alice Documentation

Release v0.3

Mathias Soeken

Sep 02, 2018

Contents

CHAPTER 1

Installation

alice is a header-only C++-14 library. Just add the include directory of alice to your include directories, and you can integrate alice into your source files using

```
#include <alice/alice.hpp>
```

1.1 Compile with readline

Alice can use the *readline* library to enable command completition and history. If one integrates alice through *CMake*, then *readline* is enabled by default. Otherwise, make sure to define `READLINE_USE_READLINE` and link against *readline*.

1.2 Building examples

In order to build the examples, you need to enable them. Run the following from the base directory of alice:

```
git submodule update --init --recursive
mkdir build
cd build
cmake -DALICE_EXAMPLES=ON ..
make
```

1.3 Building tests

In order to run the tests and the micro benchmarks, you need to enable tests in CMake:

```
git submodule update --init --recursive
mkdir build
cd build
cmake -DALICE_TEST=ON ..
make
./test/run_tests
```

CHAPTER 2

Console, Pyton, and C interface

Alice supports different usage modes. By default the code is compiled to a stand-alone executable. But the same source code can also be compiled into a Python module or C library.

- **Console mode:** the source code is compiled as **executable** and a **stand-alone program** is offering a console shell interface, which accepts commands.
- **Python mode:** the source code is compiled as **Python module** which offers an API according to the commands. Commands become function names, and command arguments become function arguments.
- **C library mode:** the source code is compiled as **C library** with functions call commands. This is useful to interface with Alice CLIs from other languages.

Which mode is taken is determined by compilation. No changes in the source code are necessary, when making use of the *Macro API*.

Let's say we have a source file *shell.cpp*, which defines an alice CLI as explained in the tutorials of this manual. Then add the following lines into a CMakeLists.txt file in order to compile it as an executable in console mode:

```
add_executable(shell shell.cpp)
target_link_libraries(shell alice)
```

The same file can be compiled as a Python module with the following command:

```
add_alice_python_module(shell shell.cpp)
```

This creates a compile target `shell_python`.

In order to compile it as C library one uses the following commands:

```
add_alice_c_library(shell shell.cpp)
```

This creates a compile target `shell_c`.

Note: The name of the Python module *must* equal the name of the prefix that was used in the *`ALICE_MAIN`* macro. Our example file shell.cpp must finish with `ALICE_MAIN(shell)`.

2.1 Shell commands as Python functions

If the alice shell has the prefix `shell`, then the corresponding Python module has the name *shell* and can be imported as follows:

```
import shell
```

Commands are mapped into Python functions with the same name. Assume there is a command called `command`, then one can call it from Python as follows:

```
import shell
shell.command()
```

Long option and flag names are mapped into keyword arguments of the corresponding Python command. Assume that the command `command` has the following synopsis:

```
shell> command -h
A test command
Usage: command [OPTIONS]

Options:
  -h,--help                    Print this help message and exit
  -s,--sopt TEXT               A string option
  -n,--nopt INT                An integer option
  -f,--flag                    A flag
```

Then the individual arguments in this command can be called in Python mode as follows:

```
import shell
shell.command(sopt = "Some text", nopt = 42, flag = True)
```

The order in which the keyword arguments are passed does not matter; also, not all of them need to be provided. Note again, that the short option and flag names cannot be used in Python mode. Also flags must be assigned a Boolean value. Assigning `False` to a flag argument is as omitting it.

The return value of a Python function corresponds to the logging output of the corresponding command. Each command can contribute to the log by implementing the `log()` function. It returns a JSON object. The return value of the function in Python mode can be considered as a Python `dict`, in which the entries correspond to the JSON object.

Assume that the example command `command` implements the following `log()` function:

```
nlohmann::json log() const
{
  return nlohmann::json({
    {"str", "Some string"},
    {"number", 42}
  });
}
```

Then one can access these values from the return value of the Python function:

```
import shell
r = shell.command()
print(r["number"])     # Prints 42
```

2.2 C library

Assuming that the alice shell has the prefix `shell`, then the C library will implement the following three functions:

```
extern void* shell_create();
extern void shell_delete( void* cli );
extern int shell_command( void* cli, const char* command, char* log, size_t size );
```

The prefix is being used as prefix for the C functions. By copying the above three lines into a C file and linking to the compiled C library allows to interact with the alice CLI shell.

The first two functions `shell_create` and `shell_delete` create and delete a CLI object. Note that the object is passed as `void*`. The third function calls a single command. The first argument is a pointer to a CLI object and the second argument is the command as string. The third argument is a string pointer which can be passed to store the JSON log produced by the command; it can also be null. If not null, the last argument should contain the maximum size of the `log` string. The function returns -1, if the command was not executed successfully, 0, if the command was executed successfully, but nothing was written into the `log` string, and otherwise the actual size of the JSON string. The actual size may be longer than `size`.

Being a C library, it can also be used in other languages, e.g., in C#. In the next example, we assume that the library has been compiled on a Linux machine and has the name `libshell_c.so`:

```
using System;
using System.Runtime.InteropServices;
using System.Text;

public class Library {
  [DllImport("libshell_c.dylib", EntryPoint = "shell_create")]
  public static extern IntPtr shell_create();

  [DllImport("libshell_c.dylib", EntryPoint = "shell_delete")]
  public static extern void shell_delete(IntPtr cli);

  [DllImport("libshell_c.dylib", EntryPoint = "shell_command")]
  public static extern int shell_command(IntPtr cli, string command, StringBuilder␣
↪json, int size);
}
```

CHAPTER 3

Change Log

3.1 v0.3 (July 22, 2018)

- Throw and catch errors in *read*.
- General commands: `ps --all` to show statistics of all store entries #5
- General commands: Read multiple files in `read` #6
- Support for default store option (enabled via setting `ALICE_SETTINGS_WITH_DEFAULT_OPTION`) #7
- General commands: `store --pop` to remove current store element #8
- Automatic `to_<tag>` in Python interface as shortcut for `write_<tag>(log=True)["contents"]` #9

3.2 v0.2 (May 7, 2018)

- Validators: `ExistingFileWordExp`
- C library interface #1
- General commands: `write_<format> --log` to write file contents to log #2

3.3 v0.1 (January 11, 2018)

