char, Block<3u> >, PositionIterator], T2 = Iter<String<char, Block<3u> >, PositionIterator]’:
seqan/basic/basic_testing.h:435:81: instantiated from ‘bool ClassTest::testEqual(const char*, int, const T1&, const char*, const T2&, ... [with T1 = Iter<String<char, Block<3u> >, PositionIterator], T2 = Iter<String<char, Block<3u> >, PositionIterator]’
tests/sequence/test_string.h:386:2: instantiated from ‘void TestStringBasics() [with TMe = String<char, Block<3u> >]’
tests/sequence/test_string.h:475:45: instantiated from here
seqan/basic/basic_testing.h:385:13: error: no match for ‘operator<<’ in ‘std::operator<< [with _Traits = ... Assertion failed : “)))), expression1))), ((const char*)” == “)))), expression2))), ((const char*)” was: ”) << value1'
```

The workaround is to use

```bash
SEQAN_ASSERT(end(str3) == begin(str3) + 7);
```

instead of
SEQAN_ASSERT_EQ(end(str3), begin(str3) + 7);

**Best Practices**

**Rules are there to make you think before you break them.** The following is not written into stone, but should be good guidelines. Improvements to the best practices is welcome.

**Be Consistent**

Whatever you do: Be consistent. If the one has read one part of your code then one should not have to adjust to different variable and function naming, comment style etc.

**Tests Should Compile Without Warnings**

Make sure that your tests compile without warnings. A common warning is “comparison of signed and unsigned integer”.

In many places, the problematic line looks like this

SEQAN_ASSERT_LT(length(ndl), 30);

The `length` function returns an unsigned integer while the string literal `30` represents a (signed) `int`. You can fix this by changing the type of the number literal:

SEQAN_ASSERT_LT(length(ndl), 30u);

**Break Your Tests Down**

Each test should isolate target an as small as possible and/or feasible unit of your code. Having short test functions makes them easier to read and maintain.

Another advantage is that bogus state does not leak into other tests: imagine, you have a test that tests a function `assign_if_positive(a, b)` that assigns `b` to `a` if `b` is positive.

SEQAN_DEFINE_TEST(test_assign)
{
    int x = 0;

    assign_if_positive(x, 5);
    SEQAN_ASSERT_EQ(x, 5);

    assign_if_positive(x, -7);
    SEQAN_ASSERT_EQ(x, 5);
}

Now, what happens if `assign_if_positive(...)` has a bug and *never* assigns a value to its first parameter or always assigns 1? Both of your assertions will fail. This means you do not really know in which case the function works well and in which case it does not work well.

Splitting the test make it more robust:
SEQAN_DEFINE_TEST(test_assign_positive)  
{  
    int x = 0;  
    assign_if_positive(x, 5);  
    SEQAN_ASSERT_EQ(x, 5);  
}  

SEQAN_DEFINE_TEST(test_assign_negative)  
{  
    int x = 0;  
    assign_if_positive(x, -7);  
    SEQAN_ASSERT_EQ(x, 0);  
}  

Use Helper Functions For Setup/TearDown  

If you need to initialize the same state for multiple tests, then the code for this should only exist once. This makes it easier to maintain since we do not have to change it in multiple places at once. This is especially useful when following the best practice Break Your Tests Down.  

Example:  

Instead of  
SEQAN_DEFINE_TEST(test_grep)  
{  
    char *contents = loadFile("corpus.txt");  
    int pos = doGrep(contents, "nonexisting pattern");  
    SEQAN_ASSERT_EQ(pos, -1);  
    pos = doGrep(contents, "existing pattern");  
    SEQAN_ASSERT_EQ(pos, 3);  
    delete contents;  
}  

do  
  // Set-up for test_grep_{success, failure}.  
void testGrepSetUp(const char *filename, char *outContents)  
{  
    outContents = loadFile(filename);  
}  

  // Tear-down for test_grep_{success, failure}.  
void testGraphTearDown(char *contents)  
{  
    delete contents;  
}  

  // Test greping for existing patterns.  
SEQAN_DEFINE_TEST(test_grep_success)  
{  
    // corpus.txt contains the string "1234existing pattern567".  
    char *contents;  
    testGrepSetUp("corpus.txt", contents);  
}
```cpp
int pos = doGrep(contents, "existing pattern");
SEQAN_ASSERT_EQ(pos, 3);

testGrepTearDown(contents);
}

// Test greping for non-existing patterns.
SEQAN_DEFINE_TEST(test_grep_failure)
{
    // corpus.txt contains the string "1234existing pattern567".
    char *contents;
    testGrepSetUp("corpus.txt", contents);

    int pos = doGrep(contents, "nonexisting pattern");
    SEQAN_ASSERT_EQ(pos, -1);

    testGrepTearDown(contents);
}
```

**Comment Your Tests**

Tests can complement examples from the documentation in that they illustrate each call to your code’s API. Thus, make sure that your tests are well-documented. Not only for users who look up how to use your code but also for the next maintainer.

There should be a documentation of the test itself and also inline comments. In your comments, you should focus on the maintainer and not so much the user. Even if some things are obvious, you might want to illustrate why you call a function with the given parameters, e.g. describe the corner cases.

Example:

```cpp
// Test abs() function with 1, a representant for positive values.
SEQAN_DEFINE_TEST(test_abs_with_one)
{
    SEQAN_ASSERT_EQ(abs(1), 1);
}

// Test abs() function with 0, the only corner case here.
SEQAN_DEFINE_TEST(test_abs_with_zero)
{
    SEQAN_ASSERT_EQ(abs(0), 0);
}

// Test abs() function with -1, a representant for negative values.
SEQAN_DEFINE_TEST(test_abs_with_minus_one)
{
    SEQAN_ASSERT_EQ(abs(-1), 1);
}
```

**2.2.23 Writing Tutorials**

At the bottom, you can find a Tutorial Template for starting a new tutorial.
Conventions

Wiki Conventions

- Use only one line per sentence. This increases the readability of the sources.

Naming Conventions

- Use headline capitalization for headlines.
- Use the tutorial’s title as the file name (e.g. /wiki/Tutorial/NameOfYourTutorial.rst).
- Assignments are numbered in the order they appear in a tutorial (e.g. Assignment 5). Do not use a section relative numbering but an absolute one. If, e.g., the last assignment of section 1 was assignment 3, the first assignment of section 2 is assignment 4).
- Place the assignment’s solutions inline.

Design & Layout Conventions

- Use back ticks (``) to denote names of variables, functions, etc. (e.g. ```append``` results in append).
- Use bold font (**word**) to denote key concepts.
- Use item and menu > sub menu > item to denote GUI entries and menu paths.
- Use the following markup to include source code

  .. includelfags:: core/demos/tutorial/alignment/alignment_msa.cpp
     :fragment: init

  where extras/demos/tutorial/alignment/alignment_msa.cpp gives the source code file in the repository and init the fragment to include in the tutorial.
- You should always build and test the tutorials code snippets before using them.

  manual # make html

- Use the following markup to format screen output:

  ::

  # Hello World!

- Use the following markup to inform about important bugs or other relevant issues. The content (and thereby the box itself) is always of temporary nature and should only be used thriftily:

  .. warning::

  Warning goes here.

- Use the following markup to give important information.

  These boxes contain information that should be kept in mind since the described phenomenon is very likely to be encountered by the reader again and again when working with SeqAn. In contrast to the .. warning::, this box type is of permanent nature and the given information are valid for a long time.

  .. important::

  Important information goes here...
Use the following markup to give further /optional information/. These are information that support the understanding but are too distinct to be put in a normal paragraph:

```plaintext
.. hint::

    Optional information goes here.
```

- Use the following markup to format assignments (for further details see Assignments):

```plaintext
.. container:: assignment

    The assignment goes here.
```

- Use `:dox:'DocItem'` to create links to the SeqAn API dox documentation.

### Important:
Note that this will merely generate the URLs that `dddoc` would create but does not perform any checking. Some examples:

- `String (:dox:'String')`
- `AllocString (:dox:'AllocString')`
- `Alloc String (:dox:'AllocString Alloc String')`
- `SequenceConcept (:dox:'SequenceConcept')`

---

**Structure**

**Meta Information**

Place the directives for the side bar and the link target for the tutorial page directly before the tutorial title.

```plaintext
.. sidebar:: ToC

.. contents::

.. _tutorial-sequences:
```

**Sequences**

---

Based on the Tutorial Template, provide information regarding:

- **learning objective** Describe the learning objective in your own words.
- **difficulty** Valid values: Very basic, Basic, Average, Advanced, Very advanced
- **duration** In average how much time will a user spend on absolving this tutorial? If you expect more than 90 minutes please split your tutorial up into multiple ones.
- **prerequisites** A list of absolved tutorials and other requirements you expect your reader to fulfill.

**Introduction**

In the next paragraph introductory information are given that answer the following questions:

- What is this tutorial about?
- Why are the information important?
• What are the communicated information used for?
• What can the reader expect to know after having absolved the tutorial?

Section

Introduction

In each section’s introduction part you answer the following questions:
• What is this section about?
• What are the central concepts in this section?
• What is your partial learning objective?

Explanations / Examples

The main part consists of the description of the topic. This is the space where enough knowledge is transmitted to enable the reader to solve all assignments. Further details are contained in the Tutorial Template and in the didactics section.

Try not to get lost in details. If you have useful but still optional information to give use a .. note:: directive.

Assignments

The assignments’ purpose in general is to support the reader’s understanding of the topic in question. For each assignment is of a special type (Review, Application and Transfer), has an objective, hints and a link to the complete solution.

Depending on the assignment’s type the reader is guided through the assignment solving by providing him with partial solutions.

There must always be an assignments of type Review. Assignments must always appear in an ascending order concerning their types and no “type gap” must occur.

Thus the only valid orders are:
• Review
• Review, application
• Review, application, transfer

The order Review, transfer is invalid since a “type gap” (application type missing) occurred.

All assignments must be accompanied by a solution.

Further Section

as many further sections as you like
Didactics

Type

As already mentioned in the assignment structure description each assignment is of one type.

These levels are

**Review** knowledge fortification (mainly through repetition, optionally with slight variations)

**Application** supervised problem solving (finely grained step-by-step assignment with at least one hint and the interim solution per step)

**Transfer** knowledge transfer (problem solving in a related problem domain / class)

Based on the chosen level you should design your assignment.

Duration

The time needed to absolve a tutorial must not exceed 90 minutes. Split your tutorial up (e.g. Tutorial I, Tutorial II) if you want to provide more information.

Language

Make use of a simple language. This is neither about academic decadence nor about increasing the learning barrier. You are not forced to over-simplify your subject but still try to use a language that is also appropriate for those who don’t fully meet the tutorials prerequisites.

Mental Model

When your describe and explain your topic give as many examples as possible. Try to adopt the reader’s perspective and imagine - based on your target group and prerequisites - your reader’s mental model. The mental model can be described as an imagination of the interaction of central concepts. Try to support the reader in developing a mental model that fits best to your topic.

Integration

- Add a link to your tutorial to `Tutorial.rst` and add a link to the .. toctree.
- Above you stated the tutorials your tutorial has as prerequisites. Add the link in a way that all required tutorials are listed above your tutorial.

Tutorial Template

```rst
.. sidebar:: ToC

.. contents::

.. _tutorial-tutorial-template:

