Programming In C++ Documentation

Release 0.0.3

Ash Matheson

Jul 27, 2017

Contents:

An introductory tutorial/review of the C++ programming language

CHAPTER 1

Goals

We understand that there is an existing level of programming expertise by the reader. You are familiar with other languages like C#, Rust, Java and Python. This tutorial series attempts to broach the topic of "How do I write code in C++" as well as touch on other aspects of the development cycle like debugging, improving the build process, and so on.

Source Materials

Source code, as well as this documentation, can be found in Github in the ProgrammingInCPP repo Feel free to fork, change and send up a Pull request with changes/additions. This will always be a 'work in progress' as long as there is interest.

The source code will either be made available in Visual Studio 2015 (or later). You can get the Community edition of the IDE from Microsoft at Visual Studio Community as well as on other web based compilers like C++ Shell or the Compiler Explorer

Syllabus

Review

- Setting up the your build environment.
- Simple language tutorial.
  - Language Basics.
  - Gotchas.
- The process of actually building a console application.
  - Different build types (Debug, Release, Release with Debug symbols).

Memory

- Pointer definition
- Stack vs Heap allocations
- Layout of classes in memory
- Finding memory leaks

Intermediate C++

- Templates
- Serialization
- Lambdas
- RTTI/Exceptions
- New data types (auto)
- Threading
- Some Libraries
- More stuff as I find it.
  - STL and STL-like libraries
  - Header only vs. Static Libs vs. DLLs

Review 01 - Where we talk about the Generating code

Summary

For a lot of folks coming in from other, more modern languages, the process of creating an executable, library or DLL in the C++ ecosystem is a little foreign in the days of JIT compilers and scripting languages. It does have it's perils as it can be a slow build process on larger codebases (compared to C#) but there is an elegance about it.

Take, for instance, the following C++ code

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
}
```

You can take that code, verbatim, and fire it into the C++ shell here C++ Shell and you see the output in the execution window.

But that is obfuscating a lot of what's happening under the hood.

1. First off, the compiler takes a look at the source code. It generates an *.obj* file which is an 'intermeditate' file. You can think of it as 'mostly' machine code, but it has additional information associated with it that is used by the next step, the Linker.
2. The linker takes a set of *.obj* files, as well as any library files, and combines them into a final, resultant file. This final file can be an executable, another library, or a DLL.

Now, that is an incredibly simple overview of what's going on. Let's actually talk about what's really happening there. Let's assume a more complex project, that contains a set of source files (a large set of *.cpp* files)

The C++ Preprocessor

The C++ preprocessor takes the source file and expands all the *preprocessor* directives in your code. The simplest definition of a preprocessor directive is - take a look at any line that starts with a # - the operation that follows that is expanded inline in the code base. So, for the *#include <stdio.h>* line, the contents of the file *stdio.h* are essentially copied and pasted into the source file.

Now, the C++ preprocessor can do a *lot* of things. We'll talk about that later, because it's incredibly important. Just be aware that it's one of the foundational pieces of the C++ language.

Compiler

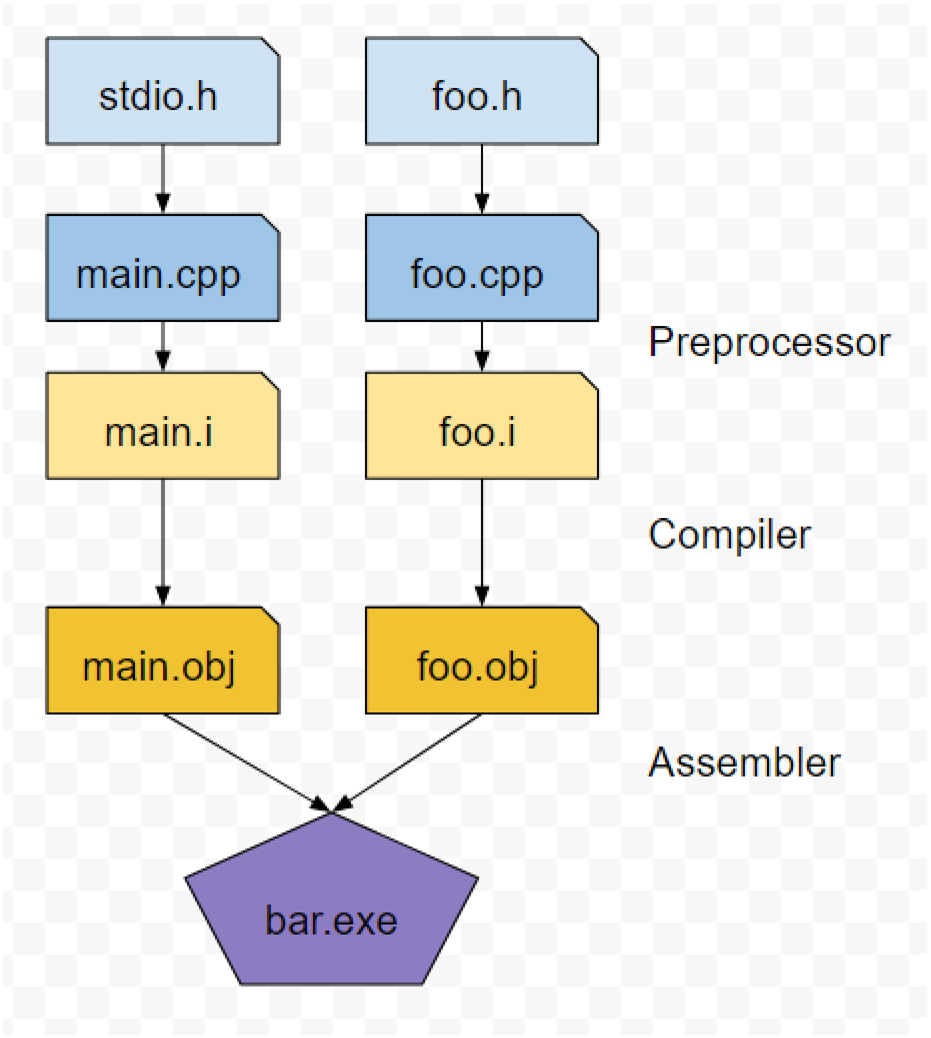
Each expanded source file is then compiled into the assembly language/machine code for the desired platform as an '.obj' file. A compiler can generate assembly language for multiple architectures, if it's capable of doing so. The compiler will also inject additional information about functions, classes and a few other bits, for the next stage.

Linker

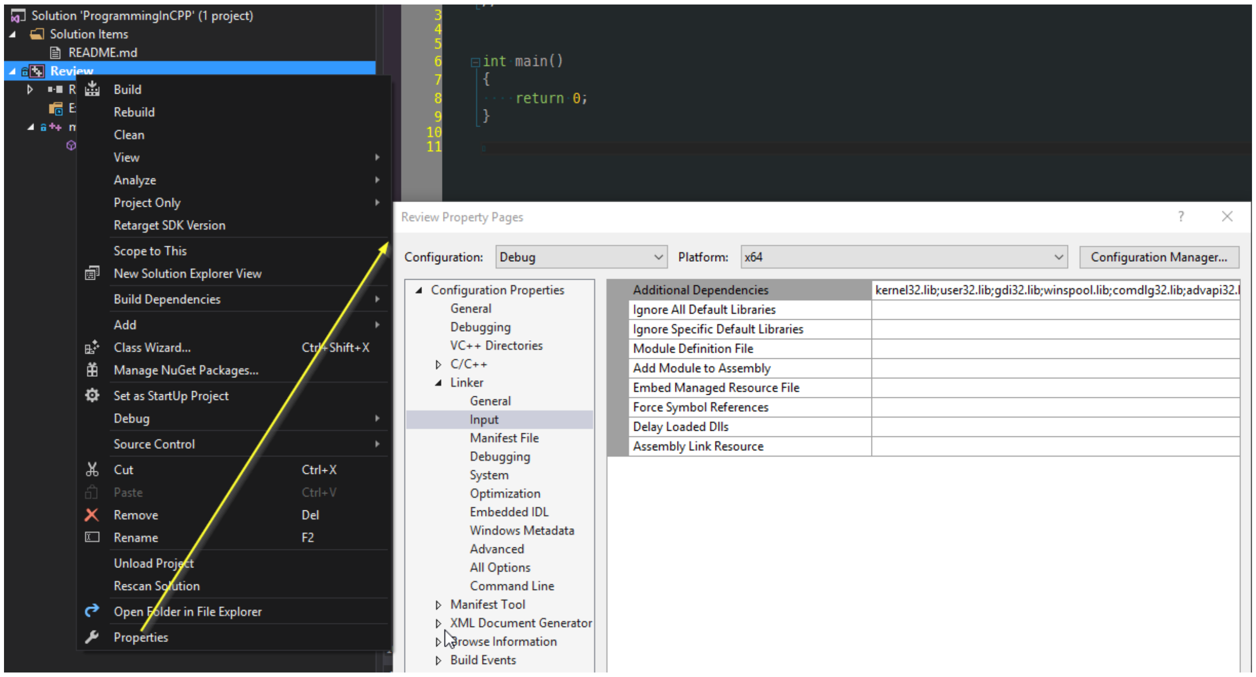
Now that we have individual *.obj* files for each source file, the linker now takes all the individual pieces and Links them together into one piece. This can be an executable, a library, or a DLL.

The other thing to note is that the Linker will also link in other 'things' from external libraries. Remember in the example 'hello world' source, we call the function *printf("Hello Worldn");*? Well, *printf* isn't a function that's native to the C++ language. It's defined in an external Library and it's function signature is defined through *stdio.h*.

Visually, it kind of looks like this process:



The next question is, how does the compiler know how to find the actual source code for the function *printf*? Trust me (or go look yourself), it's not declared in *stdio.h*. We know that the linker will eventually resolve it into a library, but how do we tell the compiler what library to use? We're lucky in Visual Studio, there's a nice dialog for that:



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

Configuration: Debug   Platform: x64

Additional Dependencies	kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32.lib;advapi32.l
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

In other build environments, those compiler options can be passed through on the command line, through a make file, or another build system (I expect XCode does something similar to Visual Studio).

More details

I'm being purposely obtuse about the format of the files the compiler/linker generates. I feel that's outside of the scope of this tutorial. However, it is something that is worth noting as there are multiple formats depending on the Operating System (ELF, COFF, PE32+ for example). I'll point you here to start the investigation if you're truly interested: wikipedia

Examining an incredibly simple program

In the source code for this tutorial, I have a very simple example program that consists of multiple files.

- *main.cpp* = This is the main entry point into our application
- *functions.h* = This is the definitions of functions we are exposing
- *functions.cpp* = this is the declaration of the functions we are using.

There's a bit there, so let's walk through this:

We have the main entry point in `main.cpp` via the `int main()` function declaration. In this we call a few external functions. Most notably, we call our own function, `Fibbonaci` to calculate a Fibbonaci series recursively. The other functions are part of the standard library (not the STL).