- Initial release
- General commands: `alias`, `convert`, `current`, `help`, `print`, `ps`, `quit`, `set`, `show`, `store`
- Shell application command line flags: `--command`, `--filename`, `--echo`, `--counter`, `--interactive`, `--log`

- Macro API: `ALICE_MAIN`, `ALICE_ADD_STORE`, `ALICE_DESCRIBE_STORE`, `ALICE_PRINT_STORE`, `ALICE_PRINT_STORE_STATISTICS`, `ALICE_LOG_STORE_STATISTICS`, `ALICE_CONVERT`, `ALICE_SHOW`, `ALICE_STORE_HTML`, `ALICE_ADD_COMAND`, `ALICE_COMMAND`, `ALICE_READ_FILE`, `ALICE_WRITE_FILE`, `ALICE_ADD_FILE_TYPE`, `ALICE_ADD_FILE_TYPE_READ_ONLY`, `ALICE_ADD_FILE_TYPE_WRITE_ONLY`
- Python special features: `__repr__` (from `print`), and `_repr_html_` (from `html_repr`)

CHAPTER 4

Tutorial 1: A minimalistic example

This tutorial shows a minimal example, the barely minimum what needs to be written in order to get an Alice shell. The source files for this tutorial are located in `examples/tutorial1`.

```
#include <alice/alice.hpp>

ALICE_MAIN( tutorial1 )
```

That's all! Two lines of code suffice. The first line includes the Alice header `alice/alice.hpp`. In all use cases, this will be the only header that needs to be included. The second line calls *ALICE_MAIN*, which takes as argument a name for the shell. Besides acting as the prompt, it will also be used as a name for the Python library, if it is build.

Compile `tutorial1.cpp` and link it to the `alice` interface library; have a look into `examples/CMakeLists.txt` to check the details. Even though we only wrote two lines of code, we already can do several things with the program. When executing the program (it will be in `build/examples/tutorial1`), we can enter some commands to the prompt:

```
tutorial1> help
General commands:
 alias            help             quit             set
```

It shows that the shell has 4 commands: `alias`, `help`, `quit`, and `set`. Further information about each commands can be obtained by calling it with the `-h` flag. We'll get to `alias` later. Command `help` lists all available commands, and it also allows to search through the help texts of all commands. Command `quit` quits the program. Command `set` can set environment variables that can be used by other programs. Possible variables and values are listed in the help strings to such commands.

CHAPTER 5

Tutorial 2: Adding a store and writing a simple command

We extend on the previous example and add a store to the shell. A shell can have several stores, each is indexed by its type.

```
#include <alice/alice.hpp>

#include <string>

namespace alice
{

ALICE_ADD_STORE( std::string, "str", "s", "string", "strings" )

ALICE_PRINT_STORE( std::string, os, element )
{
  os << element << std::endl;
}

ALICE_COMMAND( hello, "Generation", "Generates a welcome string" )
{
  store<std::string>().extend() = "hello world";
}

}

ALICE_MAIN( tutorial2 )
```

The macro *ALICE_ADD_STORE* registers a store for strings (using type `std::string`). The type is the first argument to the macro. The other four are used to build commands. The values `str` and `s` are long and short flag names, respectively, and will be used to select this type in several store-related commands, e.g., `print --str` or `print -s` to print a string to the terminal. The last two arguments are a singular and plural name that is used to generate help strings in store-related commands. Let's have a look what `help` shows for this tutorial:

```
tutorial2> help
Generation commands:
```

(continues on next page)

(continued from previous page)

```
 hello

General commands:
 alias              convert            current            help
 print              ps                 quit               select
 show               store
```

First, we see two categories of commands, the first one (*Generation commands*) listing the custom command `hello`. We'll get to that one in a bit. There are also several other general commands compared to the previous tutorial. These are called store-related commands are as follows:

`convert`	Converts a store element of one type into another
`current`	Changes the current store element
`print`	Prints the current store element
`ps`	Prints statistics about the current store element
`show`	Creates and shows a visual representation of the current store element
`store`	Shows a summary of store elements

In each command the type of store must be addressed by the flag name that was defined for the store in *`ALICE_ADD_STORE`*. For example, `print -s` prints the current element from the string store to the terminal. The code provided by the *`ALICE_PRINT_STORE`* macro is used to describe what should be printed for the specific store type. In case of this string store, we just print the string followed by a new line.

One new command is added using the macro *`ALICE_COMMAND`*. This macro only allows us to add very simple commands, with no custom arguments, and no custom logging behavior (we will see how to create more advanced commands in the next tutorials). The command `hello` is defined using two other arguments to the macro, the second being a category used to partition commands when calling `help`, the third being a description text that is printed when calling `hello -h` and is used by `help -s` to find commands. The code accesses the store of strings, using `store<std::string>()` and extends it by one element using the method `extend()`. Since `extend()` returns a reference to the newly created element, we can assign it the value `"hello world"`. Let's have a look at a session that makes use of the new command:

```
tutorial2> store -s
[i] no strings in store
tutorial2> hello
tutorial2> store -s
[i] strings in store:
  *  0:
tutorial2> print -s
hello world
tutorial2> quit
```

CHAPTER 6

Tutorial 3: Read and write commands

This tutorial shows how to integrate read and write functions to the shell interface. The general mechanism works as follows. We first define a file type, and name it using a tag. Then we can enable read and write commands for this tag to a store type, which result in commands `read_<tag> --<flag>` and `write_<tag> --<flag>`, where `<tag>` is the tag name of the file type and `<flag>` is the flag name of the store type.

Our tutorial catches up where we left in the previous tutorial and defines a single store type for strings, and adds a function to print store elements.

```
#include <alice/alice.hpp>

#include <algorithm>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

namespace alice
{

ALICE_ADD_STORE( std::string, "str", "s", "string", "strings" )

ALICE_PRINT_STORE( std::string, os, element )
{
  os << element << std::endl;
}
```

We now add the file type to read and write to text files using the macro *ALICE_ADD_FILE_TYPE*, which receives the tag name as first argument and a string defining it as second argument. The second argument will be used to generate help strings.

```
ALICE_ADD_FILE_TYPE( text, "Text" )
```

If we wish to only read from or only write to a file type, we can use the macros *ALICE_ADD_FILE_TYPE_READ_ONLY* or *ALICE_ADD_FILE_TYPE_WRITE_ONLY*, respectively. They have the same signature.