Tutorial Template
------------------
```

2.2. How Tos
Learning Objective
Describe the learning objective in your own words.
**Example:**
You will be able to write a tutorial that meets our quality standards.

Difficulty
[Very basic, Basic, Average, Advanced, Very advanced]
**Example:**
Basic

Duration
In average how much time will a user spend on absorbing this tutorial?
If you expect more than 90 minutes please **split your tutorial up** into multiple ones.
**Example:**
1 h

Prerequisites
A list of absolved tutorials and other requirements you expect your reader to fulfill.
**Example:** :ref:`tutorial-first-steps-in-seqan`, :ref:`tutorial-pattern-matching`, English language

This is the place where introductory need to be in given, e.g. "This page constitutes the template for all future SeqAn tutorials.

Use this and optional further paragraphs to give the following information:

* What is this tutorial about?
* Why are the information important?
* What are the communicated information used for?
* What can the reader expect to know after having absolved the tutorial?

.. warning::

    This is a warning message.

    Here you can inform users about important bugs or other relevant issues.

Section
^^^^^^^

Use this and optional further paragraphs to give the following information:

* What is this section about?
* What are the central concepts in this section?
* What is your partial learning objective?

When your describe and explain your topic give **as many examples as possible**.
Try to adopt the reader’s perspective and imagine - based on your target group and prerequisites - your **reader’s mental model**.
The mental model can be described as an imagination of the interaction of central concepts.
Use a **simple language** and try to support the reader in developing a mental model that fits best to the tutorial.

.. tip::

    What are tips for?

    An ".. tip" ist useful to give information that are **optional** and thus don’t need to be read.
    Typical information are **further details** that support the understanding but are too distinct to be put in a normal paragraph.

    In this example you could tell the reader more about didactics and give him some useful links.
.. important::

What are importants for?

These boxes contain information that **should be kept in mind** since the described phenomenon is very likely to be encountered by the reader again and again when working with SeqAn.

If you give code examples tell the reader what he can see and what is crucial to your snippet.

Link all classes and other resources to the SeqAn documentation system by using `":dox:`Item` (e.g.`:dox:`String). In order to include code snippets use `":includefrags:: path``.

.. includefrags:: core/demos/tutorial/alignments/alignment_banded.cpp
   :fragment: alignment

If possible also include the generated output by given code in the console.
Here is one example:

.. code-block:: console

   0: ACAG
   1: AGCC
   2: CCAG
   3: GCAG
   4: TCAG

Now that you gave an overview of important concepts of your topic let the user play with it!
Formulate **small assignments** to allow the reader to fortify his newly acquainted knowledge.

Assignment 1

***********

.. container:: assignment

Type

[Review, Application, Transfer]

Note that your readers will be in different phases of learning. For the sake of simplicity we restrict ourselves to the following three levels:

#. knowledge fortification (mainly through repetition, optionally with slight variations)
#. supervised problem solving (finely grained step-by-step assignment with at least one hint and the interim solution per step)
#. knowledge transfer (problem solving in a related problem domain / class)

**Example:** Application

Objective

The objective of the assignment.

**Example:**

Output all symbols a given alphabet can have.
The output should look like this: ...

Hints

...

Solution

.. container:: foldable

Foldable solution with description.

This part of the assignment is to give partial solutions.
A partial solution starts with a sentence of what this step is about and gives the lines of code that are needed to implement this step.

Solution Step 1
.. container:: foldable
   The given sequence are of alphabet...
   Therefore, you have to...

   .. includerfrags:: core/demos/tutorial/alignments/alignment_banded.cpp
      :fragment: main

Solution Step 2
.. container:: foldable
   The given sequence are of alphabet...
   Therefore, you have to...

   .. includerfrags:: core/demos/tutorial/alignments/alignment_banded.cpp
      :fragment: fragment

2.3 Infrastructure

This part describes the infrastructure and software tools supporting the SeqAn project. This includes a description of the repository structure, an overview of the CMake-based build system and the nightly build (aka continuous integration) system.

### Contents

- The CMake-Based Build System
  - Directory Layout
  - Target Structure
  - External Dependencies
  - Adding New Programs
  - Multiple Build Types

2.3.1 The CMake-Based Build System

We use CMake for building the SeqAn demos, applications and tests. This wiki page explains our usage of CMake, the variables we define and how to extend the build scripts for new demos, apps etc. CMake’s documentation (v2.8) supplements this document. The documentation of CTest (v2.8) could also be of interest.

Directory Layout

The CMake files live in projects/library/cmake:

```bash
$ cd projects/library/cmake
$ tree
.
|-- CMakeLists.txt
```

536 Chapter 2. API Documentation
Target Structure

There is a target for each program to be built.

Additionally, there is a target called Seqan that represents the library. When using the GCC, we need to build the generated forward headers. In this case, Seqan also depends on the generated forward headers and the CMakeList.txt files define how to generate these generated forwards.

External Dependencies

SeqAn is a C++ header library and thus does not need to build itself. However, some applications have dependencies on external libraries, such as Boost or Threading Building Blocks.

The policy is to install these external dependencies on your system and let CMake find them.

The policy for missing dependencies is not to build the programs that depend on them and print an error message.

Adding New Programs

The process of adding a new demo, test or app is really simple: create a new directory my_app under projects/library/app, “my_test” under projects/test or my_demo under projects/demos. Within this directory create a new file my_app.cpp, my_test.cpp or my_demo.cpp and write your program.

Go to projects/library/cmake in your shell and execute cmake . again. The new target will appear in your IDE. If you use Makefiles then you can now type make my_app, make my_test or make my_demo.

Multiple Build Types

You can call CMake in different directories to be able to build Debug, Release etc. binaries without having to re-cmake. The process is described here in the CMake wiki.
### 2.3.2 Documentation Infrastructure

The documentation of SeqAn has two parts: (1) the API documentation and (2) the manual that you are reading right now.

**SeqAn API Documentation**

The SeqAn API documentation is created using a customly-written system called *dox*. You can find out more about the syntax in *Dox API Docs*.

You can build the documentation in the *dox* folder:

```bash
dox # ./dox_only.sh
```

**SeqAn Manual**

The SeqAn manual is created using the *Sphinx* documentation system. You can build it from the *manual* folder:

```bash
manual # make html
```

Note that you have to first build the dox documentation since plugins for generating the `:dox:` links rely on the generated search index for checks.

### 2.3.3 SeqAn Repository Structure

This article describes the SeqAn repository structure. After reading it, you will have knowledge about the repository structure and the reasons for the design decisions.

Note that this article describes the structure of the Subversion repository, not the structure of the release version of SeqAn which you can download as a ZIP archive.

**Overview**

The main repository structure is shown in the following picture.
The repository root contains some information files such as the LICENSE, README, and GETTING_STARTED files. The file names should speak for themselves.

- The folder core contains the core area with apps, tests, and library modules that are (1) stable and (2) of general interest. Furthermore, it contains demos for the library modules in the SeqAn core.
- The folder extras contains the extras area with apps, tests, and library modules that are either (1) not stable enough yet or (2) of special interest only, as well as demos for the SeqAn extras library modules.
- The folder sandbox contains the sandbox area. Users can create their own user areas inside this folder as described in the section Sandboxes.
- The folders docs and docs2 contain the scripts for the documentation system. At the moment, there are two concurrent documentation systems. In the midterm future, we aim to replace this by one new documentation system.
- The folders misc and util contain miscellaneous files and utility code. For example the Code Generator Python scripts are located here as well as SeqAn logo image files and CMake modules.

Core Area

The core area is structured as follows. Note that we generally refer to such a structure as a repository below.

- The apps directory contains applications. Each application directory contains the source files for one or more binaries, documentation, example files, and app tests. More information is available in the section Application Structure.
- The demos directory contains demo programs. The CMakeLists.txt file in this directory is written such that each file ending in .cpp is compiled into an executable with default SeqAn flag options.
• The include directory contains library modules. This is described in more detail in the section Library Modules.

• The tests directory contains tests for library modules. For each library module, there is a directory below tests with the same name that contains the tests for this module. Simpler modules have one tests executable, whereas there might be multiple tests executables for larger modules. For example, the module index has multiple test programs test_index_qgram, test_index_shapes etc. Writing tests is explained in detail in the article Writing Tests.

Application Structure

Each application directory contains one CMakeLists.txt file and the files for compiling one binary. Usually, apps have tests, too. In this case, there is a subdirectory tests. Writing application tests is covered in detail in the article Writing App Tests.

The general structure of an app is as follows:

```
seqan/apps/razers
|-- CMakeLists.txt  CMake script file
| |-- README       Documentation and License Files
| |-- LICENSE
| |-- example      Small Example Files
|   |-- genome.fa  
|   |-- reads.fa   
|     `-- ...
| `-- razers.cpp   Source Files for Executables
|`-- razers.h      
| |-- ...         
| `-- tests       App Tests Files
```

Library Modules

The library modules area looks as follows:

```
seqan/core/include
|-- seqan
| |-- basic/   Library Module basic
| | `-- ...    
| `-- ...      
| `-- sequence/ Library Module sequence
| `-- ...      
| `-- ...      Other Library Modules
```

On the top level, there is the folder seqan that contains the library modules. Inside the folder seqan, there is one directory and one header for each module.

The folder <module-name> contains the headers for the module module-name. The header <module-name>.h includes the headers from the module module-name. Including the header makes the code in the module available.
Extras Area

The extras area has the same “repository” structure as the core area. The main difference is that it contains code that is not stable enough or not of general interest.

Sandboxes

The sandbox area is a location where users can place their own repositories (i.e. directory trees having the same structure as the core and extras area) into. Currently, the sandboxes are also stored in the SeqAn SVN repository but that will change in the near future. Sandboxes can be generated using the skel.py Code Generator.

The following example shows how to create a user sandbox in the sandboxes area in an already existing Subversion repository. We assume that https://svn.example.com/trunk is an empty directory in a Subversion repository.

```bash
seqan # cd sandbox
sandbox # svn co https://svn.example.com/trunk sandbox_example
...
sandbox # cd ..
seqan # /util/bin/skel.py --force repository sandbox/sandbox_example
```

Next, we can create an application from a simple template in this sandbox:

```bash
seqan # /util/bin/skel.py app first_app sandbox/sandbox_example
```

Finally, commit this new sandbox into your Subversion repository:

```bash
seqan # cd sandbox/sandbox_example
seqan # svn add *
...
seqan # svn commit -m "Initial sandbox structure with one app."
...
```

Note that for the Subversion repository containing sandboxes, we recommend the following layout. Using the classic SVN trunk, tags, branches structure allows for tagging releases or points of returns. Furthermore, you can create folders parallel to those for documentation (for example a folder slides parallel to trunk) without polluting your repository structure:

```
Subversion repository root
|-- trunk
 | |-- CMakeLists.txt
 | |-- apps
 | |-- demos
 | |-- include
 | `-- tests
|-- tags
 `-- branches
```

Documentation System

The folders docs and docs2 are used for building the documentation with the old and the new documentation system. You can build them by going into the directory and then calling ./make.sh. This will build the documentation into the sub directory html:

```bash
seqan # cd docs
docs # ./make.sh
...
```
If you want to include documentation for code from your sandbox then you can pass the path to the library (or library module) in your sandbox as a parameter to ./make.sh:

docs2 # ./make.sh ../sandbox/sandbox_example/include
...
This page gives an example of how to use SeqAn in your application based on your own Makefiles. You should be able to adapt the descriptions to configuring your build system and/or IDE.

**Tip:** SeqAn is a header library only. Simply add core/include and extras/include to your include path and you can use SeqAn, as seen in the Short Version. See below how to enable using zlib for BAM access, for example.

### Libraries on Linux

On Linux, you have to link against librt. For GCC, add the flag \-lrt to the g++ compiler call.

### Compiler Flags

It is recommended to compile your programs with as many warnings enabled as possible. This section explains which flags to set for different compilers.

#### GCC

For GCC, the following flags are recommended:

- \-W \-Wall \-Wno-long-long \-pedantic \-Wno-variadic-macros

Explanation:

- \-W \-Wall \-pedantic Maximal sensitivity of compiler against possible problems.
- \-Wno-variadic-macros The assertion macros are variadic. Variadic macros were standardized in C99 but are not part of C++98 so GCC warns against their usage. Disable these warnings.
- \-Wno-long-long 64 bit integers (long long) are not supported in C++98, but GCC implements them nevertheless but warns against their usage in pedantic mode. We really want 64 bit integers, though.

#### Visual Studio

For Visual Studio, the following flags are recommended:

/W2 /wd4996 -DCourtesyGracefulWarnings

Explanation:

/W2 Warning level 2 is pretty verbose already. In the future, we will support level 3 without warnings in SeqAn code.

/wd4996 Allows the use of some deprecated functions without warnings.

-DCourtesyGracefulWarnings :::: Some C functions like sprintf are prone to incorrect usage and security holes. Replacing such calls does not have a high priority right now since SeqAn is usually not used on servers facing the outside world.

### Preprocessor Defines Affecting SeqAn

There are certain preprocessor symbols that affect the behaviour of SeqAn.
SEQAN_ENABLE_DEBUG

possible value 0, 1
default 1

meaning If set to 1, assertions within SeqAn (SEQAN_ASSERT...) are enabled, they are disabled otherwise. Is forced to 1 if SEQAN_ENABLE_TESTING is true. If not set, is set to 0 if NDEBUG is defined and set to 1 if undefind and NDEBUG is not defined.

SEQAN_ENABLE_TESTING

possible value 0, 1
default 0

meaning If set to 1, checkpoints are enabled. This makes the code very slow, however, and should only be used when running the tests. Has to be set to 1 for tests to work.

SEQAN_HAS_BZIP2

possible value 0, 1
default 0

meaning If set to 1 then libbzip2 is available." You have to link against the library (e.g. add -lbz2 to your linker flags) and bzlib.h must be in your include path.

SEQAN_HAS_ZLIB

possible value 0, 1
default 0

meaning If set to 1 then zlib is available. You have to link against the library (e.g. add -lz to your linker flags) and zlib.h must be in your include path.

Settings Projects Using Seqan

You normally want to have at least two build modes: one for debugging and one for optimized compiling. The following settings have to be applied to your IDE project/Makefiles (below is an example for a Makefile based project).

Debug Builds

Besides enabling debug symbols and disabling optimization, there are the following SeqAn specific settings to be applied.

• Add the path to the directory seqan to your include path.
• Define SEQAN_ENABLE_DEBUG to be 1. Alternatively, you can leave SEQAN_ENABLE_DEBUG undefined and not define NDEBUG.
• Define SEQAN_ENABLE_TESTING to be 0.

This translates into the following GCC flags:
-g -O0 -DSEQAN_ENABLE_TESTING=0 -I${PATH_TO_CORE}/include \
-I${PATH_TO_EXTRAS}/include

### Release/Optimized Builds

Besides disabling debug symbols, enabling optimization and disabling assertions in the standard library, there are the following SeqAn specific settings to be applied.

- Add the path to the directory `seqan` to your include path.
- Define `NDEBUG`. This will make `SEQAN_ENABLE_DEBUG` be defined as 0 if you don’t defined `SEQAN_ENABLE_DEBUG` otherwise.
- Define `SEQAN_ENABLE_TESTING` to be 0.

This translates into the following GCC flags:

```
-O3 -DNDEBUG -DSEQAN_ENABLE_TESTING=0 -I${PATH_TO_CORE}/include \
-I${PATH_TO_EXTRAS}/include
```

### An Example Project Based On Makefiles

We will create a project with good old Makefiles and GCC. The program will not do much but can serve as a minimal example on how to use SeqAn with your own build process. You should be able to adapt this guide to your favourite build system or IDE.

The example project can be found in `misc/makefile_project`. The project layout looks like this:

```
|-- Makefile.rules
|-- Makefile
|-- README
|-- debug
| |-- Makefile
|-- release
| |-- Makefile
|-- src
| |-- main.cpp
```

#### main.cpp

We have one directory `src` for source files. The file `main.cpp` looks as follows:

```cpp
#include <iostream>
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main() {
    std::cout << CharString("Hello SeqAn!") << std::endl;
    return 0;
}
```
It includes SeqAn headers just as you would within the SeqAn CMake framework.

Now, consider the contents of the Makefiles:

**Makefile.rules**

Contains the necessary commands to build the object file for the program main.cpp and then make an executable main from it and clean targets. This file is included from the files release/Makefile and debug/Makefile.

```plaintext
SRC=../src
CXXFLAGS+=-I../../../core/include
CXXFLAGS+=-I../../../extras/include

default: all
all: main

main: main.o
    $(CXX) $(LDFLAGS) -o main main.o

main.o: $(SRC)/main.cpp
    $(CXX) $(CXXFLAGS) -c -o main.o $(SRC)/main.cpp

clean:
    rm -f main.o main

.PHONY: default all clean
```

**Makefile**

Allows to build both debug and release builds by calling make debug, make release or make all from the project directory. Removes all binaries with make clean.

```plaintext
default: all
all: debug release
debug:
    $(MAKE) -C debug
release:
    $(MAKE) -C release
clean:
    $(MAKE) -C debug clean
    $(MAKE) -C release clean

.PHONY: default all debug release clean
```

**debug/Makefile, release/Makefile**

The file debug/Makefile looks as follows.

```plaintext
include ../Makefile.rules

CXXFLAGS+=-g -00 -DSEQAN_ENABLE_TESTING=0 -DSEQAN_ENABLE_DEBUG=1
```
The file `release/Makefile` looks as follows.

```
include ../Makefile.rules
CXXFLAGS+=-O3 -DNDEBUG -DSEQAN_ENABLE_TESTING=0 -DSEQAN_ENABLE_DEBUG=0
```

These Makefiles include the file `Makefile.rules`. They add build type specific arguments to the variables $(CXXFLAGS). For debug builds, debug symbols are enabled, optimization level 0 is chosen, testing is enabled in SeqAn and debugging is disabled. For release builds, debug symbols are not, optimization level 3 is chosen, testing and debugging are both disabled in SeqAn. For good measure, we also disable assertions in the C library with -DNDEBUG.

**Notes**

Note we that added include path to the directory include that contains the directory seqan. By changing the include path, we can install the SeqAn library anywhere. For example, we could create a directory include parallel to src, copy the release version of SeqAn into it and then change the include path of the compiler to point to this directory (value ../include).

**Short Version**

- Add both core/include and extras/include to your include path (-I).
- Windows/MSVC flags: /W2 /wd4996 -D_CRT_SECURE_NO_WARNINGS (optional).
- Defines: NDEBUG to also disable SeqAn assertions in release mode.

**Contents**

- Using the FindSeqAn CMake Module
  - Overview
  - Input / Output of the FindSeqAn Module
    - Input
    - Output
  - Example

**2.4.2 Using the FindSeqAn CMake Module**

**Overview**

CMake is a cross-platform build system generator. That is, you describe the different executables and binaries and their dependencies CMakeLists.txt files. Then, CMake generates build systems from this, for example in the form of Makefiles or Visual Studio projects.

This article will not describe how to use CMake in general but only how to use SeqAn easily from within CMake projects. In CMake projects, one uses modules to find libraries such as SeqAn. SeqAn ships with such a module in util/cmake/FindSeqAn.cmake.
This article describes how to use this module.

**Input / Output of the FindSeqAn Module**

As with all other modules, you have to make the file `FindSeqAn.cmake` available as a CMake module, either by putting it into the same directory as the `CMakeLists.txt` that you are using it from or by adding the path to the file `FindSeqAn.cmake` to the variable `CMAKE_MODULE_PATH`.

Then, you can use it as follows (the argument `REQUIRED` is optional):

```cmake
find_package (SeqAn REQUIRED)
```

**Input**

SeqAn is somewhat special as a library since it has some optional dependencies. Certain features in SeqAn can be enabled or disabled, depending on whether the dependencies could be found.

You can set the dependencies to search for with the variable `SEQAN_FIND_DEPENDENCIES` (which is a list). For example:

```cmake
set (SEQAN_FIND_DEPENDENCIES ZLIB BZip2)
find_package (SeqAn)
```

Note that `FindSeqAn.cmake` itself will not search for its dependencies with the argument `REQUIRED`. Rather, it will set the variables `SEQAN_HAS_*` and add corresponding definitions to `SEQAN_DEFINITIONS` (see below).

Currently, you can specify the following dependencies:

- **ALL** Enable all dependencies.
- **DEFAULT** Enable default dependencies (zlib, OpenMP if available)
- **NONE** Disable all dependencies.
- **ZLIB** zlib compression library
- **BZip2** libbz2 compression library
- **OpenMP** OpenMP language extensions to C/C++
- **CUDA** CUDA language extensions to C/C++

If you want `FindSeqAn.cmake` to expect the SeqAn build system layout then set the variable `SEQAN_USE_SEQAN_BUILD_SYSTEM` to `TRUE`. In this case, it will try to locate the library parts from `core` and `extras`.

**Output**

The call to `find_package(SeqAn)` will set the following variables:

- **SEQAN_FOUND** Indicate whether SeqAn was found.
- **SEQAN_HAS_ZLIB** TRUE "if zlib was found."
- **SEQAN_HAS_BZIP2** TRUE "if libbz2 was found."
- **SEQAN_HAS_OPENMP** TRUE "if OpenMP was found."
- **SEQAN_HAS_CUDA** TRUE "if CUDA was found."
Variables to be passed to include_directories(), target_link_directories(), and add_definitions() in your CMakeLists.txt:

SEQAN_INCLUDE_DIRS A list of include directories.
SEQAN_LIBRARIES A list of libraries to link against.
SEQAN_DEFINITIONS A list of definitions to be passed to the compiler.

Required additions to C++ compiler flags are in the following variable:

SEQAN_CXX_FLAGS C++ Compiler flags to add.

The following variables give the version of the SeqAn library, its major, minor, and the patch version part of the version string.

SEQAN_VERSION_STRING Concatenated version string. ""\${SEQAN_VERSION_MAJOR}\${SEQAN_VERSION_MINOR}\${SEQAN_VERSION_PATCH}".
SEQAN_VERSION_MAJOR Major version.
SEQAN_VERSION_MINOR Minor version.
SEQAN_VERSION_PATCH Patch-level version.

The following flag defines whether this is a trunk version and the version given by the variables above is meant to be used as the previously released version.

SEQAN_VERSION_DEVELOPMENT Whether or not this is a pre-release version.

Example

Below you can find a minimal example CMakeLists.txt file that uses the FindSeqAn.cmake.

cmake_minimum_required (VERSION 2.8.2)
project (core_apps_dfi)

# Dependencies
# Only search for zlib as a dependency for SeqAn.
set (SEQAN_FIND_DEPENDENCIES ZLIB)
find_package (SeqAn REQUIRED)

# Build Setup
# Add include directories.
include_directories ("\${SEQAN_INCLUDE_DIRS}"

# Add definitions set by find_package (SeqAn).
add_definitions ("\${SEQAN_DEFINITIONS}")

# Add CXX flags found by find_package (SeqAn).
set (CMAKE_CXX_FLAGS "$\{CMAKE_CXX_FLAGS}\${SEQAN_CXX_FLAGS}"

# Add executable and link against SeqAn dependencies.
add_executable (app app.cpp)
target_link_libraries (dfi \${SEQAN_LIBRARIES}))
2.4.3 Using the SeqAn Build System

We describe the SeqAn build system from three perspectives:

- The **app user** who just wants to compile a couple of SeqAn applications from the SeqAn SVN repository.
- The **SeqAn release manager** who wants to create SeqAn releases.
- The **SeqAn developer** who wants to write his own applications using the SeqAn build system.

But first, we will give a short overview of the repository and how versioning applications and the whole project works.

Repository Structure and Versioning

The SVN repository follows the classical structure with `/trunk`, `/branches`, `/tags` directories. Below the `trunk`, project is separated into the `core` and `extras` part. Each of these parts contains apps and library modules as well as tests and demos for the library modules.

Note that there is no separation between apps and the library.

When performing a **library release**, the `/trunk` directory is copied to `/tags` with a name like `seqan-release-1.4`. Note that this creates a tag for both the library and the apps. When a bug is found then the tag has to be copied to `/branches` where bugs can be fixed. When the version has been corrected, it can be tagged again.

Independently of this, an **app release** is performed by copying the `/trunk` directory to `/branches` if stabilization is required or directly tagged, e.g. as `/tags/masai-0.6.0` for the app Masai in version 0.6.0.

Note that tags are final and a new tag has to be created if any code is to be changed.
**User Perspective**

The user can checkout /trunk or a copy thereof (e.g. /tags/masai-0.7.0 or /branches/seqan-release-1.4) to his local computer. The user could then proceed as the developer (see below) but there are dedicated modes in the SeqAn build system for easier installation. A user might also want to install the library to an include folder. We will look at both use cases.

**User App Installation**

Let us first consider a user wanting to build RazerS 3 from the trunk as the developer would.

```
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/Release
~ # cd seqan-trunk-build/Release
Release # cmake ../../seqan-trunk
Release # make razers3
```

This will check out the SeqAn trunk and create the binary `razers3` in the directory `~/seqan-trunk-build/Release/bin`.

However, it will be more convenient for the user to build the app and then install it, for example to `~/local/razers3`:

```
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/razers3
~ # cd seqan-trunk-build/razers3
razers3 # cmake ../../seqan-trunk -DCMAKE_INSTALL_PREFIX=~/local/razers3 \
    -DSEQAN_BUILD_SYSTEM=APP:razers3
razers3 # make install
```

The user could install a released version of the RazerS 3 program by using the URL http://svn.seqan.de/seqan/tags/razers3-3.2.0, for example.

After executing this, the user will find the following structure in `~/local/razers3`, including the example files and documentation.