```
// ProgrammingInCPP.cpp : Defines the entry point for the console application.
//

#include <stdio.h>
#include "Functions.h"

int main()
```

```
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }

    printf("press any key to continue");
    scanf("-");
    return 0;
}
```

Also note that we include the `Functions.h` header file. To be 100% explicit, this has the *exact* same functionality of injecting the contents of the file into `main.cpp` like so:

```
// ProgrammingInCPP.cpp : Defines the entry point for the console application.
//

#include <stdio.h>
#pragma once

unsigned int Fibbonaci(unsigned int i);

int main()
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }

    printf("press any key to continue");
    scanf("-");
    return 0;
}
```

To repeat, you could physically copy the contents of the file `Functions.h` and replace `#include "Functions.h"` with those contents. Go ahead. Try it. See what you get. And then revert it ;)

What, then, is the purpose of the *Functions.h* header file? This is where we declare the signature of a function, class, template, etc that can be referenced elsewhere. To stress this - we are defining the signature of the function `Fibbonacci` in this case. We do not actually define the implementation of that function.

Finally, we have the file `Functions.cpp`. This is where we define the implementation. Also note that I do not have any headers in this file:

```
unsigned int Fibbonaci(unsigned int i)
{
    if (i <= 1)
        return i;

    return Fibbonaci(i - 1) + Fibbonaci(i - 2);
}
```

In this case, I don't need to include any headers as I am not referencing any external functions. Also note that the C++ compiler cannot 'look ahead' to infer functions/classes that are defined later in the file. If you need to reference a function/class, you are going to need to declare it before it's used. This is why you'll see a header file for a `.cpp` file included - it does the *forward declarations* of those functions/classes for you.

OK, what else do we have in this *Fibbonacci* function implementation?

- we have simple return types defined
- we have an example of a simple argument passed into the function
- we have a conditional in the form of an *if* statement
- we have an example of recursion, where the function calls itself.

I don't think I need to review how recursion works. If I'm wrong, please let me know.

To Summarize

This is a pretty quick tutorial. I've covered some fairly straightforward concepts here. In the next example, we'll actually discuss the language basics.

What we haven't reviewed

I'm leaving it to the reader to understand how to compile the project. This is a Visual Studio 2015 solution/project. Visual Studio 2015 Community Edition was used in the development of this example project.

Enjoy for now.

Review 02 - Where we talk about some of the tools

Overview

In this section, we're going to dig further into building C++ code and less so about the language itself. To do that, we're going to be using the command line a fair bit. If you've installed Visual Studio 2015 (Community) or Visual Studio 2017 you should be good.

Additionally, I'll be looking at using clang as an alternative, so you can see how it works in a non-Visual Studio environment. You can find clang (and llvm) here.

Checking your configuration.

You should fire off the 'VS2015 x86 Native Tools Command Prompt' from your start menu for this. You'll know it's working if you can do this:

```
D:\ProgrammingInCPP\Review02>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

D:\ProgrammingInCPP\Review02>
```

For clarity, we'll call this window/toolset the `cl` compiler.

The same thing goes for clang:

```
D:\ProgrammingInCPP\Review02> clang -v
clang version 4.0.1 (tags/RELEASE_401/final)
Target: x86_64-pc-windows-msvc
Thread model: posix
InstalledDir: C:\Program Files\LLVM\bin
```

At this point, you should be good. If not, check out the faq's for each.

For clarity, we'll call this window/toolset the `clang` compiler.

What is `clang`?

Wiki defines it as

> a compiler front end for the programming languages C, C++, Objective-C, Objective-C++, OpenMP,[5] OpenCL, and CUDA. It uses LLVM as its back end and has been part of the LLVM release cycle since LLVM 2.6.
>
> It is designed to be able to replace the full GNU Compiler Collection (GCC). Its contributors include Apple, Microsoft, Google, ARM, Sony, Intel and Advanced Micro Devices (AMD). It is open-source software,[6] with source code released under the University of Illinois/NCSA License, a permissive free software licence.

From this, we see the phrase `LLVM`. What is that? Again, Wiki to the rescue Wiki LLVM

> The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and toolchain technologies"[3] used to develop compiler front ends and back ends.
>
> LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala,[9] Swift, and Xojo.
>
> The name LLVM was originally an initialism for Low Level Virtual Machine, but this became increasingly less apt as LLVM became an "umbrella project" that included a variety of other compiler and low-level tool technologies, so the project abandoned the initialism.[13] Now, LLVM is a brand that applies to the LLVM umbrella project, the LLVM intermediate representation (IR), the LLVM debugger, the LLVM C++ Standard Library (with full support of C++11 and C++14), etc. LLVM is administered by the LLVM Foundation. Its president is compiler engineer Tanya Lattner.

I don't know if I can describe it better. So I won't try! If you're looking for more infor on `clang` and `LLVM`, go to the source right here and also here

Compiling your program, redux

Let's take the example from the previous tutorial, but this time as one file.

```
// Example01.cpp
#include <stdio.h>

unsigned int Fibbonaci(unsigned int i)
{
    if (i <= 1)
        return i;

    return Fibbonaci(i - 1) + Fibbonaci(i - 2);
}
```

```
int main()
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }

    printf("press any key to continue");
    scanf("-");
    return 0;
}
```

This program works exactly the same as our previous example, but it's all contained in one file.

Using the Microsoft compiler

To compile this into an executable using `cl`, we'd do the following in the source folder:

```
D:\dev\bagofholding\ProgrammingInCPP\Review02>dir
Volume in drive D is Data
Volume Serial Number is AEB0-7600

Directory of D:\dev\bagofholding\ProgrammingInCPP\Review02

07/13/2017  12:59 PM    <DIR>          .
07/13/2017  12:59 PM    <DIR>          ..
07/13/2017  10:29 AM    <DIR>          Docs
07/13/2017  12:52 PM               398 Example01.cpp
07/13/2017  12:39 PM             1,009 README.md
            2 File(s)          1,407 bytes
            3 Dir(s)  1,841,761,034,240 bytes free

D:\dev\bagofholding\ProgrammingInCPP\Review02>cl Example01.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example01.cpp
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Example01.exe
Example01.obj

D:\dev\bagofholding\ProgrammingInCPP\Review02>dir
Volume in drive D is Data
Volume Serial Number is AEB0-7600

Directory of D:\dev\bagofholding\ProgrammingInCPP\Review02

07/13/2017  12:59 PM    <DIR>          .
07/13/2017  12:59 PM    <DIR>          ..
07/13/2017  10:29 AM    <DIR>          Docs
07/13/2017  12:52 PM               398 Example01.cpp
07/13/2017  12:59 PM           139,776 Example01.exe
07/13/2017  12:59 PM             4,299 Example01.obj
07/13/2017  12:39 PM             1,009 README.md
```

```
             4 File(s)        145,482 bytes
             3 Dir(s)  1,841,760,882,688 bytes free

D:\dev\bagofholding\ProgrammingInCPP\Review02>Example01.exe
The Fibbonaci series of 0 is 0
The Fibbonaci series of 1 is 1
The Fibbonaci series of 2 is 1
The Fibbonaci series of 3 is 2
The Fibbonaci series of 4 is 3
The Fibbonaci series of 5 is 5
The Fibbonaci series of 6 is 8
The Fibbonaci series of 7 is 13
The Fibbonaci series of 8 is 21
The Fibbonaci series of 9 is 34
press any key to continue

D:\dev\bagofholding\ProgrammingInCPP\Review02>
```

As you can see above, this ends up with the obj and the executable file living in the same folder as the source file. To redirect the files to other folders, you'd use flags to set different compiler options. Here's an example of putting obj files into an `obj` folder and the execuatble in a `bin` folder:

```
D:\dev\bagofholding\ProgrammingInCPP\Review02>mkdir obj
D:\dev\bagofholding\ProgrammingInCPP\Review02>mkdir bin
D:\dev\bagofholding\ProgrammingInCPP\Review02>cl /Fo.\obj\ Example01.cpp /Fe.
↪\bin\Example01.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example01.cpp
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:.\bin\Example01.exe
.\obj\Example01.obj

D:\dev\bagofholding\ProgrammingInCPP\Review02>
```

With `cl`, the `/F` flag represents the "Output File Options" MS Docs

Using clang

The absolute **simplest** way to use `clang` is to invoke `clang-cl`. From clang.llvm.org

> clang-cl is an alternative command-line interface to Clang, designed for compatibility with the Visual C++ compiler, cl.exe.

And thus:

```
D:\dev\bagofholding\ProgrammingInCPP\Review02> mkdir bin

D:\dev\bagofholding\ProgrammingInCPP\Review02> mkdir obj

D:\dev\bagofholding\ProgrammingInCPP\Review02> clang-cl /Fo.\obj\ Example01.cpp /Fe.
↪\bin\Example01.exe
Example01.cpp(19,5):  warning: 'scanf' is deprecated: This function or variable may␣
↪be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_
↪NO_WARNINGS. See online
```

```
    help for details. [-Wdeprecated-declarations]
    scanf("-");
    ^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h(1270,33): ␣
↪note: 'scanf' has been explicitly marked deprecated here
_CRT_STDIO_INLINE int __CRTDECL scanf(
                                ^
1 warning generated.

D:\dev\bagofholding\ProgrammingInCPP\Review02> bin\Example01.exe
The Fibbonaci series of 0 is 0
The Fibbonaci series of 1 is 1
The Fibbonaci series of 2 is 1
The Fibbonaci series of 3 is 2
The Fibbonaci series of 4 is 3
The Fibbonaci series of 5 is 5
The Fibbonaci series of 6 is 8
The Fibbonaci series of 7 is 13
The Fibbonaci series of 8 is 21
The Fibbonaci series of 9 is 34
press any key to continue
```

That said, you can do something **similar** to that using `clang++` (or `clang` if you are looking to only compile C code):

```
D:\dev\bagofholding\ProgrammingInCPP\Review02> clang++ Example01.cpp -o bin/Example01.
↪exe
Example01.cpp:19:5: warning: 'scanf' is deprecated: This function or variable may be␣
↪unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_
↪WARNINGS. See online
    help for details. [-Wdeprecated-declarations]
    scanf("-");
    ^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h:1270:33:␣
↪note: 'scanf' has been explicitly marked deprecated here
_CRT_STDIO_INLINE int __CRTDECL scanf(
                                ^
1 warning generated.
```

Note that this does not generate object files. To generate object files you have to specifcy the `-c` flag to *ONLY* generate object files; you cannot generate an executable unless you run `clang++` again (or a linker) to link the generated object files into a final target file.

Next

Something simple this time around - let's move the `Fibbonaci` function below `main` and see what happens:

```cpp
// Example02.cpp
#include <stdio.h>

int main()
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }
    printf("press any key to continue");
    scanf("-");
    return 0;
}

unsigned int Fibbonaci(unsigned int i)
{
    if (i <= 1)
        return i;

    return Fibbonaci(i - 1) + Fibbonaci(i - 2);
}
```

And the results:

cl

```
D:\dev\bagofholding\ProgrammingInCPP\Review02>cl /Fo.\obj\ Example02.cpp /Fe.
↪\bin\Example02.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example02.cpp
Example02.cpp(7): error C3861: 'Fibbonaci': identifier not found
```

clang

```
D:\dev\bagofholding\ProgrammingInCPP\Review02
> clang++ Example02.cpp -o bin/Example02.exe
Example02.cpp:7:61: error: use of undeclared identifier 'Fibbonaci'
    printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
                                                        ^
Example02.cpp:11:5: warning: 'scanf' is deprecated: This function or variable may be␣
↪unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_
↪WARNINGS. See online
    help for details. [-Wdeprecated-declarations]
scanf("-");
^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h:1270:33:␣
↪note: 'scanf' has been explicitly marked deprecated here
_CRT_STDIO_INLINE int __CRTDECL scanf(
                            ^
1 warning and 1 error generated.
```

If you've ever worked in C/C++ before, this should have been the expected result. You can't invoke a function (or class, type, class, etc) unless it's been either implemented or declared. We can fix this by defining the signature of the function, and only the signature of the function as a `Forward Declaration`, like so:

```
// Example03.cpp
#include <stdio.h>

unsigned int Fibbonaci(unsigned int i);
```

```
int main()
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }

    printf("press any key to continue");
    scanf("-");
    return 0;
}

unsigned int Fibbonaci(unsigned int i)
{
    if (i <= 1)
        return i;

    return Fibbonaci(i - 1) + Fibbonaci(i - 2);
}
```

cl

```
D:\dev\bagofholding\ProgrammingInCPP\Review02>cl /Fo.\obj\ Example03.cpp /Fe.
↪\bin\Example03.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example03.cpp
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:.\bin\Example03.exe
.\obj\Example03.obj
```

clang

```
D:\dev\bagofholding\ProgrammingInCPP\Review02
> clang++ Example03.cpp -o bin/Example03.exe
Example03.cpp:14:5: warning: 'scanf' is deprecated: This function or variable may be␣
↪unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_
↪WARNINGS. See online
    help for details. [-Wdeprecated-declarations]
    scanf("-");
    ^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h:1270:33:␣
↪note: 'scanf' has been explicitly marked deprecated here
_CRT_STDIO_INLINE int __CRTDECL scanf(
                                ^
1 warning generated.
```

Let's now move this into a separate file, in this case, called `Functions.cpp` and compile that.