Once the file type is declared, we can link the string store type to the file type and add functions to read from files and to write from files. Reading from files is enabled using the macro *ALICE_READ_FILE*, which receives four parameters. The first two parameters are store type and tag name. The third parameter is the variable name containing the filename, and the last parameter gives access to the command parsing interface, which we won't use in this tutorial.

```
ALICE_READ_FILE( std::string, text, filename, cmd )
{
  std::ifstream in( filename.c_str(), std::ifstream::in );
  std::stringstream buffer;
  buffer << in.rdbuf();
  return buffer.str();
}
```

Similarly, we enable writing from files using the macro *ALICE_WRITE_FILE*. It receives one further parameter called `element`, which is a variable accessing the current store element that should be written to a file.

```
ALICE_WRITE_FILE( std::string, text, element, filename, cmd )
{
  std::ofstream out( filename.c_str(), std::ofstream::out );
  out << element;
}
```

That's all we need to read and write from files. Finally, we add one further command to manipulate store entries. The command `upper` will allow to change string elements into upper case.

```
ALICE_COMMAND(upper, "Manipulation", "changes string to upper bound")
{
  auto& str = store<std::string>().current();
  std::transform( str.begin(), str.end(), str.begin(), ::toupper );
}

}

ALICE_MAIN( tutorial3 )
```

CHAPTER 7

Tutorial 4: Two stores and conversion

We are now describing an example in which we use two store elements, one for strings and for for integers, and we add the possibility to convert an element from the integer store into an element from the string store.

We start by defining two stores and also add methods to print the store entries using `print`. Using the macro *ALICE_DESCRIBE_STORE* we can return a short description string that is used to summarize a store element when showing the store content with `store`.

```
#include <alice/alice.hpp>

#include <fmt/format.h>
#include <string>
#include <vector>

namespace alice
{

ALICE_ADD_STORE( std::string, "str", "s", "string", "strings" )
ALICE_ADD_STORE( int, "number", "d", "number", "numbers" )

ALICE_DESCRIBE_STORE( std::string, element )
{
  return fmt::format( "{} characters", element.size() );
}

ALICE_PRINT_STORE( std::string, os, element )
{
  os << element << std::endl;
}

ALICE_DESCRIBE_STORE( int, element )
{
  return element < 10 ? "small number" : "large number";
}
```

(continues on next page)

(continued from previous page)

```
ALICE_PRINT_STORE( int, os, element )
{
  os << element << std::endl;
}
```

Using the macro `ALICE_CONVERT` we can enable conversion from entry to another, which is performed by the convert command. We plan to write a conversion routine from integers (flag name `"number"`) to strings (flag name `"string"`). By implementing the following macro, we will add a flag `--number_to_string` to the `convert` command.

```
ALICE_CONVERT( int, element, std::string )
{
  if ( element >= 0 && element < 10 )
  {
    return std::vector<std::string>{"zero", "one", "two", "three",
                                    "four", "five", "six", "seven",
                                    "eight", "nine"}[element];
  }
  else
  {
    return "many";
  }
}
```

The macro `ALICE_CONVERT` expects three arguments. The first argument is the store type that we wish to convert from, followed by an identifier that we use to access the current element from that store in the implementation of the macro. The third argument is the store type that we wish to convert to. The code will convert all numbers between 0 and 9 to respective strings, and otherwise to `"many"`.

Next, we implement a command `number` to load a number into the store. We wish to specify the number using a command option `--load`. Since we need to initialize command arguments in the constructor, we cannot use the `ALICE_COMMAND` macro but need to implement our own class by deriving from the `alice::command` base class. Make sure to use the `_command` suffix when giving a name to the new command. We can add the command using the `ALICE_ADD_COMMAND` after defining the command. The second argument to this macro is a category that is used by the `help` command.

```
class number_command : public command
{
public:
  explicit number_command( const environment::ptr& env )
      : command( env, "reads a number" )
  {
    opts.add_option( "--load,load", number, "number to load to the store" )->
↪required();
  }

protected:
  void execute()
  {
    env->store<int>().extend() = number;
  }

private:
  int number{};
};
```

(continues on next page)

(continued from previous page)

```
ALICE_ADD_COMMAND( number, "Generation" )
}
```

Finally, we call the main macro for alice.

```
ALICE_MAIN( tutorial4 )
```

CHAPTER 8

Tutorial 5: Showing store elements

We extend the example from *Tutorial 4: Two stores and conversion* by functionality that allows to show store entries for strings and numbers with `show -s` and `show -d`, respectively.

8.1 Showing strings

For strings, we will use SVG as visualization format, and since we are not adding any customized user settings, we can simply add the following code to the example:

```
ALICE_SHOW( std::string, "svg", os, element )
{
  const auto svg = R"svg(
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
  <text x="0" y="36" font-size="36" font-family="Verdana">{}</text>
</svg>
  )svg";

  os << fmt::format( svg, element );
}
```

We are using the *ALICE_SHOW* macro to implement the functionality. The first argument to the macro is the store type, followed by a default extension. This extension is used to create temporary filenames, if no filename argument is provided to `show`. The third argument is a reference to an output stream and the last argument is a reference to the current store element. In the implementation, we prepare an SVG string (using C++'s raw string literal), and insert the element text.

We can now show a string element using the default application as follows:

```
tutorial5> number 5; convert --number_to_str; print -s
five
tutorial5> show -s
```

We can also override the default program by explicitly specifying it using the `--program` option. The value should contain a `{}` where the filename is being inserted:

```
tutorial5> show -s --program "open -a \"Google Chrome\" {}"
```

The last command opens the generated SVG file using the Chrome web browser (`open` command works only in Mac OS).