```
razers3 # tree ~/local/razers3
/home/${USER}/local/razers3/
|-- bin
| `-- razers3
|-- example
| `-- genome.fa
| `-- reads2.fa
| `-- reads.fa
|-- LICENSE
|-- README
```

**User Library Installation**

The user could also want to install the library headers only. The checkout step is the same as above, but he has to create a new build directory and execute CMake with different parameters. The library will be installed to `~/local/seqan`.

```
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/library_only
~ # cd seqan-trunk-build/library_only
library_only # cmake ../../seqan-trunk -DCMAKE_INSTALL_PREFIX=~/local/seqan \
    -DSEQAN_BUILD_SYSTEM=LIBRARY_ONLY
```

---

2.4. Build Manual 551
The user can now find the SeqAn library in `~/local/seqan/include`:

```
library_only # tree ~/local/seqan/
/home/${USER}/local/seqan/
|-- include
|  |-- seqan
|     |-- align
|     |  |-- align_base.h
|     |  |-- align_cols.h
|     |  |-- align_config.h
|     |  |-- align_iterator_base.h
|     |  |-- alignment_algorithm_interface.h
|     |  |-- alignment_algorithm_tags.h
|     |  |-- alignment_operations.h
|     |  |-- align_metafunctions.h
|     |  |-- align_traceback.h
|     |  |-- gap_anchor.h
|     ...
|  |-- system.h
|  |-- version.h
|-- share
  |-- doc
     |-- seqan
        |-- LICENSE
        |-- README
```

**SeqAn Release Manager Perspective**

The SeqAn release manager wants to create release packages of (1) individual apps from the SeqAn repository, (2) create a SeqAn library release that includes the library and documentation, and (3) create a SeqAn apps release that contains the built apps. The manager wants to build the binary packages for different platforms, e.g. 32 bit and 64 bit Linux and Windows, Mac Os X, etc.

We will give examples for Unixoid operating systems.

Note that the packaging described below can be automatized. App and project releases can simply be tagged in the Subversion repository. A script that runs nightly can then pick up new tags from the Subversion repository and create binary packages for them. This can also automatize nightly builds on different platforms without much work for the release manager.

**Packaging Individual Apps**

The release manager would check out an app in a specific revision, e.g. through a tag or the trunk version:

```
~ # svn co http://svn.seqan.de/seqan/tags/masai-0.6.1 masai-0.6.1
~ # mkdir masai-0.6.1-build
~ # cd masai-0.6.1-build
masai-0.6.1-build # cmake ../masai-0.6.1 -DSEQAN_BUILD_SYSTEM=APP:masai \
         -DSEQAN_APP_VERSION=0.6.1
masai-0.6.1-build # make package
```

On Unix, this will create a Tarball (`.tar.bz2`) and a ZIP file with the binaries, documentation, and example files:
The packages have the following structure:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Permission</th>
<th>Size (bytes)</th>
<th>Date</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>masai-0.6.1-build # tar tfj masai-0.6.1-Linux-x86_64.tar.bz2</td>
<td>-rw-rw-r--</td>
<td>1094198</td>
<td>Nov 20</td>
<td>13:36</td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/bin/masai_mapper</td>
<td>-rw-rw-r--</td>
<td>1243428</td>
<td>Nov 20</td>
<td>13:36</td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/bin/masai_indexer</td>
<td>-rw-rw-r--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/bin/masai_output_se</td>
<td>-rw-rw-r--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/bin/masai_output_pe</td>
<td>-rw-rw-r--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/README</td>
<td>-rw-rw-r--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>masai-0.6.1-Linux-x86_64/LICENSE</td>
<td>-rw-rw-r--</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packaging Library Releases

Packaging the library and documentation is quite simple. Note that we have to build the documentation using `make docs` before calling `make package` because of a bug in CMake that prevents us from doing it automatically.

The version is automatically detected from the constants in the `seqan/version.h` header. There also is a marker variable that marks whether the checked out repository version has a version number or whether it is a pre-release of the next version.

```bash
~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/release_library
~ # cd seqan-trunk-build/release_library
release_library # cmake ../../seqan-trunk -DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_LIBRARY
release_library # make docs
release_library # make package
```

On Linux, this will build three archives:

```bash
release_library # ls -l seqan-library-pre1.4.0-Linux.*
-rw-rw-r-- 1 USER GROUP 3367876 Nov 20 13:57 seqan-library-pre1.4.0-Linux.deb
-rw-rw-r-- 1 USER GROUP 2357465 Nov 20 13:57 seqan-library-pre1.4.0-Linux.tar.bz2
-rw-rw-r-- 1 USER GROUP 5953550 Nov 20 13:57 seqan-library-pre1.4.0-Linux.zip
```

Let us look at the contents of one (they all contain the same files):

```bash
release_library # dpkg --contents seqan-library-pre1.4.0-Linux.deb
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/share/
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/share/seqan/
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/share/seqan/docs/
drwxr-xr-x root/root 0 2012-11-20 13:57 ./usr/share/seqan/docs/html/
...
-rw-r--r-- root/root 2940 2012-11-06 13:28 ./usr/share/doc/seqan/README
-rw-r--r-- root/root 1517 2012-11-06 13:28 ./usr/share/doc/seqan/LICENSE
-drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/include/
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/include/seqan/
drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/include/seqan/statistics/
-rw-r--r-- root/root 24044 2012-11-06 13:28 ./usr/include/seqan/statistics/statistics_markov_model.h
-rw-r--r-- root/root 15533 2012-11-06 13:28 ./usr/include/seqan/statistics/statistics_base.h
-drwxrwxr-x root/root 0 2012-11-20 13:57 ./usr/include/seqan/random/
-rw-r--r-- root/root 15590 2012-11-06 13:28 ./usr/include/seqan/random/ext_MersenneTwister.h
```
Packaging All Apps

It is simple to create a SeqAn Apps release:

~ # svn co http://svn.seqan.de/seqan/trunk seqan-trunk
~ # mkdir -p seqan-trunk-build/release_apps
~ # cd release_apps
release_apps # cmake ../../seqan-trunk -DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_APPS
release_apps # make package
release_apps # ls -l seqan-apps-pre1.4.0-Linux*
-rw-rw-r-- 1 USER GROUP 532 Nov 20 14:22 seqan-apps-pre1.4.0-Linux.deb
-rw-rw-r-- 1 USER GROUP 42 Nov 20 14:22 seqan-apps-pre1.4.0-Linux.tar.bz2
-rw-rw-r-- 1 USER GROUP 22 Nov 20 14:22 seqan-apps-pre1.4.0-Linux.zip

The contents of the archives is as follows:

release_library # dpkg --contents seqan-apps-pre1.4.0-Linux.deb
dpkg --contents seqan-apps-pre1.4.0-Linux.deb
    drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/
drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/bin/
-rw-rxr-x root/root 2253741 2012-11-20 14:27 ./usr/bin/masai_mapper
-rw-rxr-x root/root 191351 2012-11-20 14:24 ./usr/bin/tree_recon
-rw-rxr-x root/root 349878 2012-11-20 14:26 ./usr/bin/param_chooser
... 
    drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/
drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/doc/
drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/doc/tree_recon/
drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/doc/tree_recon/example/
-rw-rrxr-x root/root 475 2012-11-20 13:32 ./usr/share/doc/tree_recon/example/example.dist
-rw-r-r-- root/root 20 2012-11-20 13:32 ./usr/share/doc/tree_recon/README
-rw-r-r-- root/root 843 2012-11-20 13:32 ./usr/share/doc/tree_recon/LICENSE
... 
    drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/doc/razers3/
drwxrwxr-x root/root 0 2012-11-20 14:30 ./usr/share/doc/razers3/example/
-rw-r-r-- root/root 985 2012-11-06 13:28 ./usr/share/doc/razers3/example/genome.fa

Nightly Builds

It is also possible to create nightly builds of the library, all apps, or individual apps. Simply define the CMake variable SEQAN_NIGHTLY_RELEASE to TRUE on the command line. In the following examples, we skip the checkout step and simply show the CMake and build steps:
One App

masai-build # cmake ../masai-0.6.1 -DSEQAN_BUILD_SYSTEM=APP:masai \
-DSEQAN_NIGHTLY_RELEASE=TRUE
masai-build # make package
masai-build # ls -l masai-20121120-Linux-x86_64.*
  -rw-rw-r-- 1 USER GROUP 1091927 Nov 20 14:11 masai-20121120-Linux-x86_64.tar.bz2
  -rw-rw-r-- 1 USER GROUP 1241259 Nov 20 14:11 masai-20121120-Linux-x86_64.zip
masai-build # tar tfj masai-20121120-Linux-x86_64.tar.bz2
masai-20121120-Linux-x86_64/bin/masai_mapper
masai-20121120-Linux-x86_64/bin/masai_indexer
masai-20121120-Linux-x86_64/bin/masai_output_se
masai-20121120-Linux-x86_64/bin/masai_output_pe
masai-20121120-Linux-x86_64/README
masai-20121120-Linux-x86_64/LICENSE

All Apps

release_apps # cmake ../../seqan-trunk -DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_APPS \
-DSEQAN_NIGHTLY_RELEASE=TRUE
release_apps # make package
release_apps # ls -l seqan-apps-20121120-**
  -rw-rw-r-- 1 USER GROUP 10232442 Nov 20 14:37 seqan-apps-20121120-Linux.deb
  -rw-rw-r-- 1 USER GROUP 8847407 Nov 20 14:37 seqan-apps-20121120-Linux.tar.bz2
  -rw-rw-r-- 1 USER GROUP 10266596 Nov 20 14:37 seqan-apps-20121120-Linux.zip

Library Only

release_library # cmake ../../seqan-trunk -DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_LIBRARY \
-DSEQAN_NIGHTLY_RELEASE=TRUE
release_library # make docs
release_library # make package
release_library # ls -l seqan-library-20121120-**
  -rw-rw-r-- 1 USER GROUP 3368034 Nov 20 14:07 seqan-library-20121120-Linux.deb
  -rw-rw-r-- 1 USER GROUP 2356769 Nov 20 14:07 seqan-library-20121120-Linux.tar.bz2
  -rw-rw-r-- 1 USER GROUP 5955755 Nov 20 14:06 seqan-library-20121120-Linux.zip

SeqAn Developer Perspective

SeqAn developers want to develop their own applications using SeqAn. When they want to use the SeqAn build system, they can follow these instructions to (1) setup their sandbox, (2) setup their apps in their sandbox and later core/extras, and (3) create releases of the applications.

Creating Sandboxes

Creating sandboxes is easy with the util/skel.py script (also see Using The Code Generator).

seqan # ./util/bin/skel.py repository sandbox/my_sandbox

We will not go into detail on the structure of generated CMakeLists.txt files.
Creating Apps

Simply use the `util/skel.py` script (also see *Using The Code Generator*).

```
seqan-trunk # ./util/bin/skel.py app my_app sandbox/my_sandbox
```

This will generate a `CMakeLists.txt` file in `sandbox/my_sandbox/apps/my_app`. Since you will have to adjust the file to your project, let us have a look at the file in detail. You can look up details in the *CMake documentation* in case that some *CMake* functions are not clear to you.

The file starts out with a header describing where the file lives and what it is for. This is useful when having many `CMakeLists.txt` files open and you want to quickly identify in the file in the current window.

```
# ===========================================================================
# SeqAn - The Library for Sequence Analysis
# ===========================================================================
# File: /sandbox/my_sandbox/apps/my_app/CMakeLists.txt
#
# # CMakeLists.txt file for my_app.
# ===========================================================================

cmake_minimum_required (VERSION 2.8.2)
project (sandbox_my_sandbox_apps_my_app)
message (STATUS "Configuring sandbox/my_sandbox/apps/my_app")
```

Then comes the section that searches for the app’s dependencies. By default, the app only depends on the package *SeqAn*. By setting the variable `SEQAN_FIND_DEPENDENCIES`, we can configure which dependencies the call to `find_package (SeqAn REQUIRED)` will try to find. See the *Using the FindSeqAn CMake Module* for more details.