```
// Example04.cpp
#include <stdio.h>
```

```
unsigned int Fibbonaci(unsigned int i);

int main()
{
    for (unsigned int index = 0; index < 10; index++)
    {
        printf("The Fibbonaci series of %d is %d\n", index, Fibbonaci(index));
    }

    printf("press any key to continue");
    scanf("-");
    return 0;
}
```

```
// Functions.cpp
unsigned int Fibbonaci(unsigned int i)
{
    if (i <= 1)
        return i;

    return Fibbonaci(i - 1) + Fibbonaci(i - 2);
}
```

Trying to compile that ...

cl

```
cl /Fo.\obj\ Example04.cpp /Fe.\bin\Example04.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example04.cpp
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:.\bin\Example04.exe
.\obj\Example04.obj
Example04.obj : error LNK2019: unresolved external symbol "unsigned int __cdecl␣
↪Fibbonaci(unsigned int)" (?Fibbonaci@@YAII@Z) referenced in function main
.\bin\Example04.exe : fatal error LNK1120: 1 unresolved externals
```

clang

```
D:\dev\bagofholding\ProgrammingInCPP\Review02
> clang++ Example04.cpp -o bin/Example04.exe
Example04.cpp:14:5: warning: 'scanf' is deprecated: This function or variable may be␣
↪unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_
↪WARNINGS. See online
    help for details. [-Wdeprecated-declarations]
    scanf("-");
    ^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h:1270:33:␣
↪note: 'scanf' has been explicitly marked deprecated here
```

```
_CRT_STDIO_INLINE int __CRTDECL scanf(
                                ^
1 warning generated.
```

Neither can find the `Fibbonaci` symbol. And they can't find it because the `Functions.cpp` file hasn't been compiled. How do we fix that? Well, we compile the file!

cl

```
D:\dev\bagofholding\ProgrammingInCPP\Review02>cl /Fo.\obj\ Example04.cpp Functions.
↪cpp /Fe.\bin\Example04.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.24215.1 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Example04.cpp
Functions.cpp
Generating Code...
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:.\bin\Example04.exe
.\obj\Example04.obj
.\obj\Functions.obj
```

clang

```
D:\dev\bagofholding\ProgrammingInCPP\Review02
> clang++ Example04.cpp Functions.cpp -o bin/Example04.exe
Example04.cpp:14:5: warning: 'scanf' is deprecated: This function or variable may be␣
↪unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_
↪WARNINGS. See online
    help for details. [-Wdeprecated-declarations]
    scanf("-");
    ^
C:\Program Files (x86)\Windows Kits\10\Include\10.0.10240.0\ucrt\stdio.h:1270:33:␣
↪note: 'scanf' has been explicitly marked deprecated here
_CRT_STDIO_INLINE int __CRTDECL scanf(
                                ^
1 warning generated.
```

This is all well and good for simple projects. The problem comes into play when you are dealing with larger code bases (dozens, if not hundreds/thousands of files). It's also a complex command line to remember. You could batch this up, if you wanted to. Nothing wrong with that. Or you could use something like `make`, `nmake` or `msbuild` to better manage this process.

But go ahead and take a look at `make` or `nmake`. We'll wait. Here's even a quick tutorial

These are not easy to use systems. Once you get used to them, you can be really productive, but when you have IDEs like Visual Studio, the visual interface makes these systems look like the 1970s based tools that they are. There have been other attempts to make this kind of setup easier to use (`CMake`, `premake` come to mind), but they don't hold a candle to the simplicity of a visual editor.

That said, `make` is a standard. `Cmake` is incredibly prevalent in the Open Source community. It's worth it to investigate these (FYI, Visual Studio 2017 supports `CMake` out of the box).

Summary

That's a fair bit for this tutorial. That should give you a fair bit to chew on. In the next tutorial, I'll go ahead and go a little more into C++ as a language.

Until Next Time!

Review 03 - Where we start coding

Summary

Up to this point, we've been working through some fundamental of setting up and building code. We haven't actually made a program that **does** anything. I've spent a bit of time trying to figure out what our actual review project would be that is *something* other than a trivial `Hello World` example. I wanted it to be something that is both simple, but has some relevance.

Since we all have some interest in graphics, I took around to see what's out there for simple graphics abstractions. I don't want to go over OpenGL or DirectX just yet; I've covered both of those, in C# in other tutorials. This set of reviews is about C++.

There are a *lot* of options out there:

- SDL,
- GLFW,
- SFML,
- Cinder,
- Ogre3D,
- BGFX,
- Magnum

... the list is quite large. Don't believe me? Go to Github and search and see the results for yourself. Now, that list includes engines as well as Graphics libraries but I think it illustrates the point; picking a library isn't a trivial task.

I finally chose Allegro. I'll try out other libraries as well, but I went with Allegro because:

- It's got a good NUGet integration with Visual C++: > look here <<
- I'm not looking for 3D or complex drawing facilities.
- I've used it way, way, way back, so there's a bit of familiarity with it.

I don't have stong opinions on Allegro just yet. I've used other wrappers in the past and they all have their strengths and weaknesses. So be adventurous and try a few other libraries on your own!

NB: I may give tinyrender a try, as it illustrates mixing in source files from an external project later.

Allegro setup

I've already added to this project the Allegro NUGet package. I had to modify the `vcproj` by hand, but that was a trivial thing to do (mostly to get the NUGet package folder correct).

I'm not going to go into detail as to how I set it up - that's covered in the earlier link to Allegro. That and depending on your OS, you'll have different requirements. But feel free to use this project as a point of reference if you're going the Visual C++ route.

One thing I do want to point out is that I am linking the libraries against a single, monolithic `static library`. So what are the options here, and what do they mean.

Simply put, our options are:

- static library
- DLL
- Multithreaded DLL

What are these? What do they mean. I don't want to assume you all know the difference, so I'll take a minute to explain (and for more information, you can research further on your own).

Static Library

In our previous examples, specifically in Review 2, we were linking against a static library. What this means is that the functions/classes that live in a library are copied from the library and embedded directly into the final executable. That *printf* function that we used? Remember, that's not part of the C++ language, it's a library function that lives in the "C Run-Time" (CRT) library. Different compilers. For the Microsft compiler, that's the "libcmt*.lib" and "msvcrt*.lib".

From the Microsoft reference site for libraries here we have the following:

Library	Characteristics	Option	Preprocessor directives
libcmt.lib	Statically links the native CRT startup into your code.	/MT	_MT
libcmtd.lib	Statically links the Debug version of the native CRT startup. Not redistributable.	/MTd	_DEBUG, _MT
msvcrt.lib	Static library for the native CRT startup for use with DLL UCRT and vcruntime.	/MD	_MT, _DLL
msvcrtd.lib	Static library for the Debug version of the native CRT startup for use with DLL UCRT and vcruntime. Not redistributable.	/MDd	_DEBUG, _MT, _DLL

I've thinned out this table as we don't want to talk about managed languages yet.

`libcmt.lib` and `libcmtd.lib` take the machine code, stored in the respective `.lib` file and embeds that into your executable. It's that simple.

`msvcrt.lib` and `msvcrtd.lib` inject a ... reference, for want of a better term, into your executable that looks into a DLL (*MSVCP*.dll*) for the function to call.

So, why DLLs? Simply put, it allows the author of the library to change the function (to fix issues) or to share the function across multiple applications. Think about it this way - every application that uses *printf* as a static library has to embed that function into their executable, bloating the size of the executable. You have hundreds (if not thousands) of executables on your machine, each embedding the same functions into the execuatble and you've got and incredible amount of duplication of a single function across executables. Putting common functions into a DLL avoids that. It also means that if your function has issues, all you have to do is replace the DLL to get a new version of that function.

But that also runs us into other problems. You've no doubt heard the term on windows for 'DLL Hell'. You might not have a great definition for this - so let me lay out an example for you.

Say you have a vendor that provides application A that installs a set of DLLs. Later, the same vendor might distribute another application, B, that also installs a new set of DLLs, but they haven't updated application A to use the new DLLs. And a shared function in those DLLs has been updated. It is now completely possible that change has broken application A. So you uninstall application A, which removed the DLL shared with application B and now application B fails to work. And the cycle continues if you reinstall application A ...

It's not as bad as that, because you can resolve a number of those issues by versioning your DLL, but that comes with its own set of problems. Anyway, the long and the short of it is, for these Reviews, where possible I will be statically linking the executables.

The C/C++ Language - Essentials

In the first few reviews, I glossed over a lot of the C++ syntax to focus more on the structure of the build process. I'll fall back here a bit now and do a very high-level review of the more C like aspects of C++; looking at core language syntax, data types, functions, conditionals and loops.

But first, the code!