8.2 Showing numbers

The number stores should be visualizes as PS file, and the font size should be customizable by the user. The macro *ALICE_SHOW* does not allow to add options to the command; instead, we override the functions *can_show* and *show* directly:

```
template<>
bool can_show<int>( std::string& extension, command& cmd )
{
  extension = "ps";

  cmd.add_option<unsigned>( "--fontsize", "font size" )->group( "Numbers" );

  return true;
}
```

In the implementation of `can_show`, we first define the default extension to be `"ps"`. Afterwards, we add a command option `--fontsize`, which can take an `unsigned int` as a value. We also add it to the option group *Numbers* to organize the help string for `show -h`. Finally, we return `true` to enable the `show` command for numbers. This will add the `-d` flag to the command.

```
template<>
void show<int>( std::ostream& os, const int& element, const command& cmd )
{
  const auto ps = R"ps(
%!PS
/inch {{72 mul}} def

/Times-Roman findfont {} scalefont setfont
2.5 inch 5 inch moveto
({}) show

showpage
  )ps";

  const auto fontsize = cmd.option_value<unsigned>( "--fontsize", 30 );
  os << fmt::format( ps, fontsize, element );
}
```

Similar as to the implementation for string visualization, we first create the output for the PostScript visualization and leave to placeholders for the font size and the actual number to print. The font size is read using the function `option_value`, which takes as first parameter the same option name that was given to `add_option` and as second parameter a default value. Note that the type argument `unsigned` must match the type that was used for `add_option`.

CHAPTER 9

Macro API

Macro	Description
ALICE_MAIN	Alice main routine.
ALICE_ADD_STORE	Adds a store.
ALICE_DESCRIBE_STORE	Returns a one-line string to show when printing store contents.
ALICE_PRINT_STORE	Prints a store element to the terminal.
ALICE_PRINT_STORE_STATISTICS	Prints statistics about a store element to the terminal.
ALICE_LOG_STORE_STATISTICS	Prints statistics about a store element to the terminal.
ALICE_CONVERT	Converts store element into another store element.
ALICE_SHOW	Shows store element.
ALICE_STORE_HTML	Generates HTML output for a store element.
ALICE_ADD_COMMAND	Add a command.
ALICE_COMMAND	Add and implements a simple command.
ALICE_READ_FILE	Read from a file into a store.
ALICE_WRITE_FILE	Write to a file from a store.
ALICE_ADD_FILE_TYPE	Registers a file type to alice.
ALICE_ADD_FILE_TYPE_READ_ONLY	Registers a read-only file type to alice.
ALICE_ADD_FILE_TYPE_WRITE_ONLY	Registers a write-only file type to alice.

ALICE_MAIN (prefix)

Alice main routine.

The use of this macro is two-fold depending on whether alice is used for a stand-alone application or for creating a Python library:

- In stand-alone application mode, this starts the interactive shell which accepts commands and replaces the C++ `main` method. The prefix in the shell will be taken from the first argument.
- In Python mode, this method will create a Python module with name `prefix`.

Parameters

- `prefix`: Shell prefix or python module name (depending on mode)

9.1 Stores

`ALICE_ADD_STORE` (type, _option, _mnemonic, _name, _name_plural)

Adds a store.

Adds a new store type alice. The first parameter is a type, all other parameters are for program arguments and help strings in the general store-related commands.

Parameters

- `type`: A type to define the store
- `_option`: Long option name to select store in general store commands
- `_mnemonic`: Short option name (single character) to select store in general store commands (cannot be `n` or `v`)
- `_name`: Singular name used in help texts
- `_name_plural`: Plural name used in help texts

`ALICE_DESCRIBE_STORE` (type, element)

Returns a one-line string to show when printing store contents.

This macro is used to return a string that is shown in the output of `store`, when each store entry is listed with its index and a short one-line descriptiont text.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `element`: Reference to the store element

`ALICE_PRINT_STORE` (type, os, element)

Prints a store element to the terminal.

This macro is used to generate the code that is executed when calling `print` for a store.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `os`: Output stream (default is `std::cout` when in standalone mode)
- `element`: Reference to the store element

`ALICE_PRINT_STORE_STATISTICS`(type, os, element)

Prints statistics about a store element to the terminal.

This macro is used to generate the output that is printed when calling `ps` for a store.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `os`: Output stream (default is `std::cout` when in standalone mode)
- `element`: Reference to the store element

`ALICE_LOG_STORE_STATISTICS`(type, element)

Prints statistics about a store element to the terminal.

This macro is used to generate the JSON object that is logged when calling `ps` for a store element. The body must return an `nlohmann::json` object.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `element`: Reference to the store element

`ALICE_CONVERT`(from, element, to)

Converts store element into another store element.

This macro adds an implementation for conversion of a store element of type `from` to a store element of type `to`. It causes a new option `--<from>_to_<to>` for the `convert` command.

The macro must be followed by a code block.

Return New store element

Parameters

- `from`: Store type that should be converted from
- `element`: Reference to the store element that should be converted
- `to`: Store type that should be converted to

`ALICE_SHOW`(type, extension, os, element)

Shows store element.

This macro adds an implementation to show a store element using the command `show`. It implements the store API functions `can_show` and `show`.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `extension`: Default extension (for temporary filenames, without dot, e.g. `"svg"`)
- `os`: Output stream
- `element`: Reference to the store element that should be shown

ALICE_STORE_HTML (type, element)
Generates HTML output for a store element.

This macro is only needed when the shell is used as a Python module inside an environment such as Jupyter notebooks. Then a specialized output can be configured for a store element when calling the `print` method on it. It implements the store API functions `has_html_repr` and `html_repr`.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `element`: Reference to the current store element

9.2 Commands

ALICE_ADD_COMMAND (name, category)
Add a command.

This macro adds a command to the shell interface. When this macro is called, a class of name `<name>_command` must have been defined that inherits from *`alice::command`* or some of its subclasses.

The command is accessible from the shell interface using `name`. In Python mode, the module will contain a function `name`.

Parameters

- `name`: Name of the command
- `category`: Category of the command (as shown in `help`)

ALICE_COMMAND (name, category, description)
Add and implements a simple command.

Unline `ALICE_ADD_COMMAND`, this macro can be used to also implement a simple command. However, it allows only to implement the code of the execute function, and therefore no customization of command arguments, validators, and logging is possible.

The macro must be followed by a code block.

Parameters

- `name`: Name of the command
- `category`: Category of the command (as shown in `help`)
- `description`: Short description of the command (as shown in `help`)

ALICE_READ_FILE (type, tag, filename, cmd)
Read from a file into a store.

This macro adds an implementation for reading from a file into a store. Different file types may be supported, which are indexed using the tag.

The macro must be followed by a code block.

Parameters

- `type`: Store type

- `tag`: File tag
- `filename`: Filename
- `cmd`: Reference to the command line interface of the command

`ALICE_WRITE_FILE`(type, tag, element, filename, cmd)

Write to a file from a store.