```
# Dependencies
# ===========================================================================
# Search SeqAn and select dependencies.
set (SEQAN_FIND_DEPENDENCIES NONE)
find_package (SeqAn REQUIRED)
```

The call to `find_package (SeqAn REQUIRED)` will then set the following variables that we will then use below to add the correct parameters to the compiler and linker.

- `SEQAN_INCLUDE_DIRS`: Required include directories for the headers. Pass to `include_directories()`
- `SEQAN_DEFINITIONS`: Additional precompiler macros to pass to the compiler. Pass to `add_definitions()`
- `SEQAN_CXX_FLAGS`: Additional C++ compiler flags. Extend `CMAKE_CXX_FLAGS` by this list.
- `SEQAN_LIBRARIES`: The libraries to link against. Pass to `target_link_libraries()` for each target.

We then need one `add_executable()` call for each program executable that we want to build. We also need to link the libraries into the program.

```
# Build Setup
# ===========================================================================
# Add CXX flags found by find_package(SeqAn).
set (CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} ${SEQAN_CXX_FLAGS})
```
# Add include directories.
include_directories (${SEQAN_INCLUDE_DIRS})

# Add definitions set by find_package(SeqAn).
add_definitions (${SEQAN_DEFINITIONS})

# Update the list of file names below if you add source files to your application.
add_executable (dfi dfi.cpp)

# Add dependencies found by find_package(SeqAn).
target_link_libraries (dfi ${SEQAN_LIBRARIES})

We then configure the app for installation. Note that this is a distinct step than configuring CPack for packaging. The following controls which files to copy when calling make install. CPack will use the result of make install for creating its packages.

We first call seqan_setup_install_vars() (to set the variable SEQAN_PREFIX_SHARE_DOC. This is required for installing documentation and example files to share/${PROGRAM_NAME} when building multiple apps and directly to the current directory when building only one app.

The macro seqan_setup_install_vars is specific to the SeqAn build system.

The util/skel.py script will create files LICENSE and README for you. If you want to include additional files then you should use one of the given install() calls. Install documentation to ${SEQAN_PREFIX_SHARE_DOC} and examples to ${SEQAN_PREFIX_SHARE_DOC}/example.

# Installation
# --------------------------------------------------------------------------

# Set variables for installing, depending on the selected build type.
if (NOT SEQAN_PREFIX_SHARE_DOC)
  seqan_setup_install_vars (dfi)
endif (NOT SEQAN_PREFIX_SHARE_DOC)

# Install dfi in ${PREFIX}/bin directory
install (TARGETS dfi
  DESTINATION bin)

# Install non-binary files for the package to "." for app builds and
# ${PREFIX}/share/doc/dfi for SeqAn release builds.
install (FILES LICENSE
  README
  DESTINATION ${SEQAN_PREFIX_SHARE_DOC})

#install (FILES example/example.txt
#  DESTINATION ${SEQAN_PREFIX_SHARE_DOC}/example)

Then, we can use the macro seqan_add_app_test() from the SeqAn build system to register app tests. If you want to add an app test for your program then simply uncomment the seqan_add_app_test() call and follow the instructions in Writing App Tests to write such an app tests.

# App Test
# --------------------------------------------------------------------------

#seqan_add_app_test(dfi)

Finally, we configure the application packaging system for building individual apps.
Building Apps

Simply use CMake to generate project files for the whole SeqAn repository and your sandbox. Let us say that we want to build the app `my_app` in your sandbox:

```
~ # mkdir -p seqan-trunk-build/Release
~ # cd seqan-trunk-build/Release
Release # cmake ../../seqan-trunk
Release # make my_app
```

Note that the default build type is the Release mode. The binaries will be built with optimization and without debug symbols. To build apps with debug symbols and without optimization with Makefiles, use the CMake parameter `-DCMAKE_BUILD_TYPE=Debug`. When using IDE files such as for Xcode, you can select the optimization state from within the IDE.

```
Release # cd ../..
~ # mkdir -p seqan-trunk-build/Debug
~ # cd seqan-trunk-build/Debug
Debug # cmake ../../seqan-trunk
Debug # make my_app
```

Windows Notes

The descriptions above apply to Linux/Mac systems. On Windows, things are only slightly different:

- There are packages available that provide the `svn.exe` command line client or users might use the GUI client TortoiseSVN.
- The `mkdir` command differs slightly (the `-p` parameter can be omitted).
- Instead of using the backslash \ two split one command over two lines in the Command Prompt, we have to use the Windows equivalent ^.
- The `cmake` command line program is also available for Windows.

The main difference is that when building with the Visual Studio tools, one does not use `make` for building applications. When developing, users can simply open the generated Visual Studio *.sln solution files and then use Visual Studio for building the applications. When packaging, users can use the `msbuild` command as described below.

As an example, we adapt the description of creating an application release for Masai on Windows. The next steps are typed into the Command Prompt (Start > All Programs > Accessories > Command).
SeqAn Manual, Release 1.4.2

C:\> svn co http://svn.seqan.de/seqan/tags/masai-0.6.1 masai-0.6.1
C:\> mkdir masai-0.6.1-build
C:\> cd masai-0.6.1-build
C:\masai-0.6.1-build> cmake ..\masai-0.6.1 -DSEQAN_BUILD_SYSTEM=APP:masai ^
-DSEQAN_APP_VERSION=0.6.1

So far, the only difference to the Unix descriptions is the using backslashes instead of forward slashes for paths.
You can then open the generated seqan.sln file in C:\masai-0.6.1-build with Visual studio and build the
packages from there.
Alternatively, msbuild can be used. This program is only available when using the Visual Studio Command Prompt.
For Visual Studio 2010, you can start it through the start menu as follows: Start > Programs > Microsoft
For other Visual Studio versions, the path is similar. If you want 64 bit builds then you have to start Visual
Studio x86 Win64 Command Prompt (2010).
C:\> cd masai-0.6.1-build
C:\masai-0.6.1-build> msbuild /p:Configuration=Release PACKAGE.vcxproj

This will create a ZIP file with the app build of Masai.
Note that you could also input the first part of commands from this example into the Visual Studio Command Prompt.
Using CUDA
To use cuda, simply insert the following section into your CMakeLists.txt behind the Dependencies section.
This consists of the following step:
1. Find CUDA package
2. If CUDA could not be found then stop.
3. Disabling propagating host flags to the cuda compiler, some visual studio configuration.
4. Removing the -pedantic flag from the compiler flags.
5. Register .cu as the extension for C++ files, required for linking.
6. Register the include directory for the cut (CUDA Toolkit) library.
# ---------------------------------------------------------------------------# CUDA Setup
# ---------------------------------------------------------------------------# The CUDA setup is a bit verbose so it gets its own section.
# Search for CUDA.
find_package (CUDA)
# Stop here if we cannot find CUDA.
if (NOT CUDA_FOUND)
message (STATUS " CUDA not found, not building cuda_ex.")
return ()
endif (NOT CUDA_FOUND)
# Set CUDA options.
set (CUDA_PROPAGATE_HOST_FLAGS OFF)
set (CUDA_ATTACH_VS_BUILD_RULE_TO_CUDA_FILE OFF)
# Remove -pedantic flag.

2.4. Build Manual

559


string (REGEX REPLACE "\\-pedantic" "
    CUDA_CXX_FLAGS ${CUDA_NVCC_FLAGS} ${CMAKE_CXX_FLAGS})

# Enable .cu as a CXX source file extension for linking.
list (APPEND CMAKE_CXX_SOURCE_FILE_EXTENSIONS "cu")
# Add CUT include directories for CUDA.
cuda_include_directories(${CUDA_CUT_INCLUDE_DIR})

## 2.5 SeqAn Style Guides
# Contents

- SeqAn C++ Code Style
  - C++ Features
    * Reference Arguments
    * Use C-Style Logical Operators
    * Default Arguments
    * Exceptions
    * Virtual Member Functions
    * `static_cast<>`
    * `const_cast<>`
    * `reinterpret_cast<>`
    * pre/post increment/decrement
  - Code Quality
    * Const-Correctness
    * Compiler Warnings
    * Style Conformance
  - Semantics
    * Parameter Ordering
  - Scoping, Helper Code
    * Global Variables
    * Tags In Function Arguments
  - Structs and Classes
    * Visibility Specifiers
    * Tag Definitions
    * In-Place Member Functions
  - Formatting
    * Constructor Initialization Lists
    * Line Length
    * Non-ASCII Characters
    * Spaces VS Tabs
    * Indentation
    * Trailing Whitespace
    * Inline Comments
    * Brace Positions
    * Conditionals
    * Loops and Switch Statements
    * Expressions
    * Type Expressions
    * Function Return Types
    * Inline Functions
    * Function Argument Lists
    * Template Argument Lists
    * Function Calls
  - Naming Rules
    * Macros
    * Variable Naming
    * Constant / Enum Value Naming
    * Struct / Enum / Class Naming
    * Metafunction Naming
    * Function Naming
    * Names In Documentation
  - Comments
    * File Comments
    * Class, Function, Metafunction, Enum, Macro DDDoc Comments
    * Implementation Comments
    * TODO Comments
  - Source Tree Structure

## 2.5. SeqAn Style Guides

- File Name Rules
- Include Order
- CPP File Structure
- Application Header Structure
- Library Header Structure

---

2.5. SeqAn Style Guides
2.5.1 SeqAn C++ Code Style

The aim of this style guide is to enforce a certain level of canonicality on all SeqAn code. Besides good comments, having a common style guide is the key to being able to understand and change code written by others easily.

(The style guide partially follows the Google C++ Code Style Guide.)

C++ Features

Reference Arguments

We prefer reference arguments over pointer arguments. Use `const` where possible.

Use C-Style Logical Operators

Use `&&`, `||`, and `!` instead of `and`, `or`, and `not`.

While available from C++98, MSVC does not support them out of the box, a special header `<iso646.h>` has to be included. Also, they are unfamiliar to most C++ programmers and nothing in SeqAn is using them.

Default Arguments

Default arguments to global functions are problematic with generated forwards. They can be replaced with function overloading, so do not use them.

You can replace default arguments with function overloading as follows. Do not do this.

```cpp
inline double f(int x, double y = 1.0) {
    // ...
}
```

Do this instead.

```cpp
inline double f(int x, double y) {
    // ...
}
inline double f(int x) {
    return f(x, 1.0);
}
```

Exceptions

Currently, the SeqAn code does not use any exceptions and is not exception safe. Do not use any exceptions yourself, instead use return codes.

The following is an example where two chars are read using `<cstdio>` I/O. We use a return code of 0 to indicate no errors.
int readSome(char & c, FILE * fp)
{
    int res = fgetc(fp);
    if (res < 0)
        return res;
    res = fgetc(fp);
    if (res < 0)
        return res;
    c = res;
    return 0;
}

Virtual Member Functions

Do not use virtual member functions. Since we mostly use template subclassing instead of C++ built-in subclassing, there rarely is the need for member functions. In the case where there are member functions, they should not be virtual since this is slow when used in tight loops.

static_cast<>

Prefer static_cast<> over C-style casts.

const_cast<>

Use const-casts only to make an object const, do not remove consts. Rather, use the mutable keyword on selected members. const_cast<> is allowed for interfacing with external (C) APIs where the const keyword is missing but which do not modify the variable.

The following is an example where const_cast<> is OK:

    template <typename T>
    bool isXyz(T const & x)
    {
        return x._member == 0;
    }

    template <typename T>
    bool isXyz(T & x)
    {
        return const_cast<T const &>(x)._member == 0;
    }

reinterpret_cast<>

Only use reinterpret_cast<> when you absolutely have to and you know what you are doing! Sometimes, it is useful for very low-level code but mostly it indicates a design flaw.

pre/post increment/decrement

Prefer the “pre” variants for decrement and increment, especially in loops. Their advantage is that no copy of an object has to be made.
Good:

```cpp
typedef Iterator<TContainer>::Type TIterator;
for (TIterator it = begin(container); atEnd(it); ++it)
{
    // do work
}
```

Bad:

```cpp
typedef Iterator<TContainer>::Type TIterator;
for (TIterator it = begin(container); atEnd(it); it++)
{
    // do work
}
```

**Code Quality**

**Const-Correctness**

Write const correct code. Read the C++ FAQ const correctness article for more information. Besides other things, this allows to use temporary objects without copying in functions that do not need to change their arguments.

**Compiler Warnings**

All code in the “core” and “extras” repository must compile without any warnings using the flags generated by the CMake system.

Currently, the GCC flags are:

```
:: -W -Wall -Wstrict-aliasing -pedantic -Wno-long-long -Wno-variadic-macros
```

**Style Conformance**

Follow this code style whenever possible. However, prefer consistency over conformance.

If you are editing code that is non-conforming consider whether you could/should adapt the whole file to the new style. If this is not feasible, prefer consistency over conformance.

**Semantics**

**Parameter Ordering**

The general parameter order should be (1) output, (2) non-const input (e.g. file handles), (3) input, (4) tags. Within these groups, the order should be from mandatory to optional.

In SeqAn, we read functions `f(out1, out2, out3, ..., in1, in2, in3, ...) as (out1, out2, out3, ...) <- f(in1, in2, in3, ...)`.  
E.g. `assign()`:

```cpp
template <typename T>
void f(T & out, T const & in)
{
```
Scoping, Helper Code

Global Variables

Do not use global variables. They introduce hard-to-find bugs and require the introduction of a link-time library.

Tags In Function Arguments

Tags in function arguments should always be const.