```
// Review03.cpp : Defines the entry point for the application.
//

#include <stdio.h>
#include <allegro5\allegro.h>
#include <allegro5\allegro_image.h>
#include <allegro5\allegro_primitives.h>
#include <allegro5\allegro_font.h>

const int maxiterations = 50;

void DrawFrame(int width, int height);

int main(int argc, char* argv[])
{
    al_init();
    al_init_font_addon();
    al_init_image_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    ALLEGRO_FONT* font = al_create_builtin_font();
    ALLEGRO_EVENT_QUEUE* eventQueue = nullptr;

    eventQueue = al_create_event_queue();
    al_register_event_source(eventQueue, al_get_display_event_source(display));

    al_clear_to_color(al_map_rgb(0, 0, 0));

    al_draw_text(font,
                 al_map_rgb(255, 255, 255),
                 400, 300,
                 ALLEGRO_ALIGN_CENTER,
                 "Welcome to Review03");

    while (true)
    {
        ALLEGRO_EVENT event;
        ALLEGRO_TIMEOUT timeout;
        al_init_timeout(&timeout, 0.06);

        bool get_event = al_wait_for_event_until(eventQueue, &event, &timeout);

        if (get_event && event.type == ALLEGRO_EVENT_DISPLAY_CLOSE)
        {
            break;
        }

        DrawFrame(800, 600);
        al_flip_display();
    }

    al_destroy_font(font);
    al_destroy_display(display);

    return 0;
}

void DrawFrame(int width, int height)
{
    // Drawing individual pixels in this manner is incredibly slow. This is only for␣
↪illustration
    // on the C syntax.
    for (int index = 0; index < maxiterations; index++)
    {
        al_put_pixel(rand() % width, rand() % height, al_map_rgb(rand()%255, rand()
↪%255, rand()%255));
    }
}
```

And, as an output, we get the following:

C++ Comments

Not going to say a lot about comments. There are two types:

- //: Begins a comment which continues until the end of the line. Can be put anywhere in the line.
- /* */: Begins a comment block that starts with the /* and ends with the */. Can start or end anywhere.

Preprocessor Macros

When you start with the # symbol, you are beginning a ‘Preprocessor Directive’. Each directive occupies one line (and can be extended across multiple lines using the ‘ *continuation character) and has the following format: - after the ‘#* symbol, you can invoke one of the following commands

- define
- undef
- include
- if
- ifdef
- ifndef
- else
- elif
- endif
- line
- error

- pragma
- you can then add any arguments, based on the aforementioned instruction.

What we are currently doing with the preprocessor is including a header file in lines 4-8 of the example program. These bring in the function signatures as well as other elements defined in the header files. Feel free to peruse the files to see what jumps out at you.

Later, we'll discuss more about preprocessor macros. But for now, it's enough to understand that there is more to the preprocessor than just includes.

Constants

The next line:

```
const int maxiterations = 50;
```

defines a variable of type *int* (integer value) that is constant - it can be set once and cannot be changed after the fact.

We'll also dig into the *const* keyword later as it has multiple uses.

Forward Declarations

Next, we see this:

```
void DrawFrame(int width, int height);
```

This tells the compiler that we have a function called *DrawFrame* the has no return (thus the *void* in front of the function) and takes two arguments (an integer *width* and *height*). Note that this is exactly what you would put into a header file.

The Entry Point Into Our Application

In C/C++ we define the entry point to our application as:

```
int main(int argc, char* argv[])
```

Actually, that's a bit of a lie. For a 'console' application, we define the entry point as above. You can also define the entry point into your application like so:

```
int main()
```

or

```
void main()
```

`int main()` requires you to return an 'error code' back to the Operating System. A return value of `0` indicates no error in program execution. Anything else is an error. These 'error results' can be processed by batch files (or shell scripts) to control batch processing flow. But that discussion is outside of the scope of this article.

`void main()` requires no return value. The OS assumes that the program has other means of determining or logging error conditions.

However, back to the original definition:

```
int main(int argc, char* argv[])
```

The two parameters passed into the `main` function, `argc` and `argv`:

`argc` - represents the number of 'arguments' passed in on the command line. - we always include the application name in that count

- eg:
  - *app.exe* has an *argc* value of 1
  - *app.exe /F /S /N* has an *argc* value of 4

argv - This is an array (that's what [] represents in C/C++) of char pointers. - char * is a 'null terminated string' - it's the C/C++ way of defining strings. - All native strings in C/C++ have a null that defines the end of the string. - ** this does not include other string types, like STL's string.

What does this mean? In the example of the command line looking like this:

app.exe /F /S /N

You can access each element in the command line like so:

array element	value
argv[0]	"app.exe"
argv[1]	"/F"
argv[2]	"/S"
argv[3]	"/N"
argv[4]	*an error*

We'll go into character strings later. For now, understand that the *char* C/C++ data type maps to a single byte.

Calling Functions

Like printf, calling a function is pretty straightforward:

```
al_init();
al_init_font_addon();
al_init_image_addon();
al_init_primitives_addon();
```

Each of those lines represents a call to an Allegro function. Each of those functions are defined in a header file. They should map to:

Function Name	Header File
al_init	allegro.h
al_init_font_addon	allegro_font.h
al_init_primitives_addon	allegro_primitives.h

Declaring Variables

Nothing fancy about the following - we're just declaring variables:

```
ALLEGRO_DISPLAY* display = al_create_display(800, 600);
ALLEGRO_FONT* font = al_create_builtin_font();
ALLEGRO_EVENT_QUEUE* eventQueue = nullptr;
```

OK, one thing that may be a bit odd, if you're coming from an older version of C++, we have a nullptr keyword. This was added into the language spec back in C++ 11. This is a 'pointer literal'. When we dig into pointers later, we'll go over it more, but understand that nullptr is much more useful to us that null was. So if your compiler supports it, use it.

What kinds of variables do we have availble to us in C/C++? There actually aren't that many:

Type Name	Description
char	A single byte worth of information. This usually maps to the ASCII code table. But not necessarily.
int	An integer. This can have modifers added to it like *unsigned*, *short* and *long*.
float	An IEEE floating point number. A great breakdown of it is this.
double	See the above for a breakdown of a double.
bool	True or False, 1 or 0, on or off - it's a boolean!
void	Represents nothing. Used to define no return value in a function, but also has other (pointer) meanings.

We can enhance the base types even further using additional keywords:

Classification	Type names	Width (bits)	Notes
Character types	`char`	8	
	`char16_t`	16	At least as big as a `char`.
	`char32_t`	32	At least as big as a `char16_t`
	`wchar_t`	8/16/32	Wide character - supports the largest character set based on compiler.
Signed Integer	`short int`	16	Optimized for space to have at *least* 16 bits.
	`int`	16/32	First number is the C++ standard definition. Additional number is the max based on specialized compiler.
	`long int`	32/64	First number is the C++ standard definition. Additional number is the max based on specialized compiler.
	`long long int`	64	
Unsigned Integer	`unsigned short int`	16	Optimized for space to have at *least* 16 bits.
	`unsigned int`	16/32	First number is the C++ standard definition. Additional number is the max based on specialized compiler.
	`unsigned long int`	32/64	First number is the C++ standard definition. Additional number is the max based on specialized compiler.
	`unsigned long long int`	64	
Floating point	`float`	32	float types
	`double`	64	double types
	`long double`	80	long double types
Boolean	`bool`	8	No, it is not 1 bit. Each bool takes up 8 bits. This is why bitmasks/bitflags are useful.
Miscelleaneous	`void`	0	No data storage for a *void*.
	`nullptr`	•	nullptr is the same size as a pointer. This can vary.

We are not limited to just these data types. We can create our own 'types' via structures and classes. However, they must be composed, at a bare minimum, of these types.

I'll leave it to the reader to understand the min/max values that can be stored in each numerical data type.

Loops and Conditionals

Just like every language out there, C/C++ has loops and conditionals.

Loops look like this:

```
// while loop
// while (<iteration condition>)
// {
//     // do stuff
// }

int index = 0;
while (index < 10)
{
    index = index + 1; // can also be written 'index++;' or '++index;'
}

// for loop
// for ( <init>; <iteration condition>; <expression>)
// {
//     // do stuff
// }

for (int counter = 0; counter < 10; counter++)
{
    printf("counter: %d\n", counter);
}

// do while loop
// do
// {
//     // do stuff
// } while (<iteration condition>)

index = 0; // index was already declared above
do
{
    index++;
} while (index < 10)

// More advanced looping structures that we'll cover later
// but show here for a level of completeness, as this was
// introduced in C++ 11 and has a series of additional features.
// This, however, is the simplest case.
int localVector[] = {5, 10, 20, 100, 1024, 5150};
for (int value : localVector)
{
    printf("%d ", value);
}
```

C++ also has conditional statements:

```
// If condition
// if (<expression evaluates to true>)
// {
//      // do stuff
// }
```

```
int value = 0;
if (value == 1)
{
    // Do stuff
}

// If-Else condition
// if (<expression evaluates to true>)
// {
//      // do stuff
// }
// else
// {
//      // do something else
// }
if (value == 1)
{
    printf("We don't see this\n");
}
else
{
    printf("We see this\n");
}

// If-ElseIf condition
// if (<expression evaluates to true>)
// {
//      // do stuff
// }
// else if (<expression evaluates to true>)
// {
//      // do something else
// }
// else // optional
// {
//      // otherwise we do this
// }

value = 3;
if (value == 0)
{
    printf("We don't see this\n");
}
else if (value == 1)
{
    printf("We don't see this either\n");
}
else
{
    printf("We see this\n");
}

// Switch statement
// switch (<expression>)
// {
//      case <constant1>:
//      { // Brace is optional, but recommended for scoping
//          // Do stuff
//      }
//      break;
//
//      case <constant2>:
//      { // Brace is optional, but recommended for scoping
//          // Do stuff
//      }
//      break;
//
//      case <constantN>:
//      { // Brace is optional, but recommended for scoping
//          // Do stuff
//      }
//      break;
//
//      default:
//      { // Brace is optional, but recommended for scoping
//          // Do stuff
//      }
//      break;
// }

switch (value)
{
    case 0:
    {
        printf("Value is a zero\n");
    }
    break;

    case 1:
    {
        printf("Value is a one\n");
    }
    break;

    default:
    {
        printf("I don't know!\n");
    }
    break;
}
```

I expect there's nothing new there for everyone, but I wanted to add it just for completeness sake.

In the code, the only thing I'll point out is:

```
while (true)
{
    ALLEGRO_EVENT event;
    ALLEGRO_TIMEOUT timeout;
    al_init_timeout(&timeout, 0.06);

    bool get_event = al_wait_for_event_until(eventQueue, &event, &timeout);

    if (get_event && (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE))
    {
        break;
    }
```

In the above, the conditional `if (get_event && (event.type == ALLEGRO_EVENT_DISPLAY_CLOSE))` has a `&&` in it. That is the boolean `AND` operator. The Boolean `OR` operator is defined as `||`.

Please note that there is a difference between `&&` and `&` as well as `||` and `|`. The first, as pointed out earlier defines a boolean `AND`/`OR` operation. The latter defines a *bitwise* `AND`/`OR` operation. If you don't know what a bitwise operation is, we need to talk.

To Summarize

That's one fully functional C/C++ bit of code. We've looked at the language from a fairly simple starting point with this bit of code. In the next Review, we'll look at classes to round out the simple language review. We'll also talk about different build types and what they're used for.

With that, I'm out.

Review 04 - Where we discuss structs, classes and other stuff

Overview

The source code this time around is going to be a little different. We're going to have multiple examples to go over. So this time, the *main.cpp* will work as a driver, executing optional *examples*.

The code structure will look like this:

```
main.cpp
- example01.cpp
- example02.cpp
- and so on.
```

The `main` function itself will look like this:

```
enum Executables
{
    ex01 = 0,
    ex02,
    endoflist
};

void main()
{
    Executables toRun = ex01;

    switch (toRun)
    {
        case ex01:
        {
            example01();
        }
        break;

        case ex02:
        {
            example02()
        }
        break;

        default:
            break;

    };
}
```

Change the value of `toRun` to try a different example. Yes, I could drive this via the keyboard and a prompt, but you'll want to evaluate (and change) each of the examples. Changing a single int value will, in my option, be the faster way to iterate.

The `struct` keyword and our first example (example01.h/cpp)

Here's a look at the code behind our first example:

```
///␣
↪=====================================================================================
/// In this example, we look at defining new data types using the `struct` keyword
///␣
↪=====================================================================================

#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_primitives.h>

extern ALLEGRO_FONT* gFont;

/// standard C++ definition of a struct
struct Vertex
{
    float x;
    float y;
};

/// definition of a struct with an instance of the struct (a variable,
/// in this case, called `gGlobalPoint`)
struct Point3D
{
    float x;
    float y;
    float z;
} gGlobalPoint;

/// The more C friendly version
extern "C"
{
    typedef struct Point2D
    {
        float x;
        float y;
    } Point2D;
}
```

```
const float width = 800.0f;
const float height = 600.0f;

void Example01()
{
    struct Vertex point1; // This is valid (and very c like)
    Vertex point2;  // This is valid, and C++

    gGlobalPoint.x = 0.5f * width;
    gGlobalPoint.y = 0.25f * height;

    point1.x = 0.0f;
    point1.y = 0.0f;

    point2.x = 1.0f * width;
    point2.y = 1.0f * height;

    /// An anonymous struct
    struct
    {
        float x;
        float y;
    } point3;
    point3.x = .75f * width;
    point3.y = .5f * height;

    Point2D point4; // This is C like
    point4.x = .25f * width;
    point4.y = .4f * height;

    // Initialization of the new variable `textPos`
    Vertex textPos =
    {
        500.0f,  // x field of Vertex
        20.0f    // y field of Vertex
    };

    // And draw some text to show what points we're looking at
    al_draw_textf(gFont,
        al_map_rgb(255, 255, 255),
        textPos.x, textPos.y,
        ALLEGRO_ALIGN_LEFT,
        "Point1 (%f, %f)", point1.x, point1.y);
    textPos.y += 15;

    al_draw_textf(gFont,
        al_map_rgb(255, 255, 255),
        textPos.x, textPos.y,
        ALLEGRO_ALIGN_LEFT,
        "Point2 (%f, %f)", point2.x, point2.y);
    textPos.y += 15;

    al_draw_textf(gFont,
        al_map_rgb(255, 255, 255),
        textPos.x, textPos.y,
        ALLEGRO_ALIGN_LEFT,
        "Point3 (%f, %f)", point3.x, point3.y);
    textPos.y += 15;
    al_draw_textf(gFont,
        al_map_rgb(255, 255, 255),
        textPos.x, textPos.y,
        ALLEGRO_ALIGN_LEFT,
        "Point4 (%f, %f)", point4.x, point4.y);
    textPos.y += 15;

    al_draw_textf(gFont,
        al_map_rgb(255, 255, 255),
        textPos.x, textPos.y,
        ALLEGRO_ALIGN_LEFT,
        "gGlobalPoint (%f, %f)", gGlobalPoint.x, gGlobalPoint.y);
    textPos.y += 15;

    // Finally, draw some lines.
    al_draw_line(point1.x, point1.y, point2.x, point2.y, al_map_rgb(255, 255, 255),␣
↪1);
    al_draw_line(point2.x, point2.y, point3.x, point3.y, al_map_rgb(255, 0, 0), 1);
    al_draw_line(point3.x, point3.y, point4.x, point4.y, al_map_rgb(0, 255, 0), 1);
    al_draw_line(point4.x, point4.y, gGlobalPoint.x, gGlobalPoint.y, al_map_rgb(255,␣
↪0, 255), 1);
    al_draw_line(gGlobalPoint.x, gGlobalPoint.y, point1.x, point1.y, al_map_rgb(255,␣
↪255, 0), 1);

    al_flip_display();
    al_rest(5.0);
};
```

We have a bit more code this time around. What I've done here is enumerate the number of ways that we can use a `struct` (and `typedef` for that matter) to create our own new data type, variants of a 2D vector.

I'm fairly certain most of this will be common knowledge to most readers, so I'm not going to review every line of code here. However, this bit may be unfamiliar to new C++ coders:

```
extern "C"
{
    typedef struct Point2D
    {
        float x;
        float y;
    } Point2D;
}
```

We've seen the `extern` keyword used earlier in this codebase, declaring an `ALLEGRO_FONT`

`extern ALLEGRO_FONT* gFont;`

If you check out line 8 in `main.cpp`, you'll see the definition of the `gFont` variable:

`ALLEGRO_FONT* gFont = nullptr;`

The `extern` keyword just says that the declaration of this variable is done outside this file, and it's up to the linker to resolve it. Nothing more than that.

But then we have that `extern "C"` ... that sure doesn't look like that's what's going on here.

But it kind of is. Here's what Microsoft has to say about the `extern` keyword:

> The `extern` keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined). When modifying a variable, `extern`

> specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file. Declarations of variables and functions at file scope are external by default.

The key takeaway from that is that is specifies the linkage conventions. From a little further down in the docs Microsoft Docs

> In C++, when used with a string, extern specifies that the linkage conventions of another language are being used for the declarator(s). C functions and data can be accessed only if they are previously declared as having C linkage. However, they must be defined in a separately compiled translation unit.
>
> Microsoft C++ supports the strings "C" and "C++" in the string-literal field. All of the standard include files use the extern "C" syntax to allow the run-time library functions to be used in C++ programs.

In short, this enforces the C linkage rules for anything encompassed in the the braces. This isn't a cheat to force code to be 'pure C', but it does help enfoce *some* rules (alllinker based rules). Read: This doesn't make the code compile in C - you're using a C++ compiler, it'll still compile it as C++. It just links it like it's C. Test this theory if you'd like by removing the `typedef`.

Or, crack open the following link: Compiler Explorer to see the warnings.

Anyway, this was a bit off-topic. We'll look at `extern` after a bit, when looking at linking in C libraries.

When we run this bit of code, we get the following result:

Images/review04/example01.png

Counting bytes and a knock to our sanity (example02.h/cpp)

How much space does a struct take up? From our previous review (Review03), we had a table that illustrated how big each native data type would be. To help illustrate, let's take the following code:

```
// Example program
#include <stdio.h>

int main()
{
    printf("Size of a char: %lu byte(s)\n", sizeof(char));
    printf("Size of a float: %lu byte(s)\n", sizeof(float));
    printf("Size of a double: %lu byte(s)\n", sizeof(double));
    printf("Size of a int: %lu byte(s)\n", sizeof(int));
    printf("Size of a unsigned int: %lu byte(s)\n", sizeof(unsigned int));
}
```

and run it in the C++ shell C++ shell example

Here's the output:

```
Size of a bool:         1 byte(s)
Size of a char:         1 byte(s)
Size of a float:        4 byte(s)
Size of a double:       8 byte(s)
Size of a int:          4 byte(s)
Size of a unsigned int: 4 byte(s)
```

This is a great little table for us to use now. In *example02* I've done the same, so that we have a reference point to work back from.

```
#include <allegro5/allegro_font.h>

struct Point2D
{
    float x;     // 4 bytes
    float y;     // 4 bytes
};               // 8 bytes total

struct RGB
{
    float r;     // 4 bytes
    float g;     // 4 bytes
    float b;     // 4 bytes
};               // 12 bytes total

struct RGBA
{
    float r;     // 4 bytes
    float g;     // 4 bytes
    float b;     // 4 bytes
    float a;     // 4 bytes
};               // 16 bytes total

struct UV
{
    float u;     // 4 bytes
    float v;     // 4 bytes
};               // 8 bytes total


struct Vertex
{
    Point2D position;   // 8 bytes
    RGB     color;      // 12  bytes
    UV      texCoord;   // 8 bytes
};                      // 28 bytes total

struct VisibleVertex01
{
    bool    visible;    // 1 byte
    Point2D position;   // 8 bytes
    RGB     color;      // 12 bytes
    UV      texCoord;   // 8 bytes
};                      // 29 bytes

extern ALLEGRO_FONT* gFont;

void Example02()
{
    Point2D textPos;
    textPos.x = 10.0f;
    textPos.y = 20.0f;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of char:             %d bytes", sizeof(char));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of float:            %d bytes", sizeof(float));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of bool:             %d bytes", sizeof(bool));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of Point2D:          %d bytes", sizeof(Point2D));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of RGB:              %d bytes", sizeof(RGB));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of RGBA:             %d bytes", sizeof(RGBA));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of UV:               %d bytes", sizeof(UV));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of Vertex:           %d bytes", sizeof(Vertex));
    textPos.y += 15;
    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of VisibleVertex01: %d bytes", sizeof(VisibleVertex01));
    textPos.y += 15;


    al_flip_display();

    al_rest(5.0);

}
```

Now, we run this code:

Images/review04/struct_sizes.png

Wait now ... What's going on with *VisibleVertex01*? Shouldn't that be 29?

It gets worse. Go ahead and inject the following into *VisibleVertex01*:

```
struct VisibleVertex01
{
    bool    visible;     // 1 byte
    Point2D position;    // 8 bytes
    bool    dummyA;      // 1 byte
    RGB     color;       // 12 bytes
    UV      texCoord;    // 8 bytes
};                       // 30 bytes total?
```

Images/review04/struct_sizes_02.png

```
struct VisibleVertex01
{
    bool    visible;     // 1 byte
    Point2D position;    // 8 bytes
    bool    dummyA;      // 1 byte
    RGB     color;       // 12 bytes
    bool    dummyB;      // 1 byte
    UV      texCoord;    // 8 bytes
};                       // 31 bytes total?
```

Images/review04/struct_sizes_03.png

```
struct VisibleVertex01
{
    bool    visible;     // 1 byte
    Point2D position;    // 8 bytes
    bool    dummyA;      // 1 byte
    RGB     color;       // 12 bytes
    bool    dummyB;      // 1 byte
    UV      texCoord;    // 8 bytes
    bool    dummyC;      // 1 byte
};                       // 32 bytes total?
```

Images/review04/struct_sizes_04.png

What's going on? Is the `sizeof` funtion not working?

I mean, that's not a lot of wasted space for an individual element, but it adds up quickly. Thus we really can't ignore it. In the last version of the *VisibleVertex01* struct, we see that we've wasted 8 bytes per *VisibleVertex01*. If we were to have a mesh with 65,000 unique instances of that type, that's 520,000 bytes.

So, how can we fix that? Well, we can use the preprocessor like so:

```
#pragma pack(push)
#pragma pack(1)
struct VisibleVertex02
{
    bool    visible;     // 1 byte
    Point2D position;    // 8 bytes
    RGB     color;       // 12 bytes
    UV      texCoord;    // 8 bytes
};                          // 29 bytes total
#pragma pack(pop)

// a little further down ...

    al_draw_textf(gFont, al_map_rgb(255, 255, 255), textPos.x, textPos.y, ALLEGRO_
↪ALIGN_LEFT, "Size of VisibleVertex02: %02d bytes", sizeof(VisibleVertex02));
    textPos.y += 15;
```

And now run the program, you get the following:

Images/review04/struct_sizes_success.png

That's great, but that still doesn't answer *why*.

Let's start with something simpler:

```
// Example program
#include <stdio.h>

struct TestA
{
    char a; // 1 byte
    int  b; // 4 bytes
};

int main()
{
    printf("Size of a char:  %02lu byte(s)\n", sizeof(char));
    printf("Size of a int:   %02lu byte(s)\n", sizeof(int));
    printf("Size of a TestA: %02lu byte(s)\n", sizeof(TestA));

}
```

Which results in:

```
Size of a char:  01 byte(s)
Size of a int:   04 byte(s)
Size of a TestA: 08 byte(s)
```

That *should* have been 5 bytes, no matter how you slice it. So, how does this work? What's happening here is that *something* in that structure is adding padding. *Why* is it doing that and *where* is it doing it are the questions we need to answer.

Fundamentally, when dealing with memory, CPUs access memory in *word* sized chunks. I purposely didn't explicitly say how big a word is, because that actually varies on architecture. For our purposes, this will be either 4 byte or 8 byte alignment (4 for 32 bit systems, 8 for 64 bit systems).

For now, let's assume a 4 byte alignment (makes the math easier to work with). In the *TestA* struct we have the first field *a* starting at byte 0. This is A-OK (0 is always a good starting point). And it is a byte long. So we can assume that the next field *b* starts on the second byte, right?

Nope!

Remember, the CPU is reading in the value from the *word* aligned boundary. In this case, 4 bytes. So there is padding added into the struct between fields a and b of 3 bytes. In essence, the structure looks like this:

```
struct TestA
{
    char a; // 1 byte
    char pad[3];
    int  b; // 4 bytes
};
```

Don't believe me? Let's write some code!