This macro adds an implementation for writing to a file from a store. Different file types may be supported, which are indexed using the tag.

The macro must be followed by a code block.

Parameters

- `type`: Store type
- `tag`: File tag
- `element`: Reference to the store element
- `filename`: Filename
- `cmd`: Reference to the command line interface of the command

`ALICE_ADD_FILE_TYPE`(tag, name)

Registers a file type to alice.

Calling this macro will mainly cause the addition of two commands `read_<tag>` and `write_<tag>` to alice to read from files and write to files. The actual implementation is done using `ALICE_READ_FILE` and `ALICE_WRITE_FILE` which will also associate store types to file tags.

Parameters

- `tag`: File tag
- `name`: Name that is used for help strings

`ALICE_ADD_FILE_TYPE_READ_ONLY`(tag, name)

Registers a read-only file type to alice.

Like `ALICE_ADD_FILE_TYPE` but only adds `read_<tag>`.

Parameters

- `tag`: File tag
- `name`: Name that is used for help strings

`ALICE_ADD_FILE_TYPE_WRITE_ONLY`(tag, name)

Registers a write-only file type to alice.

Like `ALICE_ADD_FILE_TYPE` but only adds `write_<tag>`.

Parameters

- `tag`: File tag
- `name`: Name that is used for help strings

CHAPTER 10

Store API

Function	Description
to_string	Produce short one-line description of store element.
print	Routine to print a store element to an output stream.
print_statistics	Routine to print statistics of a store element to an output stream.
log_statistics	Statistics to log when calling
can_read	Controls whether a store entry can read from a specific format.
read	Reads from a format and returns store element.
can_write	Controls whether a store entry can write to a specific format.
write	Writes store element to a file format.
write	Writes store element to log file.
can_convert	Controls whether a store entry can be converted to an entry of a different store type.
convert	Converts a store entry into an entry of a different store type.
can_show	Controls whether a store element can be visualized.
show	Generates the file to visualize a store element.
has_html_repr	Controls whether store element has specialized HTML output.
html_repr	Returns an HTML representation for a store element in Python mode.

10.1 Declaring a new store type

template <typename StoreType>

struct store_info

Empty prototype class for store information.

You need to specialize this struct in order declare a new store type for the CLI. In this specialization five `static constexpr const char*` variables must be defined:

- `key`: A unique key for internal storing in the `environment`
- `option`: A long option name for commands (without dashes)
- `mnemonic`: A single character for short option (without dash, cannot be `n` or `v`)
- `name`: A singular name that is used in help texts
- `name_plural`: A plural name that is used in help texts

Note: Make sure to specialize `store_info` inside the `alice` namespace. You can use the *`ALICE_ADD_STORE`* macro instead of the partial specialization. Also *`ALICE_MAIN`* will automatically pick up all stores that were defined using *`ALICE_ADD_STORE`*.

Here is an example code to define a store type for a fictional type `graph`:

```
namespace alice {

template<>
struct store_info<graph>
{
  static constexpr const char* key = "graph";
  static constexpr const char* option = "graph";
  static constexpr const char* mnemonic = "g";
  static constexpr const char* name = "graph";
  static constexpr const char* name = "graphs";
};

}
```

You can then use this data structure as part of the CLI when listing its type in the declaration of `cli`:

```
alice::cli<..., graph, ...> cli( "prefix" );
```

10.2 Customizing store functions

template <typename StoreType>
std::string `alice`**::to_string**(StoreType **const** &*element*)

Produce short one-line description of store element.

You can use *`ALICE_DESCRIBE_STORE`* to implement this function.

element Store element

template <typename StoreType>
void `alice`**::print**(std::ostream &*out*, StoreType **const** &*element*)

Routine to print a store element to an output stream.

This routine is called by the *print* command. You can use *`ALICE_PRINT_STORE`* to implement this function.

Parameters

- `out`: Output stream
- `element`: Store element

template <typename StoreType>
void `alice::`**print_statistics**(std::ostream &*out*, StoreType **const** &*element*)

Routine to print statistics of a store element to an output stream.

This routine is called by the `ps` command.

Parameters

- `out`: Output stream
- `element`: Store element

template <typename StoreType>
nlohmann::json `alice::`**log_statistics**(StoreType **const** &*element*)

Statistics to log when calling `ps`

This routine is called by the *ps* command, if logging is enabled.

Parameters

- `element`: Store element

template <typename StoreType, **typename** Tag>
bool `alice::`**can_read**(*command* &*cmd*)

Controls whether a store entry can read from a specific format.

If this function is overriden to return true, then also the function `read` must be impemented for the same store element type and format tag.

You can use *`ALICE_READ_FILE`* to implement this function together with `alice::read`. However, if you need custom command line arguments, the macro cannot be used and one needs to specialize using these functions as described by the following example.

```
template<>
bool can_read<std::string>( command& cmd )
{
  cmd.add_flag( "--flag", "some flag" );
  cmd.add_option<std::string>( "--option", "an option stored in a string" );
}

template<>
std::string read<std::string>( const std::string& filename, const command& cmd )
{
  auto flag = cmd.is_set( "flag" );
  auto option = cmd.option_value<std::string>( "option" );

  // read the file and return a string...
}
```

Parameters

- `cmd`: Mutable reference to command, e.g., to add custom options

template <typename StoreType, **typename** Tag>

StoreType `alice::`**`read`**`(`**`const`** std::string &*filename*, **`const`** *command* &*cmd*`)`

Reads from a format and returns store element.

This function must be enabled by overriding the `can_read` function for the same store element type and format tag. See `can_read` for more details and an example.

The `read` function may throw an exception. In this case, no new element is added to the store. Anything can be thrown, but if a `std::string` is thrown, this string is used to output an error message to the error stream.

Parameters

- `filename`: Filename to read from
- `cmd`: Reference to command, e.g., to check whether custom options are set

`template` <**`typename`** StoreType, **`typename`** Tag>
bool `alice::`**`can_write`**`(`*command* &*cmd*`)`

Controls whether a store entry can write to a specific format.

If this function is overriden to return true, then also the function `write` must be impemented for the same store element type and format tag. See `can_read` for an example which can easily be adapted for `can_write` and `write`.