```cpp
// somewhere in your code:
struct Move_;
typedef Tag<Move_> Move;

// then, later:
void appendValue(TContainer, Move const &)
{
   // ...}
```

Structs and Classes

Visibility Specifiers

Visibility specifiers should go on the same indentation level as the class keyword.

Example:

```cpp
class MyStruct
{
 public:
 protected:
 private:
};
```

Tag Definitions

Tags that are possibly also used in other modules must not have additional parameters and be defined using the Tag<> template. Tags that have parameters must only be used within the module they are defined in and have non-generic names.

Tags defined with the Tag<> template and a typedef can be defined multiply. These definitions must have the following pattern:

```cpp
structTagName_;
typedef Tag<TagName_> TagName;
```
This way, there can be multiple definitions of the same tag since the struct `TagName_` is only declared but not defined and there can be duplicate typedefs.

For tags (also those used for specialization) that have template parameters, the case is different. Here, we cannot wrap them inside the `Tag<>` template with a typedef since it still depends on parameters. Also we want to be able to instantiate tags so we can pass them as function arguments. Thus, we have to add a struct body and thus define the struct. There cannot be multiple identical definitions in C++. Thus, each tag with parameters must have a unique name throughout SeqAn. Possibly too generic names should be avoided. E.g. Chained should be reserved as the name for a global tag but `ChainedFile<>` can be used as a specialization tag in a file-related module.

Note that this restriction does not apply for internally used tags (e.g. those that have an underscore postfix) since these can be renamed without breaking the public API.

### In-Place Member Functions

Whenever possible, functions should be declared and defined outside the class. The constructor, destructor and few operators have to be defined inside the class, however.

The following has to be defined and declared within the class (also see Wikipedia):

- constructors
- destructors
- function call operator `operator()`
- type cast operator `operator T()`
- array subscript operator `operator[]()`
- dereference-and-access-member operator `operator->()`
- assignment operator `operator=()`

### Formatting

#### Constructor Initialization Lists

If the whole function prototype fits in one line, keep it in one line. Otherwise, wrap line after column and put each argument on its own line indented by one level. Align the initialization list.

Example:

```cpp
class Class
{
    MyClass() :
        member1(0),
        member2(1),
        member3(3)
    {}
};
```

#### Line Length

The maximum line length is 120. Use a line length of 80 for header comments and the code section separators.
Non-ASCII Characters

All files should be UTF-8, non-ASCII characters should not occur in them nevertheless.
In comments, use ss instead of ß and ae instead of ä etc.
In strings, use UTF-8 coding. For example, "\xEF\xBB\xBF" is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Spaces VS Tabs

Do not use tabs, use spaces. Use "\t" in strings instead of plain tabs.
After some discussion, we settled on this. All programmer's editors can be configured to use spaces instead of tabs.
We use a four spaces to a tab.
There can be problems when indenting in for loops with tabs, for example. Consider the following (-->| is a tab, _ is a space):

```cpp
for (int i = 0, j = 0, k = 0, ...;
     _cond1 && cond2 &&; ++i)
{
    // ...
}
```

Here, indentation can happen up to match the previous line. Mixing tabs and spaces works, too. However, since tabs are not shown in the editor, people might indent a file with mixed tabs and spaces with spaces if they are free to mix tabs and spaces.

```cpp
for (int i = 0, j = 0, k = 0, ...;
    -->|_cond1 && cond2 &&; ++i)
{
    // ...
}
```

Indentation

We use an indentation of four spaces per level.
Note that "namespaces do not cause an increase in indentation level."

```cpp
namespace seqan {

class SomeClass {
    
};

} // namespace seqan
```

Trailing Whitespace

Trailing whitespace is forbidden.
Trailing whitespace is not visible, leading whitespace for indentation is perceptible through the text following it. Anything that cannot be seen can lead to “trash changes” in the SVN when somebody accidently removes it.
Inline Comments

Use inline comments to document variables.
Possibly align inline comments.

```cpp
short x;    // a short is enough!
int myVar;  // this is my variable, do not touch it
```

Brace Positions

Always put brace positions on the next line.

```cpp
class MyClass
{
public:
    int x;

    MyClass() : x(10)
    {
    }
};

void foo(char c)
{
    switch (c)
    {
        case 'X':
            break;
        // ...
    }
}
```

Conditionals

Use no spaces inside the parantheses, the else keyword belongs on a new line, use block braces consistently.
Conditional statements should look like this:

```cpp
if (a == b)
{
    return 0;
}
else if (c == d)
{
    int x = a + b + d;
    return x;
}

if (a == b)
return 0;
else if (c == d)
    return a + b + d;
```

Do not leave out the spaces before and after the parantheses, do not put leading or trailing space in the paranthesis. The following is wrong:
if (foo){
    return 0;
}
if (foo)
    return 0;
if (foo )
    return 0;

Make sure to add braces to all blocks if any block has one. The following is wrong:
if (a == b)
    return 0;
else if (c == d)
{
    int x = a + b + d;
    return x;
}

Loops and Switch Statements

Switch statements may use braces for blocks. Empty loop bodies should use {} or continue. Format your switch statements as follows. The usage of blocks is optional. Blocks can be useful for declaring variables inside the switch statement.

switch (var)
{
    case 0:
        return 1;
    case 1:
        return 0;
    default:
        SEQAN_FAIL("Invalid value!");
}

switch (var2)
{
    case 0:
        return 1;
    case 1:
    {
        int x = 0;
        for (int i = 0; i < var3; ++i)
            x += i;
        return x;
    }
    default:
        SEQAN_FAIL("Invalid value!");
}

Empty loop bodies should use {} or continue, but not a single semicolon.

while (condition)
{
    // Repeat test until it returns false.
}

for (int i = 0; i < kSomeNumber; ++i)
while (condition) continue;  // Good - continue indicates no logic.

Expressions

Binary expressions are surrounded by one space. Unary expressions are preceded by one space.

Example:

```c
if (a == b || c == d || e == f || !x)
{
    // ...
}
bool y = !x;
unsigned i = ~j;
```

Type Expressions

No spaces around period or arrow. Add spaces before and after pointer and references. const comes after the type.

The following are good examples:

```c
int x = 0;
int * ptr = x;                        // OK, spaces are good.
int const & ref = x;                 // OK, const after int
int main(int argc, char ** argv);    // OK, group pointers.
```

Bad Examples:

```c
int x = 0;
int* ptr = x;                        // bad spaces
int *ptr = x;                        // bad spaces
const int & ref = x;                 // wrong placement of const
int x = ptr -> z;                    // bad spaces
int x = obj. z;                      // bad spaces
```

Function Return Types

If a function definition is short, everything is on the same line. Otherwise, split.

Good example:

```c
int foo();
```

```c
template <typename TString>
typename Value<TString>::Type anotherFunction(TString const & foo, TString const & bar, /*...*/)
{
    // ...
}
```
Inline Functions

If a function definition is short, everything is on the same line. Otherwise put inline and return type in the same line.

Good example:

```cpp
inline int foo();
```

```cpp
template <typename TString>
inline typename Value<TString>::Type anotherFunction(TString const & foo, TString const & bar, /*...*/) {
    // ...
}
```

Function Argument Lists

If it fits in one line, keep in one line. Otherwise, wrap at the paranthesis, put each argument on its own line. For very long function names and parameter lines, break after opening bracket.

Good example:

```cpp
template <typename TA, typename TB>
inline void foo(TA & a, TB & b);
```

```cpp
template </*...*/>
inline void foo2(TA & a, TB & b, ... TY & y, TZ & z);
```

```cpp
template </*...*/>
inline void _functionThisIsAVeryVeryLongFunctionNameSinceItsAHelper(TThisTypeWasMadeToForceYouToWrapInTheLongNameMode & a, TB & b, TC & c, TB & d, ...);
```

Template Argument Lists

Follow conventions of function parameter lists, no blank after opening <.

As for function parameters, try to fit everything on one line if possible, otherwise, break the template parameters over multiple lines and put the commas directly after the type names.

```cpp
template <typename T1, typename T1>
void foo() {}
```

```cpp
template <typename T1, typename T2, ...
    typename T10, typename T11>
void bar() {}
```

Multiple closing > go to the same line and are only separated by spaces if two closing angular brackets come after each other.
typedef Iterator<Value<TValue>::Type, Standard> ::Type
typedef String<char, Alloc<> > TMyString
// -------------------------^
int x;
int myVar;
int saValue(/*...*/);
int getSAValue(/*...*/);

struct FooBar
{
    int _x;
};

Constant / Enum Value Naming

Constant and enum values are named like macros: all-upper case, separated by dashes.

Example:

enum MyEnum
{
    MY_ENUM_VALUE1 = 1,
    MY_ENUM_VALUE2 = 20
};

int const MY_VAR = 10;

Struct / Enum / Class Naming

Types are written in camel case, starting with an upper case character.
Internal library types have an underscore suffix.

Example:

struct InternalType_
{
};

struct SAValue
{
};

struct LcpTable
{
};

Metafunction Naming

Metafunctions are named like structs, defined values are named VALUE, types Type.
Metafunctions should not export any other types or values publically, e.g. they should have an underscore suffix.

Example:

template <typename T>
struct MyMetaFunction
{
    typedef typename RemoveConst<T>::Type TNoConst_;
    typedef TNonConst_ Type;
};
template <typename T>
struct MyMetaFunction2
{
    typedef True Type;
    static bool const VALUE = false;
};

Function Naming

The same naming rule as for variables applies.

Example:

void fooBar();

template <typename T>
int saValue(T & x);

template <typename T>
void lcpTable(T & x);

Names In Documentation

In the documentation, classes have the same name as in the source code, e.g. the class StringSet is documented as “class StringSet.” Specializations are named “$SPEC $CLASS”, e.g. “Concat StringSet”, “Horspool Finder.”

Comments

File Comments

Each file should begin with a file header.

The file header has the format. The skel.py tool automatically generates files with appropriate headers.

// ==========================================================================
// $PROJECT NAME
// ==========================================================================
// Copyright (C) 2010 $AUTHOR, $ORGANIZATION
//
// $LICENSE
//
// $FILE_DESCRIPTION
//

Class, Function, Metafunction, Enum, Macro DDDoc Comments

Each public class, function, metafunction, enum, and macro should be documented using dox API docs. Internal code should be documented, too.

Example:
/*! 
 * @class IntervalAndCargo
 * @headerfile <seqan/refinement.h>
 * @brief A simple record type that stores an interval and a cargo value.
 *
 * @signature template <typename TValue, typename TCargo>
 * struct IntervalAndCargo;
 *
 * @tparam TValue The value type of the record, defaults to int.
 * @tparam TCargo The cargo type of the record, defaults to int.
 */

```cpp
template <typename TValue = int, typename TCargo = int>
class IntervalAndCargo
{
    // ...
};

// This functions helps the XYZ class to fulfill the ABC functionality.
// It corresponds to function FUNC() in the paper describing the original
// algorithm. The variables in this function correspond to the names in the
// paper and thus the code style is broken locally.