```
// Example program
#include <stdio.h>
#include <memory.h>

struct TestA
{
    char a; // 1 byte
    int  b; // 4 bytes
};

struct TestB
{
    char a;      // 1 byte
    char pad[3]; // 3 bytes
    int  b;      // 4 bytes
};

int main()
{
    printf("Size of a char:  %02lu byte(s)\n", sizeof(char));
    printf("Size of a int:   %02lu byte(s)\n", sizeof(int));
    printf("Size of a TestA: %02lu byte(s)\n", sizeof(TestA));
    printf("Size of a TestB: %02lu byte(s)\n", sizeof(TestB));

    TestA testA;

    testA.a = 'G';
    testA.b = 76;

    TestB testB;

    memcpy(&testB, &testA, sizeof(TestA));

    printf("testA.a [%c] testB.a [%c]\n", testA.a, testB.a);
    printf("testA.b [%d] testB.b [%d]\n", testA.b, testB.b);

    int result = memcmp(&testB, &testA, sizeof(TestA));

    if (result == 0)
    {
        printf("The memory blocks are equal!\n");
    }
    else
    {
        printf("The memory blocks are UNEQUAL!!!\n");
    }
}
```

```
```

And the results?

```
Size of a char:  01 byte(s)
Size of a int:   04 byte(s)
Size of a TestA: 08 byte(s)
Size of a TestB: 08 byte(s)
testA.a [G] testB.a [G]
testA.b [76] testB.b [76]
The memory blocks are equal!
```

You can see this in the C++ shell padding example 01

The struct should give you a good idea as to what it looks like, but I like pictures ...

```
| word boundary | word boundary |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |   Bytes
| a |           | b               |   |   TestA structure
| a |     pad   | b               |   |   TestB structure
```

Let's try it with some different sized data types.

```
// Example program
#include <stdio.h>

struct TestA
{
    char a;      // 1 byte
    double b;    // 8 bytes
};               // 9 bytes?

int main()
{
    printf("Size of a char:   %02lu bytes\n", sizeof(char));
    printf("Size of a double: %02lu bytes\n", sizeof(double));
    printf("Size of a TestA:  %02lu bytes\n", sizeof(TestA));
}
```

```
Size of a char:   01 bytes
Size of a double: 08 bytes
Size of a TestA:  16 bytes
```

[cppshell link](pp.sh/5byeg)

OK, now it's just messing with us. If we're locking into *word* sized boundaries, shouldn't that have been 12?

```
char   a; // 4  bytes
double b; // 8  bytes
// total      12 bytes
```

There's one more piece to the puzzle, and the c-faq has a great blurb on it On the C-faq

> On modern 32-bit machines like the SPARC or the Intel 86, or any Motorola chip from the 68020 up, each data iten must usually be "self-aligned", beginning on an address that is a multiple of its type size. Thus, 32-bit types must begin on a 32-bit boundary, 16-bit types on a 16-bit boundary, 8-bit types may begin anywhere, struct/array/union types have the alignment of their most restrictive member. The first member of each continuous group of bit fields is typically word-aligned, with all following being

continuously packed into following words (though ANSI C requires the latter only for bit-field groups of less than word size).

Let's try something to test this theory:

```
// Example program
#include <stdio.h>

struct TestA
{
    char a;
    double b;
    char c;
};

int main()
{
    printf("Size of a char:   %02lu bytes\n", sizeof(char));
    printf("Size of a double: %02lu bytes\n", sizeof(double));
    printf("Size of a TestA:  %02lu bytes\n", sizeof(TestA));
}
```

and the output?

```
Size of a char:   01 bytes
Size of a double: 08 bytes
Size of a TestA:  24 bytes
```

link to the code

I think you're starting to see a pattern here. Padding is done, per element, based on the largest type alignment requirements.

For completeness, here's the C++ shell version of the *TestA*/*TestB* structs using the `#pragma pack(1)`: reference link

I've purposely avoided talking about pointers. I've done this on purpose as this throws a little more complexity into the mix.

I will be talking about them at a later point, but for now, I'd like to move on to classes.

Where we add functionality to our types (example03.h/cpp)

Object Oriented programming.

I'm not goint to talk about Object Oriented programming.

I mean, seriously, in 2017, I think there's enough material on the web to cover Inheritance, Encapsulation, Abstraction, interfaces ... that's not what I wanted to write this series about. What I want to talk about is the nitty-gritty of the C++ implementation of classes.

If you're looking for an introduction to Object Oriented Programming in C++, I'd recommend you start OOP Introduction here, Welcome to OOP here to start. As far as printed material, it's been so long since I've taught C++/OOP, I don't think I can recommend anything remotely good. I'm not sure how well Scott Meyers' series of books holds up these days, but they were decent reading back in '03. I do remember using "C++ How to Program" as a teaching resource back in the 90s, but I haven't looked at it in over a decade How to Program C++ by Paul Deitel here

What I do want to talk about is the syntax of Classes. I think that tends to get lost in the shuffle of folks learning C++, so I don't mind burning a bit of time as part of the refresher.

But first, let's look at the `struct` keyword again. We know that it allows us to set up a collection of fields to layout a structure in memory. But what if we were had the ability to bind Functions as a field?

like so:

```
struct Vertex
{
    float x;
    float y;

    void Set(float inX, float inY)
    {
        x = inX;
        y = inY;
    }
};
```

We've essentially merged fields with functions. Could that work? Go ahead and throw that into the C++ shell example for fields with functions here

It compiles!

So, ... how do we use it? I mean, we have a new struct that has a function, but how do we go about *doing* something with it?

Well, let's create a new variable of type *Vertex* called *point01*:

```
int main()
{
    Vertex point1;
}
```

Add that and compile. Aside from some warnings, that works fine.

So, we already know how to access the fields. What's the chances that accessing the functions is as trivial?

Try this:

```
int main()
{
    Vertex point1;

    point1.Set(10.0f, 20.0f);

    printf("The value of point1 is (%f, %f)\n", point1.x, point1.y);
}
```

Run it in the C++ shell and ...

```
The value of point1 is (10.000000, 20.000000)
```

That's pretty much it. The `struct` keyword is all you need to create objects with data and functions! We're done! There's nothing left to learn about C++!

That's total nonsense, I know. There's so much more to cover.

The thing with the `struct` keyword is that everything we've done so far is of `public` scope. That's the default scope for anything defined in a struct. That's mostly for backwards compatability, as the original definition of the struct keyword in C didn't have a concept of 'data/functional hiding'.

So, scoping in structs. Like I said before, the default scope for a `struct` is `public`. There's also `private` and `protected`.

Both the `private` and `protected` keywords hide elements of your structure. So if you were to do the following:

```
struct Vertex
{
    float x;
    float y;

    void Set(float inX, float inY)
    {
        x = inX;
        y = inY;
    }

    private:
    double mX;

    protected:
    char buffer[3];
};
```

You would not be able to access either `mX` or `buffer` in the main function:

```
int main()
{
    Vertex point1;

    point1.Set(10.0f, 20.0f);
    point1.mX = 5.0;

    printf("The value of point1 is (%f, %f)\n", point1.x, point1.y);
}
```

When compiling produces:

```
In function 'int main()':
16:12: error: 'double Vertex::mX' is private
28:12: error: within this context
```

Check it out with a private and protected example here

However, we can access it from inside the `Vertex` class:

```
struct Vertex
{
    float x;
    float y;

    void Set(float inX, float inY)
    {
        x = inX;
        y = inY;
        mX = inX - inY;
    }

    private:
    double mX;
    protected:
    char buffer[3];
};
```

Code with another a private and protected example here

What we've seen so far is that we're hiding `private` and `protected` behind the `struct` barrier. We can also use derivation of structs to build object hierarcies:

Creating a new struct called `ColorVertex` like so:

```
struct Vertex
{
    float x;
    float y;

    void Set(float inX, float inY)
    {
        x = inX;
        y = inY;
        mX = inX - inY;
    }

    private:
    double mX;

    protected:
    char buffer[3];
};

struct ColorVertex : Vertex
{
    int r;
    int g;
    int b;
};
```

Allows *ColorVertex* to access all *public* and *protected* members of `` Vertex`, but it hides everything that's ``private ``. Go ahead, try and access *mX* and the *buffer* members of `Vertex` through `ColorVertex`. Sandbox is here

OK, so that's a very quick run-though of the *struct* usage as an object.

But we never use it.

NEVER.

OK, that's a lie. We tend to use `structs` when talking about POD (Plain Old Data) types. But when you want to define
Classes, that's when you use the `class` keyword.

What's the difference between `struct` and `class`? One thing, and one thing only - the default access level. For the *struct* keyword, the default access level is `public`. For `class` it's `private`.

- For completeness, POD (Plain Old Data) means nothing more than a struct that contains nothing but data. It can be compelex data, but it contains no 'logic'.

To convert the example over to classes?