Parameters

- `cmd`: Mutable reference to command, e.g., to add custom options

`template` <**`typename`** StoreType, **`typename`** Tag>
void `alice::`**`write`**`(`StoreType **`const`** &*element*, **`const`** std::string &*filename*, **`const`** *command* &*cmd*`)`

Writes store element to a file format.

This function must be enabled by overriding the `can_write` function for the same store element type and format tag.

Parameters

- `element`: Store element to write
- `filename`: Filename to write to
- `cmd`: Reference to command, e.g., to check whether custom options are set

`template` <**`typename`** StoreType, **`typename`** Tag>
void `alice::`**`write`**`(`StoreType **`const`** &*element*, std::ostream &*os*, **`const`** *command* &*cmd*`)`

Writes store element to log file.

This function should be enabled by overriding the `can_write` function for the same store element type and format tag.

Parameters

- `element`: Store element to write
- `os`: Output stream to write to
- `cmd`: Reference to command, e.g., to check whether custom options are set

`template` <**`typename`** SourceStoreType, **`typename`** DestStoreType>

bool `alice::`**`can_convert`**`()`

Controls whether a store entry can be converted to an entry of a different store type.

If this function is overriden to return true, then also the function `convert` must be implemented for the same store types.

You can use *`ALICE_CONVERT`* to implement this function together with `convert`.

`template` **<`typename`** SourceStoreType, **`typename`** DestStoreType>
DestStoreType `alice::`**`convert`**`(`SourceStoreType **`const`** *&element*`)`

Converts a store entry into an entry of a different store type.

This function must be enabled by overriding the `can_convert` function for the same store element types.

You can use *`ALICE_CONVERT`* to implement this function together with `can_convert`.

Return Converted store element

Parameters

- `element`: Store element to convert

`template` **<`typename`** StoreType>
bool `alice::`**`can_show`**`(`std::string *&extension*, *command* *&cmd*`)`

Controls whether a store element can be visualized.

If this function is overriden to be true, then also the function `show` mustbe implement for the same store type. The command `show` allows to visualize a store element. For this purpose a text file is written containing the visual representation, e.g., in terms of DOT or SVG (but other formats are possible).

When implementing this function, it should return true and assign `extension` a default file extension for the representation format, which will be used to name temporary files.

Parameters

- `extension`: Default extension, without the dot (e.g., `"svg"`)
- `cmd`: Mutable reference to command, e.g., to add custom options

`template` **<`typename`** StoreType>
void `alice::`**`show`**`(`std::ostream *&out*, StoreType **`const`** *&element*, **`const`** *command* *&cmd*`)`

Generates the file to visualize a store element.

This function is function can be enabled by overriding the `can_show` function to return true. It takes as parameter an output stream `out` to a file that is shown using a program by the `show` command.

Parameters

- `out`: Output stream
- `element`: Store element to show
- `cmd`: Reference to command, e.g., to check whether custom options are set

`template` **<`typename`** StoreType>
bool `alice::`**`has_html_repr`**`()`

Controls whether store element has specialized HTML output.

This function can be used to customize the output of the `print` in Python mode, when being used in Jupyter notebooks. If this function returns true, the output of `html_repr` is used to create HTML output for a store element.

Works only in Python mode.

template <typename StoreType>
std::string `alice::`**`html_repr`**(StoreType **const** &*element*)

Returns an HTML representation for a store element in Python mode.

This method enables to return a specialized HTML output for a store element when calling `print` as a function in Python mode. This output can be used in environments such as Jupyter notebook.

Works only in Python mode.

Parameters

- `element`: Store element

CHAPTER 11

CLI

Method	Description
cli	Default constructor.
set_category	Sets the current category.
insert_command	Inserts a command.
insert_read_command	Inserts a read command.
insert_write_command	Inserts a write command.
run	Runs the shell.

`template` <class... *S*>
`class cli`

CLI main class.

The stores of a CLI are passed as type arguments to `cli`. For example, if the CLI has stores for Graphs and Trees which are handled by classes `graph` and `tree`, respectively, the class instantiation is `cli<graph, tree>`.

Public Functions

`cli` (**`const`** std::string &*prefix*)

Default constructor.

Initializes the CLI with a prefix that is used as a command prefix in stand-alone application mode and as a module name when build as Python module.

The constructor will add the default commands to the CLI. If no store type is specified, then no store-related command will be added.

Parameters

- `prefix`: Either command prefix or module name (depending on build mode)

void **`set_category`**(**`const`** std::string &*_category*)

Sets the current category.

This category will be used as category for all commands that are added afterwards, until this method is called again with a different argument.

The categories are used in the `help` command to organize the commands.

The macros :c:macro:`ALICE_COMMAND` and :c:macro:`ALICE_ADD_COMMAND` will automatically call this method.

Parameters

- `_category`: Category name

void **`insert_command`**(**`const`** std::string &*name*, **`const`** std::shared_ptr<*command*> &*cmd*)

Inserts a command.

Inserts a command (as a shared pointer) to the CLI.

The macro :c:macro:`ALICE_ADD_COMMAND` will automatically call this method with a convention that a command with name `<name>` must be called `<name>_command`.

Parameters

- `name`: Name of the command
- `cmd`: Shared pointer to a command instance

`template <typename` Tag>
void **`insert_read_command`**(**`const`** std::string &*name*, **`const`** std::string &*label*)

Inserts a read command.

Inserts a read command for a given file tag. The name of the command can be arbitrary but the default convention is to prefix it with `read_`. The macro :c:macro:`ALICE_ADD_FILE_TYPE` together :c:macro:`ALICE_READ_FILE` will automatically add a read command called `read_<tagname>`.

Parameters

- `name`: Name of the command
- `label`: Label for the file type (used in help string)

`template <typename` Tag>
void **`insert_write_command`**(**`const`** std::string &*name*, **`const`** std::string &*label*)

Inserts a write command.

Inserts a writ command for a given file tag. The name of the command can be arbitrary but the default convention is to prefix it with `write_`. The macro :c:macro:`ALICE_ADD_FILE_TYPE` together :c:macro:`ALICE_WRITE_FILE` will automatically add a write command called `write_<tagname>`.

Parameters

- `name`: Name of the command
- `label`: Label for the file type (used in help string)

int **`run`**(int *argc*, char ***argv*)

Runs the shell.