void _helperFunction( /*...*/ )
{
}
```

**Implementation Comments**

All functions etc. should be well-documented. In most cases, it is more important how something is done instead of
of what is done.

**TODO Comments**

TODO comments have the format // TODO($USERNAME) : $TODO_COMMENT. The username is the username
of the one writing the item, not the one to fix it. Use GitHub issues for this.

**Source Tree Structure**

**File Name Rules**

File and directories are named all-lower case, words are separated by underscores.
Exceptions are INFO, COPYING, README, ... files.

Examples:

- string_base.h
- string_packed.h
- suffix_array.h
- lcp_table.h
File Structure

Header `#define` guard

The header `#define` include guards are constructed from full paths to the repository root.

Example:

<table>
<thead>
<tr>
<th>filename</th>
<th>preprocessor symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>seqan/core/include/seqan/basic/iterator_base.h</td>
<td>SEQAN_CORE_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_</td>
</tr>
</tbody>
</table>

```cpp
#ifndef SEQAN_CORE_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
define SEQAN_CORE_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
#endif // #ifndef SEQAN_CORE_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
```

Include Order

The include order should be (1) standard library requirements, (2) external requirements, (3) required SeqAn modules.

In SeqAn module headers (e.g. `basic.h`), then all files in the module are included.

CPP File Structure

```cpp
// ==========================================================================
// $APP_NAME
// ==========================================================================
// Copyright (c) 2006-2011, Knut Reinert, FU Berlin
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of Knut Reinert or the FU Berlin nor the names of
//   its contributors may be used to endorse or promote products derived
//   from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.
//
// Author: $AUTHOR_NAME <$AUTHOR_EMAIL>
```
```cpp
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/misc/misc_cmdparser.h>
#include "app_name.h"

using namespace seqan;

// Program entry point
int main(int argc, char const ** argv)
{
    // ...
}
```

**Application Header Structure**

// ==========================================================================
// $APP_NAME
// ==========================================================================

// Copyright (c) 2006-2010, Knut Reinert, FU Berlin
// All rights reserved.
/
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
// //
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
// * Neither the name of Knut Reinert or the FU Berlin nor the names of
// its contributors may be used to endorse or promote products derived
// from this software without specific prior written permission.
// //
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.
// //
// Author: $AUTHOR_NAME <$AUTHOR_EMAIL>
// //
// $FILE_COMMENT
// ==========================================================================

2.5. SeqAn Style Guides
#ifndef CORE_APPS_APP_NAME_HEADER_FILE_H_
define CORE_APPS_APP_NAME_HEADER_FILE_H_

// Forwards

// Tags, Classes, Enums

// Class ClassName

// Metafunctions

// Metafunction MetafunctionName

// Functions

// Function functionName()

#endif // CORE_APPS_APP_NAME_HEADER_FILE_H_

// Libray Header Structure

// SeqAn - The Library for Sequence Analysis
// Copyright (c) 2006-2010, Knut Reinert, FU Berlin
// All rights reserved.

// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
// *
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
// * Neither the name of Knut Reinert or the FU Berlin nor the names of
// its contributors may be used to endorse or promote products derived
// from this software without specific prior written permission.
// *
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
Contents

• Dox API Docs
  – General Documentation Structure
  – Tag Documentation
    * @adaption
    * @aka
    * @brief
    * @class
    * @code
    * @concept
    * @defgroup
    * @deprecated
    * @enum
    * @extends
    * @fn
    * @headerfile
    * @implements
    * @include
    * @internal
    * @link
    * @macro
    * @mfn
    * @note
    * @page
    * @param
    * @return
    * @throw
    * @section
    * @see
    * @tag
    * @tparam
    * @typedef
    * @var
    * @val
    * @warning
  – Best Practice
    * Clarifying Links
    * Documentation Location
    * Signatures
  – HTML Subset
    * Tag Ordering
    * Documenting Concepts
    * Documenting Classes
    * Documenting Functions
    * Documenting Metafunctions
    * Documenting Enums
  – Difference to Doxygen
2.5.2 Dox API Docs

Since the 1.4.1 release, SeqAn uses a new documentation system. The syntax is similar to Doxygen but slightly different. The main differences are (1) not identifying functions by their signatures but only by their names, (2) adding the idea of metafunctions, (3) adding the idea of interface functions and (4) an extension to SeqAn-specific things like documenting concepts.

General Documentation Structure

Dox comments are placed in C-style comments with an exclamation mark, see below. The first dox tag should be placed on the next line, each line should begin with a correctly indented star. The first line only contains the slash-star-exclamation-mark and the last line only contains the star-slash.

```
(/^
 * @fn myFunction
 * @signature void myFunction()
 */
```

The documentation and the code are independent. Each item to be documented (adaption, class, concept, enum, function, group, macro, metafunction, page, tag, typedef, variable) has to be explicitly given (see tags below). The available top level tags are @adaption, @class, @concept, @defgroup, @enum, @fn, @macro, @mfn, @page, @tag, @typedef, and @var.

Each top level tag creates a documentation entry. For example, the following defines a class Klass with two global interface functions f1 and f2 for this class:

```
(/^
 * @class Klass
 * @fn Klass#f1
 * @fn Klass#f2
 */
```

Member functions are given using ::, the same as in the C++ language:

```
(/^
 * @class Klass
 * @fn Klass::memberFunc
 */
```

Global interface functions are global functions that belong to the interface of a type. Similar, interface metafunctions are metafunctions that belong to the interface of a type. Their fully qualified name for dox consists of the type name, followed by a hash # and the function/metafunction name:

```
(/^
 * @class Klass
 * @mfn Klass#InterfaceMetaFunc
 */
```

Below the top-level tags, come the second-level tags. The first kind of second-level tags defines properties of an entry. Important such second-level entries are @brief, @signature, @see, @param, @tparam, @return. You can also write text for the description of your entity and use tags such as @section, @subsection, @snippet, @code to format the description. You can use HTML tags for formatting the documentation.

Example:
/*!
 * @class Align
 * @brief Store a tabular alignment.
 *
 * @signature template <typename TSource, typename TSpec>
 * class Align;
 *
 * @tparam TSource The type of the underlying sequence.
 * @tparam TSpec Tag for selecting the specialization of the Align class.
 *
 * The <tt>Align</tt> class provides a tabular alignment of sequences with the
 * same type. The sequences are given with <tt>TSource</tt>. An <tt>Align</tt>
 * object will use a <a href="seqan:Gaps">Gaps</a> object for each sequence.
 * The specialization of the <tt>Align</tt> object can be selected
 * using the <tt>TSpec</tt> template parameter.
 *
 * @see Gaps
 * @see globalAlignment
 */

Images are included using <img src="${PATH}"> where ${PATH} is relative to the source image directory.

Tag Documentation

Below, we differentiate between names and labels.

Names are used to identify documentation items and must follow extended C++ identifier rules. An sub name consists
of only alphanumeric characters and the underscore is allowed, must not start with a number. Sub names can be glued
together with :: for class members and # for interface functions. In contracts, labels are used for the display to
the user. For example, the alloc string has the name AllocString but the label “Alloc String”, the constructor of
AllocString has name AllocString::String, and its length function has name AllocString::length.

@adaption

Signature @adaption AdaptionName [Adaption Label]

Top-level tag.

Definition of an adaption with the given name and an optional label.

An adaption is a collection of global interface functions and metafunctions that adapt a type outside the SeqAn library
to a concept in the SeqAn library. For example, the STL std::string class can be adapted to the interface of the
SequenceConcept concept.

/!*!
 * @adaption StdStringToSequenceConcept std::string to Sequence concept
 * @brief The <tt>std::string</tt> class is adapted to the Sequence concept.
 */

@aka

Signature @aka OtherName

Second-level entry.
Assign an alias name for a function, metafunction, class, concept, or enum. The list of aliases will be printed for each code entry. Also, the aliases will be incorporated into search results.

```cpp
/*! *
 * @class InfixSegment
 * @brief Represents a part of a string.
 *
 * @aka substring
 */

template <typename TSequence>
class InfixSegment<TSequence, Infix>;
```

@brief

**Signature** @brief Brief description.

Second-level tag.

Defines the brief description of the top-level entry it belongs to. You can use HTML in the description.

```cpp
/*! *
 * @fn f
 * @brief A minimal function.
 * @signature void f();
 */

void f();
```

@class

**Signature** @class ClassName [Class Label]

Top-level tag.

Define a class with the given name `ClassName` and an optional label.

```cpp
/*! *
 * @class AllocString Alloc String
 * @extends String
 * @brief Implementation of the String class using dynamically allocated array.
 *
 * @signature template <typename TAlphabet, typename TSpec>
 * class String<TAlphabet, Alloc<TSpec> >;
 * @tparam TAlphabet Type of the alphabet (the string’s value).
 * @tparam TSpec Tag for the further specialization.
 */

template <typename TAlphabet, typename TSpec>
class String<TAlphabet, Alloc<TSpec> >
{
    // ...
};
```
@code

Signature @code{.ext} ... @endcode
Second-level tag.
Provides the means to display code blocks in the documentation. The extension .ext is used for identifying the type (use .cpp for C++ code) and selecting the appropriate highlighting.

```cpp
/*! *
 * @fn f
 * @brief Minimal function.
 * @signature void f();
 * *
 * @code{.cpp}
 * int main()
 * {
 *   f(); // Call function.
 *   return 0;
 * }
 * @endcode
 */

void f();

Note that you can use the extension value .console to see console output.

```cpp
/*! *
 * @fn f
 * @brief Some function
 * *
 * @section Examples
 * *
 * @include demos/module/demo_f.cpp
 * *
 * The output is as follows:
 * *
 * @code{.console}
 * This is some output of the program.
 * @endcode
 */
```

@concept

Signature @concept ConceptName [Concept Label]
Top-level tag.
Create a documentation entry for a concept with the given name and an optional label. All concept names should have the suffix Concept. Use the fake keyword concept in the @signature.

A concept is the C++ equivalent to interfaces known in other classes. C++ provides no real way for concepts so at the moment they are a formal construct used in the documentation.

```cpp
/*! *
 * @concept SequenceConcept Sequence
 * @signature concept SequenceConcept;
 * @extends ContainerConcept
 */
```
@brief Concept for sequence types.
*/

@defgroup

**Signature** @defgroup GroupName [Group Label]

Top-level tag.

Create a documentation entry for a group with a given name and an optional label. Groups are for rough grouping of global functions and/or tags.

You can put types and functions into a group similar to making global interface functions and metafunctions part of the interface of a class or concept.

/!*!
 * @defgroup FastxIO FASTA/FASTQ I/O
 * @brief Functionality for FASTA and FASTQ I/O.
 *
 * @fn FastxIO#readRecord
 * @brief Read one record from FASTA/FASTQ files.
 *
 * @fn FastxIO#writeRecord
 * @brief Write one record to FASTA/FASTQ files.
 *
 * @fn FastxIO#readBatch
 * @brief Read multiple records from FASTA/FASTQ file, limit to a given count.
 *
 * @fn FastxIO#writeBatch
 * @brief Write multiple records to FASTA/FASTQ file, limit to a given count.
 *
 * @fn FastxIO#readAll
 * @brief Read all records from a FASTA/FASTQ file.
 *
 * @fn FastxIO#writeAll
 * @brief Write all records to a FASTA/FASTQ file.
 */

@deprecated

**Signature** @deprecated message

Second-level entry.

Mark a given function, metafunction, class, concept, or enum as deprecated. A deprecation message will be generated in the API documentation.

/!*!
 * @fn f
 * @deprecated Use @link g @endlink instead.
 * @brief Minimal function.
 */

void f();
@enum

Signature @enum EnumName [Enum Label]

Top-level entry.

Documentation for an enum with given name and optional label.

```c
/*! 
 * @enum MyEnum 
 * @brief An enum. 
 * 
 * @val MyEnum VALUE1 
 * @brief VALUE1 value of enum MyEnum. 
 * 
 * @val MyEnum VALUE2 
 * @brief VALUE2 value of enum MyEnum. 
 */
enum MyEnum 
{
    VALUE1,
    VALUE2
};
```

@extends

Signature @extends OtherName

Gives a parent class for a given class or a parent concept for a given concept.

```c
/*! 
 * @concept OneConcept 
 * 
 * @concept TwoConcept 
 * @extends OneConcept 
 * 
 * @class MyClass 
 * 
 * @class OtherClass 
 * @extends MyClass 
 */
```

@fn

Signature @fn FunctionName [Function Label]

Top-level entry.

Document a function (global, global interface, or member) with given name and label. The type of the function is given by its name.

```c
/*! 
 * @fn globalAlignment 
 * @brief Pairwise, DP-based global alignment. 
 */
```
@headerfile

**Signature** @headerfile path

Second-level entry.