```
// Example program
#include <stdio.h>
```

```
class Vertex
{
    public:
        float x;
        float y;

        void Set(float inX, float inY)
        {
            x = inX;
            y = inY;
            mX = inX - inY;
        }

    private:
        double mX;

    protected:
        char buffer[3];
};

class ColorVertex : public Vertex
{
    int r;
    int g;
    int b;
};

int main()
{
    ColorVertex point1;

    point1.Set(10.0f, 20.0f);

    printf("The value of point1 is (%f, %f)\n", point1.x, point1.y);
}
```

A couple of things to note:

1. When deriving from `Vertex`, we need to qualify it as `public`. eg `class ColorVertex : public Vertex`
2. If you do not do that, and define it as `class ColorVertex : Vertex` the scope of *Vertex* defaults to `private`

So what do you get from doing all that? The current set of code fails to compile due to data and functions being inaccessible? It's not trivial to describe the implementation details of `private` inheritance. Think of it like this:

When you privately derive from a base class, all `public` members become `private`. You still have access to all `protected` members, but that's it.

So, as an example:

```
// Example program
#include <stdio.h>

class Vertex
{
    public:
        float x;
        float y;

        void Set(float inX, float inY)
        {
            x = inX;
            y = inY;
            mX = inX - inY;
        }

    private:
        double mX;

    protected:
        void ProtectedSet(float inX, float inY)
        {
            x = inX + 5;
            y = inY + 5;
        }
        char buffer[3];
};

class ColorVertex : protected Vertex
{
    public:
    int r;
    int g;
    int b;

    void ExposedSet(float inX, float inY)
    {
        ProtectedSet(inX, inY);
    }

    float GetX() { return x; }
    float GetY() { return y; }
};

int main()
{
    ColorVertex point1;

    point1.ExposedSet(10.0f, 20.0f);

    printf("The value of point1 is (%f, %f)\n", point1.GetX(), point1.GetY());
}
```

link to the example is here

That is an incredibly convoluted example. I'll see if I can come up with a better one, but in all honesty, you tend *not* to use this pattern. I think in all my years of coding, I've run across it a handfull of times.

Additional references

- The Lost Art of C Structure Packing: Read this. Seriously.
- Coding for Performance: Data alignment and structures: another good read.

- Packed Structures
- Sven-Hendrik Haase paper on Alignment in C
- C-Faq blurb
- And, of course, Wikipedia

Review 05 - Where we talk about memory

Summary

Throughout these tutorials, I've purposely shyed away from talking about memory as much as I possibly can, doing a fair bit of hand-waving, talking about the size of structures, but avoiding talking about actual allocation of things in C++. In this review, I'm going to cover the intricacies of memory and memory management in C++.

Previously ...

Let's take a look at the 'boilerplate' code I have set up for a `main.cpp` file:

```
enum Executables
{
    ex01 = 0,
    ex02,
    endoflist
};

void main()
{
    Executables toRun = ex02;

    switch (toRun)
    {
    case ex01:
    {
        break;
    }

    case ex02:
    {
        break;
    }

    default:
        break;
    }
}
```

I'm actually allocating memory in this code snippet. Where? Every time I allocate a variable. So `toRun` is allocated, on the stack. It's done automatically for me byt the compiler.

But you hear all the time about C/C++ and memory leaks. Am I leaking memory here? How do I fix it?

Well, we're not leaking anything because of two things:

1. The allocation is happening on the stack
2. C/C++ scoping rules

When we allocate memory, we can allocate it from two places: the "stack" and the "heap". Every function that you call creates it's own 'stack' memory space where it stores variables local to that function. It also stores arguments passed in, and out of, the function during the invocation of said function.

We allocate memory from the heap via the `malloc`, `new` functions (and their variants). Anything we allocate from the heap, we eventually have to reclaim via `free` and `delete`, otherwise we end up with a memory leak. More on that shortly.

We also need to talk about scoping rules. We've already seen C++ keywords that help with defining the scope of a variable, when we looked at the `extern` keyword in "Review04". Remember, that allowed us to tell the compiler/linker that the variable was declared elsewhere and the linker would resolve it. Let's talk more about this. In "Review04", the allegro font variable we used (`extern ALLEGRO_FONT* gFont;`) was arguably a 'global' variable; with the right keywords, you can access that variable anywhere in your application, in any file.

Does the same hold true for the variable `toRun` in the above example? No, it does not. There is no equivalent of `main.toRun` that you can use to access that variable. So this:

```
enum Executables
{
    ex01 = 0,
    ex02,
    endoflist
};

void main()
{
    Executables toRun = ex02;

    switch (toRun)
    {
    case ex01:
    {
        break;
    }

    case ex02:
    {
        break;
    }

    default:
        break;
    }
}

void foo()
{
    main.toRun = ex01; // compile error
}
```

Fails to compile.

However, moving it out of `main()` promotes it to a more 'global' scope:

```
enum Executables
{
    ex01 = 0,
    ex02,
    endoflist
};

Executables gToRun;

void main()
{
    gToRun = gToRun;

    switch (gToRun)
    {
    case ex01:
    {
        break;
    }

    case ex02:
    {
        break;
    }

    default:
        break;
    }
}

void foo()
{
    gToRun = ex01; // compiles fine
}
```

Scoping points

Let's say we have a function with a loop:

```
void foo()
{
    for (int index = 0; index < 10; index++)
    {
        printf("%i\n", index);
    }

    printf("%i\n", index);
}
```

What do you think the output from that will be?

Trick question! It doesn't actually compile. Specifically with this error:

```
In function 'void foo()':
43:17: error: 'index' was not declared in this scope
```

That error refers to:

```
void foo()
{
    for (int index = 0; index < 10; index++)
    {
        printf("%i\n", index);
    }

    printf("%i\n", index);    // <<== error here
}
```

The compiler can't find `index` because it is only scoped to the `for` loop - initializing the variable `index` inside the `for` loop automatically enforces the scope.

A quick change fixes this:

```
void foo()
{
    int index = 0;
    for (; index < 10; index++)
    {
        printf("%i\n", index);
    }

    printf("%i\n", index);    // <<== error here
}
```

But is that a change for the better? Depends on your needs. Initializing the `index` variable inside the `for` loop automatically enforces cleanup of that variable; you don't have to worry about accidentally reusing that variable. That's a good thing, IMO. You can now legally do the following:

```
void foo()
{
    for (int index = 0; index < 10; index++)
    {
        printf("%i\n", index);
    }

    // do other stuff

    for (int index = 5; index < 10; index++)
    {
        // do something else here
    }
}
```

You can actually enforce deeper scoping of variables using the curly braces:

```
void foo()
{
    {
        int innerScope = 10;
        printf("innerScope: %d\n", innerScope);
    }

    {
        int innerScope = 20;
        printf("innerScope: %d\n", innerScope);
    }
}
```

Each `innerScope` is scoped between it's curly braces. The following:

```
void foo()
{
    {
        int innerScope = 10;
        printf("innerScope: %d\n", innerScope);
    }

    {
        int innerScope = 20;
        printf("innerScope: %d\n", innerScope);
    }

    printf("innerScope: %d\n", innerScope); // <<== error: 'innerScope' was not␣
↪declared in this scope
}
```

Fails with the error "'innerScope' was not declared in this scope".

So, what happens if we declare `innerScope` at the top of the function?

```
void foo()
{
    int innerScope = 5;
    {
        int innerScope = 10;
        printf("innerScope: %d\n", innerScope);
    }

    {
        int innerScope = 20;
        printf("innerScope: %d\n", innerScope);
    }

    printf("innerScope: %d\n", innerScope);
}
```

The results are:

```
innerScope: 10
innerScope: 20
innerScope: 5
```

Simply put, local scope overrides the parent scope. Let's say you wanted to access the 'parent' scope of the variable `innerScope`. We can use the "scope resoloution operator" to do that - it's the `::` operator. However, it is not applicable to the curly brace scoping cheat. An example helps clarify this:

```
int innerScope = 2;
void foo()
{
    int innerScope = 5;
    {
        int innerScope = 10;
        printf("innerScope: %d\n", innerScope);
    }

    {
        int innerScope = 20;
        printf("innerScope: %d\n", innerScope);
        printf("outer innerScope: %d\n", ::innerScope);
    }

    printf("innerScope: %d\n", innerScope);
}
```

Results in the output:

```
innerScope: 10
innerScope: 20
outer innerScope: 2
innerScope: 5
```

More Scoping

Let's say we have a class called `Utility`. In it, we want to define some ... utility functions.

Like the ability to test to see if a file exists.

Like this:

```
#include <stdio.h>

// On windows, use <io.h> instead. And call _access(..) instead of access(..)_
#include <unistd.h>


class Utility
{
    public:
    static bool Exists(const char* filename)
    {
        bool result = false;
        if (access(filename, 0) == 0)
            result = true;

        return result;
    }
};

int main()
{
    printf("Does foo.txt exist? %s\n", Utility::Exists("foo.txt") ? "Yes" : "No");
}
```

And since we're running on a remote box, our result:

```
Does foo.txt exist? No
```

link here

Some points about what's going on here, as we've added a few things that you may not be familiar with.

1. The `Utility` class has one static function (classes tend to describe these as 'methods'). Because it's static, you don't have to initialize an instance of the class to use it.
2. However, in order to use it, we have to use the scope resolution operator, along the the class name and the method name, to access it.

3. The method we define must be defined as static for it to be used in this 'global' manner (without instantiating an instance of `Utility`).

That's all well and good, but is that the right way to do this? Maybe we want to have a `Utility` class that has a `File` class inside of it. Can we do that? Is that a thing we can do in C++?

Let's try it!

```
#include <stdio.h>
#include <unistd.h>

class Utility
{
    public:
    class File
    {
        public:
        static bool Exists(const char* filename)
        {
            bool result = false;
            if (access(filename, 0) == 0)
                result = true;

            return result;
        }
    };
};

int main()
{
    printf("Does foo.txt exist? %s\n", Utility::File::Exists("foo.txt") ? "Yes" : "No
↪");
}
```

Running it in the C++ shell classes in classes?

```
Does foo.txt exist? No
```

Sweet Zombie Jesus it works!

And there's no real reason for it not to work. We're accessing something that's defined as `static`.

So, what does `static` actually mean?

From cppreference.com

> Inside a class definition, the keyword static declares members that are not bound to class instances.

Is this the best way to implement this? Well, let's give you another option, so you can make up your own mind. Let's re-write the class we just wrote using a `namespace` instead.

```
#include <stdio.h>
#include <unistd.h>

namespace Utility
{
    class File
    {
        public:
        static bool Exists(const char* filename)
        {
            bool result = false;
            if (access(filename, 0) == 0)
                result = true;

            return result;
        }
    };
};

int main()
{
    printf("Does foo.txt exist? %s\n", Utility::File::Exists("foo.txt") ? "Yes" : "No
↪");
}
```

C++ Shell link

The difference between the two implementations?