This function is only used if the CLI is used in stand-alone mode, not when used as Python module. The values `argc` and `argv` can be taken from the `main` function. For some flags, such as `-f` and `-c`, the CLI will read commands from a file or the command line, respectively, and then stop (unless flag `-i` is set). Otherwise, the CLI will enter a loop that accepts commands as user inputer.

Parameters

- `argc`: Number of arguments (incl. program name, like `argc` in `main`)
- `argv`: Argument values (like `argv` in `main`)

CHAPTER 12

Command

Method	Description
command	Default constructor.
validity_rules	Returns rules to check validity of command line arguments.
execute	Executes the command.
log	Returns logging data.
caption	Returns command short description.
add_flag	Adds a flag to the command.
add_flag	Adds a flag with variable binding to the command.
add_option	Adds an option to the command.
add_option	Adds an anonymous option to the command.
option_value	Returns the value for an anonymous option.
is_set	Checks whether an option was set when calling the command.
store	Returns a store.

`class command`
Command base class.

Public Types

`using rule` = std::pair<std::function<bool ( ) >, std::string>
Rule.

A rule consists of a nullary predicate (validator) and a string (error message). The validator should return `true` in the correct case.

using rules = std::vector<*rule*>
Rules.

Vector of rules.

Public Functions

command(**const** *environment*::*ptr* &*env*, **const** std::string &*caption*)
Default constructor.

The shell environment that is passed as the first argument should be the one from the *`alice::cli`* instance. Typically, commands are constructed and added using the macro API, e.g., `ALICE_COMMAND` or `ALICE_ADD_COMMAND`.

Parameters

- `env`: Shell environment
- `caption`: Short (one-line) description of the command

const auto &**caption**() **const**
Returns command short description.

auto **add_flag**(**const** std::string &*name*, **const** std::string &*description*)
Adds a flag to the command.

This function should be called in the constructor when the program options are set up. See https://github.com/CLIUtils/CLI11#adding-options for more information.

This is a shortcut to `opts.add_flag`.

Return Option instance

Parameters

- `name`: Flag names (short flags are prefixed with a single dash, long flags with a double dash), multiple flag names are separated by a comma.
- `description`: Description for the help text

auto **add_flag**(**const** std::string &*name*, bool &*value*, **const** std::string &*description*)
Adds a flag with variable binding to the command.

This function should be called in the constructor when the program options are set up. See https://github.com/CLIUtils/CLI11#adding-options for more information.

This is a shortcut to `opts.add_flag`.

Return Option instance

Parameters

- `name`: Flag names (short flags are prefixed with a single dash, long flags with a double dash), multiple flag names are separated by a comma.
- `value`: Reference where flag value is stored

- `description`: Description for the help text

template <typename T>
auto **add_option**(**const** std::string &*name*, T &*value*, **const** std::string &*description*, bool *defaulted* = false)

Adds an option to the command.

This function should be called in the constructor when the program options are set up. See https://github.com/CLIUtils/CLI11#adding-options for more information.

This is a shortcut to `opts.add_option`.

Return Option instance

Parameters

- `name`: Option names (short options are prefixed with a single dash, long options with a double dash, positional options without any dash), multiple option names are separated by a comma.
- `value`: Reference where option value is stored
- `description`: Description for the help text
- `defaulted`: Use initial value to `value` as default value

template <typename T = std::string>
auto **add_option**(**const** std::string &*name*, **const** std::string &*description*)

Adds an anonymous option to the command.

Unlike the other method, this method adds an option, but takes the value reference from the command itself. This is especially helpful when using the store API, e.g., `can_read` together with `read`, where command line options are setup in one function but used in another.

Use a type as template argument to specify the type of the option value and use `option_value` to return the option value using any of the option names (incl. possible dashes).

Return Option instance

Parameters

- `name`: Option names (short options are prefixed with a single dash, long options with a double dash, positional options without any dash), multiple option names are separated by a comma.
- `description`: Description for the help text

template <typename T = std::string>
T **option_value**(**const** std::string &*name*, **const** T &*default_value* = T()) **const**

Returns the value for an anonymous option.

Use any of the option names to access a value. For example, if the option names were `"--option,-o"`, then one can use both `"--option"` and `"-o"` as value for the `name` parameter. It is important that the same type is used to retrieve the option value that was used to add the anonymous option. If the option name does not point to an anonymous option, a default value is returned.

Return Option value

Parameters

- `name`: One of the option names that was used to create the option
- `default_value`: Dafault value, if name does not point to anonymous option

bool **is_set**(**const** std::string &*name*) **const**

Checks whether an option was set when calling the command.

Any of the option names can be passed, with our without dashes.

Parameters

- `name`: Option name

template <typename T>
store_container<T> &**store**() **const**

Returns a store.

Short cut for `env->store<T>()`.

Protected Functions

virtual *rules* **validity_rules**() **const**

Returns rules to check validity of command line arguments.

This returns a vector of `rule` objects (`rules`), which are pairs of a nullary predicate (Function with `bool` return value and no arguments) and an error message as string. For each pair, in order of their position in the vector, the predicate is evaluated. If any of the predicates evalutes to `false`, the command will not be executed and the error message will be printed.

By default, an empty vector is returned.

The following code checks that a store element is present for `std::string` and that not both flags `-a` and `-b` are set at the same time. For the first check, a predefined rule can be used. Note also that the CLI11 interface allows to put several checks on single options (see https://github.com/CLIUtils/CLI11#adding-options).

```
command::rules example_command::validity_rules() const
{
  return {
    has_store_element<std::string>( env ),
    {
      [this]() { return !( is_set( "a") && is_set( "b") ); },
      "not both -a and -b can be set"
    }
  };
}
```

virtual void **execute**() = 0

Executes the command.

This function must be implemented and contains the main routine that the command executes. At this point all options have been parsed and the corresponding variables are assigned values. Also all validity checks have been made.

virtual nlohmann::json **log**() **const**

Returns logging data.

Logging data is returned in terms of a JSON object using the JSON API from https://github.com/nlohmann/json. This object can be nested, i.e., some keys can map to other objects or arrays.

```
nlohmann::json example_command::log() const
{
  return nlohmann::json({
    {"number", 42},
    {"float", 9.81},
    {"string", "alice"}
  });
}
```

CHAPTER 13

Environment

Method	Description
store	Retrieves store from environment.
has_store	Checks whether environment has store for some data type.
out	Retreives standard output stream.
err	Retreives standard error stream.
reroute	Changes output and error streams.
commands	Returns map of commands.
categories	Returns map of categories.
aliases	Returns a map of aliases.
set_default_option	Sets default store option.
default_option	Returns the current default store option.
has_default_option	Checks whether a default store option is enabled and set.
is_default_option	Checks whether option is default store option.