Give the required `#include` path for a code entry.

**Note:** Use angular brackets as below for SeqAn includes.

```c
/*! 
 * @fn f 
 * @brief A minimal function. 
 * @headerfile <seqan/module.h>
 */
```

@implements

**Signature** @implements ConceptName

Second-level entry.

Marks a class to implement a given concept.

```c
/*! 
 * @concept MyConcept 
 */
```

@include

**Signature** @include path/to/file

Second-level entry.

Include a C++ source file as an example. See [#snippet @snippet] for including fragments.

```c
/*! 
 * @fn f 
 * @brief Minimal function. 
 */
```

@internal

**Signature** @internal [ignored comment]

Second-level entry.

Mark a given function, metafunction, class, concept, or enum as internal. You can also provide a comment that is ignored/not used in the output.
/**
 * @fn f
 * @internal
 * @brief Minimal function.
 */

void f();

@link

**Signature** @link TargetName target label

In-text tag.

Tag to link to a documentation entry with a given label.

The difference to [#see @see] is that @link .. @endlink is used inline in text whereas @see is a second-level tag and adds a see property to the documented top-level entry. Use @link to link to entries within the documentation and the HTML <a> tag to link to external resources.

/**
 * @fn f
 * @brief Minimal function.
 * 
 * The function is mostly useful with the @link String string class@endlink.
 */

@macro

**Signature** @macro MacroName [Macro Label]

Top-level tag.

Document a macro.

/**
 * @macro MY_MACRO
 * @brief Multiply two values.
 * 
 * @signature #define MY_MACRO(i, j) ...
 * @param i A value for i.
 * @param j A value for j.
 * @return The product of i and j: (i * j)
 */

#define MY_MACRO(i, j) (i * j)

@mfn

**Signature** @mfn MetafunctionName [Metafunction Label]

Top-level tag.

Document a metafunction.
@brief Identity function for types.
*
* @signature Identity<T>::Type
* @tparam T The type to pass in.
* @returns The type T.
 */

template <typename T>
struct Identity
{
    typedef T Type;
};

@note

Signature @note message
Second-level entry.
Add an informative note to a function, metafunction, class, concept, enum, or group.

/*@!
* @fn f
* @note Very useful if used together with @link g @endlink.
* @brief Minimal function.
*/

void f();

@page

Signature @page PageName [Page Title]
Top-level entry.
Create a documentation page.

/*@!
* @page SomePage Page Title
* 
* A very simple page
* 
* @section Section
* 
* A section!
* 
* @subsection Subsection
* 
* A subsection!
*/

@param

Signature @param Name Label

2.5. SeqAn Style Guides
Document a value (and non-type) parameter from a function or member function.

```c++
int square(int x);
```

@return

**Signature** @return Type Label

Define the return value for a function or metafunction.

Also see the example for [#param @param].

When documenting functions and the result type is the result of a metafunction then use a TXYZ return type in @return and document TXYZ in the text of @return as follows:

```c++
template<typename TContainer>
typename Size<TContainer>::Type lengthSquare(TContainer const & c);
```

@throw

**Signature** @return Exception Label

Add note on a function or macro throwing an exception.

```c++
void myFunction(char const * filename);
```
@section

**Signature @section Title**

Second-level entry.

Adds a section to the documentation of an entry.

See the example for [#page @page].

@see

**Signature @see EntryName**

Second-level entry.

Add “see also” link to a documentation entry.

```c
/*!
 * @fn f
 * @brief A simple function.
 * * Here is a snippet:
 * * @snippet core/demos/use_f.cpp Simple Function
 */
```

And here is the file with the snippet.

```c
int main(int argc, char const ** argv)
{
   //![Simple Function]
   return 0;
   //![Simple Function]
}
```

/* Some more code */

@tag

**Signature @tag TagName**

Top-level entry.

Document a tag. Mostly, you would group tags in a group using [#defgroup @defgroup].

```c
/*!
 * @defgroup MyTagGroup My Tag Group
 * * @tag MyTagGroup#TagName
 * * @tag MyTagGroup#MyOtherTagName
 */
```
@tparam

Signature @tparam TArg
Second-level entry.
Document a template parameter of a metafunction or class template.

/*! 
 * @mfn MetaFunc
 * @signature MetaFunc<T1, T2>::Type
 *
 * @tparam T1 First type.
 * @tparam T2 Second type.
 */

@typedef

Signature @typedef TypedefName
Top-level entry.
Document a typedef.

/*! 
 * @typedef CharString
 * @brief An AllocString of character.
 *
 * @signature typedef String<char, Alloc<> > CharString;
 */

@var

Signature @var VariableType VariableName
Top-level entry. Document a global variable or member variable.

/*! 
 * @class MyClass
 *
 * @var int MyClass::iVar
 */

```cpp
class MyClass
{
public:
    int iVar;
};
```

@val

Signature @val EnumType EnumValueName
Top-level entry. Document an enum value.
/*!
* @enum EnumName
* @brief My enum.
* @signature enum EnumName;
*
* @val EnumName::VALUE1;
* @brief The first enum value.
*
* @val EnumName::VALUE2;
* @brief The second enum value.
*/

enum MyEnum
{
    VALUE1,
    VALUE2
};

@warning

Signature @warning message
Second-level entry.
Add a warning to a function, metafunction, class, concept, enum, or group.

/*!
* @fn f
* @note Using this function can lead to memory leaks. Try to use @link g @endlink instead.
* @brief Minimal function.
*/

void f();

Best Practice
This section describes the best practice when writing documentation.

Clarifying Links
Our usability research indicates that some functionality is confusing (e.g. see #1050) but cannot be removed. One example is the function reserve() which can be used to increase the capacity of a container whereas the function resize() allows to change the size of a container, increasing or decreasing its size.

The documentation of such functions should contain a clarifying text and a link to the other function.

/*!
* @fn Sequence#reserve
* Can be used to increase the <b>capacity</b> of a sequence.
* Note that you can only modify the capacity of the sequence. If you want to modify the <b>length</b> of the sequence then you have to use @link Sequence#resize @endlink.
*/
Documentation Location

Add the documentation where it belongs. For example, when documenting a class with multiple member functions, put the doc comments for the class before the class, the documentation of the member functions in front of the member functions. For another example, if you have to define multiple signatures for a global interface function or metafunctions, put the documentation before the first function.

```cpp
/*! *
 * @class Klass
 * @brief A class.
 */
class Klass
{
public:
    /*! *
     * @var int Klass::x
     * @brief The internal value.
     */
    int x;

    /*! *
     * @fn Klass::Klass
     * @brief The constructor.
     *
     * @signature Klass::Klass()
     * @signature Klass::Klass(i)
     * @param i The initial value for the member <tt>x</tt> (type <tt>int</tt>).
     */
    Klass() : x(0)
    {}
    Klass(int x) : x(0)
    {}

    /*! *
     * @fn Klass::f
     * @brief Increment member <tt>x</tt>
     */
    void f()
    {
        ++x;
    }
};
```

Signatures

Always document the return type of a function. If it is the result of a metafunction or otherwise depends on the input type, use TResult or so and document it with @return.

HTML Subset

You can use inline HTML to format your description and also for creating links.

- Links into the documentation can be generated using `<a>` if the scheme in `href` is `seqan:`:
  `<a href="seqan:AllocString">the alloc string</a>`.  

594 Chapter 2. API Documentation
• Use <i> for italic/emphasized text.
• Use <b> for bold text.
• Use <tt> for typewriter text.

Tag Ordering

Keep consistent ordering of second-level tags. The following order should be used, i.e. if several of the following tags appear, they should appear in the order below.

1. @internal
2. @deprecated
3. @warning
4. @note
5. @brief
6. @extends
7. @implements
8. @signature
9. @param
10. @tparam
11. @return
12. @headerfile
13. The documentation body with the following tags in any order (as fit for the documentation text) and possibly interleaved with text: @code, @snippet, @include, @section, @subsection.
14. @see
15. @aka

Documenting Concepts

All concepts should have the suffix Concept.

Use the pseudo keyword concept in the @signature.

Use the following template:

```
/*! *
 * @concept MyConcept
 * @brief The concept title.
 * *
 * @signature concept MyConcept;
 * *
 * The concept description possibly using include, snippet, and <b><i>formatting</i></b> etc.
 */
```
Documenting Classes

Use the following template:

```
/*! *
 * @class AllocString Alloc String
 * @brief A string storing its elements on dynamically heap-allocated arrays.
 * *
 * @signature template <typename TAlphabet, typename TSpec>
 * class AllocString<TAlphabet, Alloc<TSpec> >;
 * @tparam TAlphabet The alphabet/value type to use.
 * @tparam TSpec The tag to use for further specialization.
 * *
 * The class description possibly using include, snippet, and <b><i>formatting</i></b> etc.
 */
```

Documenting Functions

Use the following template:

```
/*! *
 * @fn globalAlignment
 * @brief Global DP-based pairwise alignment.
 * *
 * @signature TScore globalAlignment(align, scoringScheme);
 * @signature TScore globalAlignment(align, scoringScheme, lowerBand, upperBand);
 * @param align Align object to store the result in. Must have length 2 and be filled with sequences.
 * @param scoringScheme Score object to use for scoring.
 * @param lowerBand The lower band of the alignment (<tt>int</tt>).
 * @param upperBand The upper band of the alignment (<tt>int</tt>).
 * @return TScore The alignment score of type <tt>Value<TScore>::Type</tt> where <tt>TScore</tt> is the type of <tt>scoringScheme</tt>.
 * *
 * The function description possibly using include, snippet, and <b><i>formatting</i></b> etc.
 */
```

Documenting Metafunctions

Use the following template:

```
/*! *
 * @mfn Size
 * @brief Return size type of another type.
 * *
 * @signature Size<T>::Type
 * @tparam T The type to query for its size type.
 * @return TSize The size type to use for T.
 * *
 * The class description possibly using include, snippet, and <b><i>formatting</i></b> etc.
 */
```

Documenting Enums
Difference to Doxygen

If you already know Doxygen, the following major differences apply.

- The documentation is more independent of the actual code. Doxygen creates a documentation entry for all functions that are present in the code and allows additional documentation, e.g. using `@fn` for adding functions. With the SeqAn dox system, you have to explicitly use a top-level tag for adding documentation.
- Documentation entries are not identified by their signature but by their name.
- We allow the definition of interface functions and metafunctions (e.g. `@fn Klass#func` and `@mfn Klass#Func`) in addition to member functions (`@fn Klass::func`).
- We do not allow tags with backslashes but consistently use at signs (`@`).

2.5.3 SeqAn JavaScript Style Guide

Follow the *SeqAn C++ Style Guide* in spirit.

2.5.4 SeqAn Python Style Guide

Some very few points:

- Follow PEP 8.
- Use single-quotes for strings, i.e. ‘this is a string’ and double-quotes for docstrings, e.g. """This is a docstring."""
- Name functions, classes, constants as in SeqAn, variables and member variables are named `lower_case_with_underscores`.

2.6 Glossary

2.6.1 Suffix Tree

We consider an alphabet $\Sigma$ and a sentinel character $\$ $ that is smaller than every character of $\Sigma$. A suffix tree of a given non-empty string $s$ over $\Sigma$ is a directed tree whose edges are labeled with non-empty substrings of $s\$ $ with the following properties:

1. Each outgoing edge begins with a different letter and the outdegree of an internal node is greater than 1.
2. Each suffix of $s\$ $ is the concatenation of edges from the root to a leaf node.
3. Each path from the root to a leaf node is a suffix of $s$.

The following figure shows the suffix tree of the string $s$="mississippi" (suffix nodes are shaded):

![Suffix tree of "mississippi"](image)

Many suffix tree construction algorithms expect $\$\$ to be part of the string alphabet which is undesirable for small bit-compressible alphabets (e.g. DNA). In SeqAn there is no need to introduce a $. We relax suffix tree criterion 2. and consider the relaxed suffix tree that arises from the suffix tree of $s$ by removing the $\$\$ character and all empty edges. In the following, we only consider relaxed suffix trees and simply call them suffix trees. In that tree a suffix can end in an inner node as you can see in the next figure (suffix "i"):

2.7 References
Figure 2.12: Figure 2: Relaxed suffix tree of “mississippi”