`class environment`

Shell environment.

The environment gives access to shell related properties, e.g., the commands and stores.

Public Types

`using ptr` = std::shared_ptr<*environment*>

Smart pointer alias for environment.

Public Functions

template <typename T>
store_container<T> &**store**() **const**

Retrieves store from environment.

The store can be accessed using its type.

template <typename T>
bool **has_store**() **const**

Checks whether environment has store for some data type.

Stores are defined by their type.

std::ostream &**out**() **const**

Retreives standard output stream.

This method returns a reference to the current standard output stream. In stand-alone application mode, this is `std::cout` by default, but can be changed. Users should aim for not printing to `std::cout` directly in a command, but use `env->out()` instead.

std::ostream &**err**() **const**

Retreives standard error stream.

This method returns a reference to the current standard error stream. In stand-alone application mode, this is `std::cerr` by default, but can be changed. Users should aim for not printing to `std::cerr` directly in a command, but use `env->err()` instead.

void **reroute**(std::ostream &*new_out*, std::ostream &*new_err*)

Changes output and error streams.

This method allows to change the output streams which are returned by `out()` and `err()`.

const std::unordered_map<std::string, std::shared_ptr<*command*>> &**commands**() **const**

Returns map of commands.

The keys correspond to the command names in the shell.

const std::unordered_map<std::string, std::vector<std::string>> &**categories**() **const**

Returns map of categories.

Keys are catgory names pointing to a vector of command names that can be used to index into `commands()`.

const std::unordered_map<std::string, std::string> &**aliases**() **const**

Returns a map of aliases.

Keys are the alias regular expressions mapping to substitutions.

const std::string &**variable**(**const** std::string &*key*, **const** std::string &*default_value* = std::string()) **const**

Get environment variable.

Finds an environment variable or returns a default value. Variables can be set with the `set` command.

Parameters

- `key`: Key for the value
- `default_value`: Default value

void **set_default_option**(**const** std::string &*default_option*)

Sets default store option.

The environment can keep track of a default store option that can be changed after every command. For example, if one has a store for strings (accessed via option `--str`) and one for numbers (access via option `--int`), then a call to `read_text --str file` would set the default option to `--str`, such that a immediate call to `print` would not need the `--str` option to print the string. The default store option is displayed in the prompt.

This behavior needs to be enabled by defining the macro `ALICE_SETTINGS_WITH_DEFAULT_OPTION` to `true`, before the `alice.hpp` is included.

Parameters

- `default_option`: Updates default store option for next commands

const std::string &**default_option**() **const**

Returns the current default store option.

bool **has_default_option**() **const**

Checks whether a default store option is enabled and set.

bool **is_default_option**(**const** std::string &*option*) **const**

Checks whether option is default store option.

This method also checks whether default store options are enabled. If not, this method always returns `false`.

Parameters

- `option`: Option argument to check (fill name without dashes)

CHAPTER 14

Settings

Global settings for alice can be configured by defining macros before any alice header is included. The following settings are supported:

`ALICE_SETTINGS_WITH_DEFAULT_OPTION`

Controls whether default store options are supported.

Default store options are useful, when a shell interface contains several stores. If one command uses a store, e.g., to read a file, it can remember the last store, such that a following command does not necessarily need to be provided with a flag to select the store again. For example, if one has a store for strings (accessed via option `--str`) and one for numbers (access via option `--int`), then a call to `read_text --str file` would set the default option to `--str`, such that a immediate call to `print` would not need the `--str` option to print the string. The default store option is displayed in the prompt.

The default value for this setting is `false`.

CHAPTER 15

Stores

Method	Description
store_container	Default constructor.
current	Retrieve mutable reference to current store item.
current	Retrieve constant reference to current store item.
*operator**	Retrieve mutable reference to current store item.
*operator**	Retrieve constant reference to current store item.
operator[]	Retreive mutable reference at given index.
operator[]	Retreive const reference at given index.
empty	Returns whether store is empty.
size	Returns the number of elements in the store.
data	Constant access to store elements.
current_index	Returns the current index in the store.
extend	Extend the store by one element and update current element.
pop_current	Removes current store element.
clear	Clears all elements in the store.

`template` `<`**`class`** `T>`
`class store_container`

Store container.

Public Functions

store_container(**const** std::string &*name*)
Default constructor.

Parameters

- `name`: Store name

T &**current**()
Retrieve mutable reference to current store item.

const T &**current**() **const**
Retrieve constant reference to current store item.

T &**operator***()
Retrieve mutable reference to current store item.

const T &**operator***() **const**
Retrieve constant reference to current store item.

T &**operator[]**(std::size_t *index*)
Retreive mutable reference at given index.

Parameters

- `index`: Index

const T &**operator[]**(std::size_t *index*) **const**
Retreive const reference at given index.

Parameters

- `index`: Index

bool **empty**() **const**
Returns whether store is empty.

auto **size**() **const**
Returns the number of elements in the store.

const std::vector<T> &**data**() **const**
Constant access to store elements.

int **current_index**() **const**
Returns the current index in the store.

T &**extend**()
Extend the store by one element and update current element.

The current element is set to the added store element.

void **pop_current**()
Removes current store element.

If the current element is the only element, the store is empty after this operation. If the current element is last element, the store points to the next last-but-one element after this operation. Otherwise, the element after the operation will be the one at the same position.

void **clear**()

Clears all elements in the store.

CHAPTER 16

Validators

Function	Description
ExistingFileWordExp	Checks whether file exists (after expansion)

std::string `alice::`**`ExistingFileWordExp`**`(`**`const`** std::string &*filename*`)`

Checks whether file exists (after expansion)

Before checking whether the file exists, tilde characters for home directories and environment variables are expanded (using `wordexp`).

In Windows this method is just forwarding to `CLI::EistingFile` and does not perform any substitution.

Camel case convention for this function is intentional to match the convention of CLI11, since this function is used when declaring CLI options.

Parameters

- `filename`: Filename

CHAPTER 17

Indices and tables

- genindex
- search

Index