```
// =======================
// Previous
class Utility
{
    public:
    class File

// =======================
// New
namespace Utility
{
    class File
```

That's the difference. Well, there are other differences ...

1. The default visibility in a namespace is `public`. `classes` would be `private`, which is why we had the `public:` in the first definition.
2. `namespace` affords us another keyword, `using`. This allows us to do the following:

```
// =======================
// Previous
int main()
{
    printf("Does foo.txt exist? %s\n", Utility::File::Exists("foo.txt") ? "Yes" : "No
↪");

// =======================
// New
using namespace Utility;

int main()
{
    printf("Does foo.txt exist? %s\n", File::Exists("foo.txt") ? "Yes" : "No");
}
```

C++ Shell link 001

And we get the same output. Why would this be a better solution? Well, for starters, if we had multiple calls to `File::WhateverMethod`, it saves us some typing. Is that a big deal?

Well ...

If you've done any digging on the internet for C++ tutorials, you've probably seen a lot of tutorials using something akin to: `cout << "Here is a string" << endl;`

`cout` and `endl` live in the Standard library and are in the namespace `std`. Which means you'd need to actually do this: `std::cout << "Here is a string" << std::endl;`

Like so:

```
// Example program
#include <iostream>

void PrintMe()
{
    using namespace std;
    cout << "Here is some text" << endl;
}

void PrintMeAgain()
{
    std::cout << "Here is some more text" << std::endl;
}

int main()
{
    PrintMe();
    PrintMeAgain();
}
```

```
Here is some text
Here is some more text
```

C++ Shell Link 002

So, what do we use? Here's a rule I like to go by:

> If what we're designing needs to *only* have a scope, then a `namespace` is the right choice. Remember that a `class` is a type, and by definition requires data. So if it smells like data, then it's a class.

This is a crazy oversimplification of the process I have in my head, but I think it holds up 90% of the time.

MEMORY!!!

That's been a fairly long setup to get to this point - a program that actually allocates memory. I've use C++ Shell a lot to help kickstart some simple concepts, but this next bit is going to use Allegro, so we will be referencing the `Review05` project.

!!!WARNING!!! The following code is purely for educational purposes ONLY!

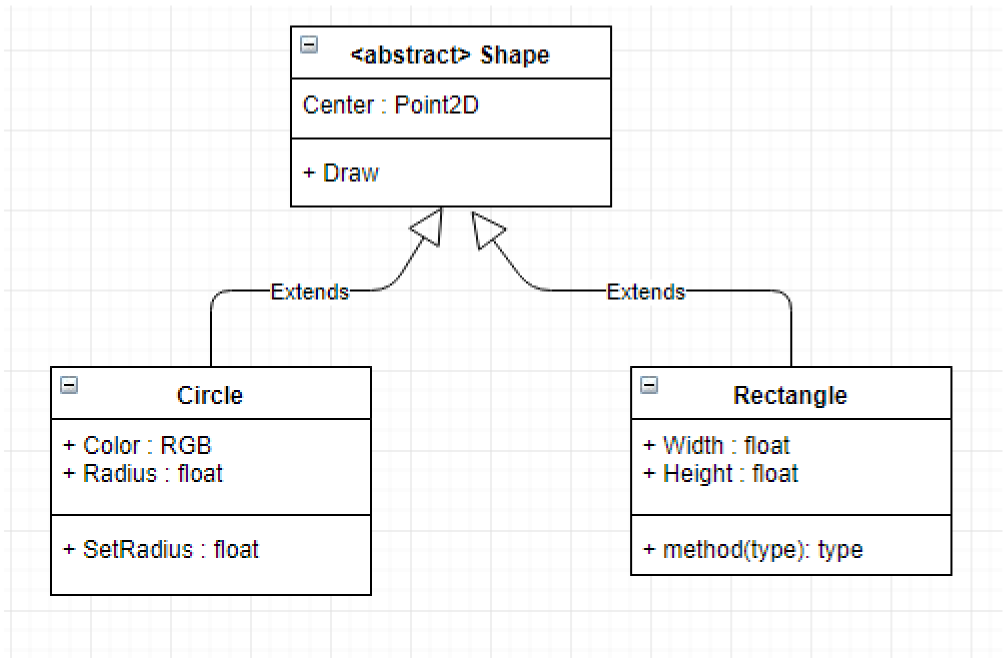
I can't stress this enough - the code you are about to see is only for educational purposes. If you were to write production code this way, you should be publicly reprimanded for it.

Anyway ...

I'm going to create a simple application that creates a random shape on-screen depending on where you click on the mouse. We won't have any idea how many elements we're going to create beforehand so we will be dynamically allocating them! There will also be a lot of code to do this. But I'm going to err on the side of verbosity and clarity, rather than performance (read - it's going to be far from optimal).

Design

I'll do a little bit of inheritance here to help illustrate some more OOP idioms. This means I'll end up with a set of classes that, in UML, looks like this:



`Shape` will be an abstract class, with two classes derived from it, `Circle` and `Rectangle`.

Let's look at what we have here in `Review05`

Shape.h/cpp

```
/// Shape.h -------------------------------------------
#pragma once

struct Point2D
{
    float x;
    float y;

    Point2D(float inX, float inY)
    {
        x = inX;
        y = inY;
    }
};

class Shape
{
public:
    Shape();
    Shape(float inX, float inY);

    virtual ~Shape();

    Point2D mCenter;

    // Adding an `= 0` to the end of a method declaration
    // signifies that this is a pure virtual method, and thus
    // makes this an abstract base class.
    virtual void Draw() = 0;
};

/// === File Separator ================snip-snip-snip==

/// Shape.cpp ------------------------------------------
#include "Shape.h"

Shape::Shape() : mCenter(0.0f, 0.0f) {}

Shape::Shape(float inX, float inY) : mCenter(inX, inY) {}

Shape::~Shape() {}
```

Circle.h/cpp

```
/// Circle.h -------------------------------------------
#pragma once

#include "Shape.h"

class Circle : public Shape
{
public:
    Circle();
    Circle(float inX, float inY, float radius);

    virtual ~Circle();

    virtual void Draw() override;

    float mRadius;
};

/// === File Separator ================snip-snip-snip==

/// Circle.cpp -----------------------------------------
#include "Circle.h"

#include <allegro5/allegro_primitives.h>

Circle::Circle() : mRadius(1.0f) {}

Circle::Circle(float inX, float inY, float radius) : Shape(inX, inY), mRadius(radius)
↪{}

Circle::~Circle() {}
```

```
void Circle::Draw()
{
    al_draw_circle(mCenter.x, mCenter.y, mRadius, al_map_rgb(255, 255, 255), 1.0f);
}
```

Rectangle.h/cpp

```
/// Rectangle.h ---------------------------------------
#pragma once
#include "Shape.h"

class Rectangle : public Shape
{
public:
    Rectangle();
    Rectangle(float inX, float inY, float width, float height);

    virtual ~Rectangle();

    virtual void Draw() override;

    float mWidth;
    float mHeight;
};

/// === File Separator ================snip-snip-snip==

/// Rectangle.cpp -------------------------------------
#include "Rectangle.h"
#include <allegro5/allegro_primitives.h>

Rectangle::Rectangle() : Shape() {}

Rectangle::Rectangle(float inX, float inY, float width, float height) : Shape(inX,␣
↪inY), mWidth(width), mHeight(height) {}

Rectangle::~Rectangle() {}

void Rectangle::Draw()
{
    al_draw_rectangle(
        mCenter.x - (mWidth / 2.0f), mCenter.y - (mHeight / 2.0f),
        mCenter.x + (mWidth / 2.0f), mCenter.y + (mHeight / 2.0f),
        al_map_rgb(255, 255, 255),
        1.0f);
}
```

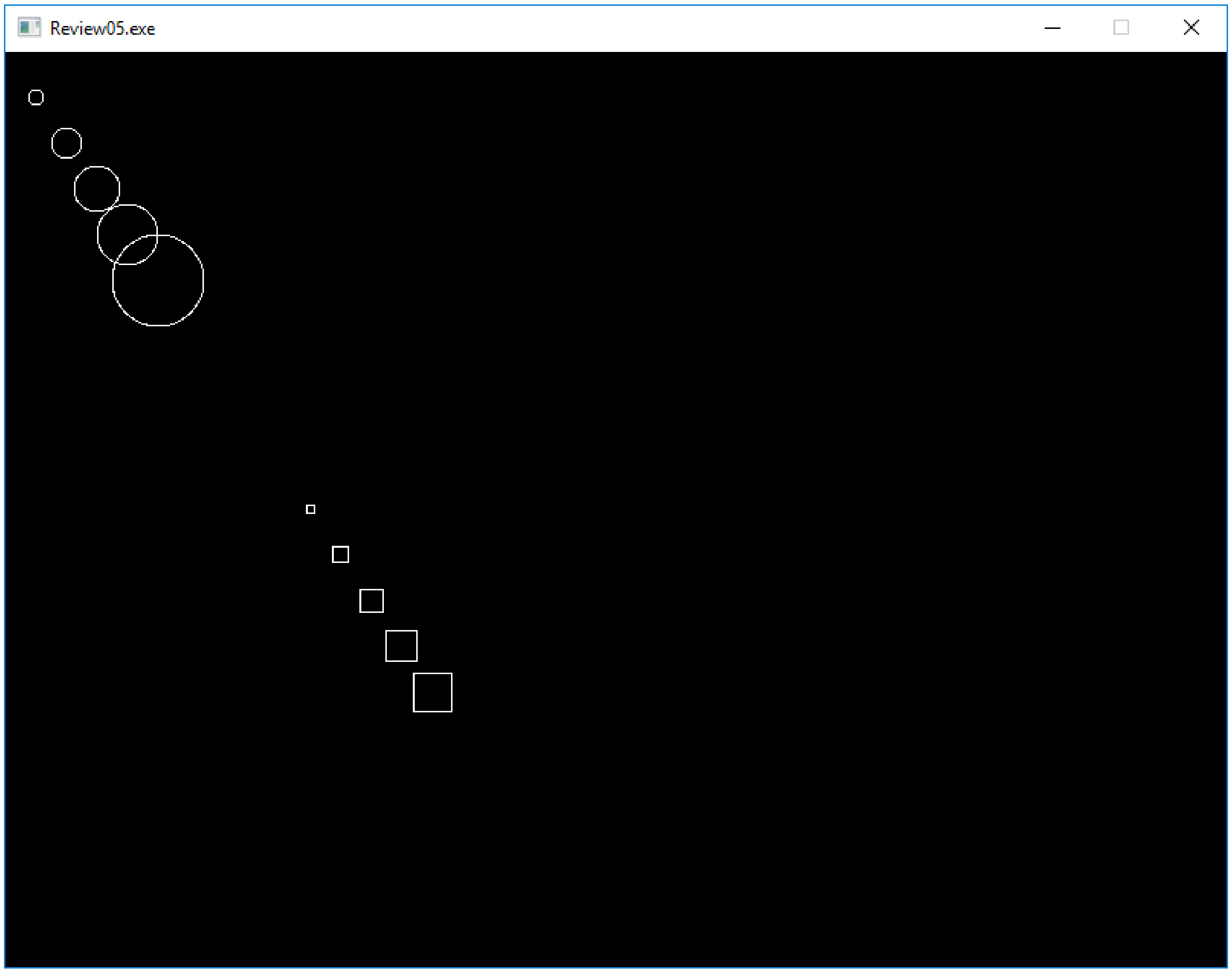
There's not very fancy with these classes. A couple of things to point out:

1. `Shape` is an abstract base class because of the pure virtual function declaration. This means we can't directly instantiate a 'Shape'
2. However, we can instantiate the derived classes (the concrete classes) `Circle` and `Rectangle`.
3. We also have `virtual` descructors in the base and in derived classes. This isn't strictly necessary right now, but let's put it in and talk about it later.

main.cpp

```
/// Main.cpp -------------------------------------------
#include <allegro5/allegro.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_primitives.h>

#include "Shape.h"
#include "Circle.h"
#include "Rectangle.h"

ALLEGRO_FONT* gFont = nullptr;

void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape* shapes[10];

    shapes[0] = new Circle(20.0f, 30.0f, 5.0f);
    shapes[1] = new Circle(40.0f, 60.0f, 10.0f);
    shapes[2] = new Circle(60.0f, 90.0f, 15.0f);
    shapes[3] = new Circle(80.0f, 120.0f, 20.0f);
    shapes[4] = new Circle(100.0f, 150.0f, 30.0f);
    shapes[5] = new Rectangle(200.0f, 300.0f, 5.0f , 5.0f);
    shapes[6] = new Rectangle(220.0f, 330.0f, 10.0f, 10.0f);
    shapes[7] = new Rectangle(240.0f, 360.0f, 15.0f, 15.0f);
    shapes[8] = new Rectangle(260.0f, 390.0f, 20.0f, 20.0f);
    shapes[9] = new Rectangle(280.0f, 420.0f, 25.0f, 25.0f);

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
    }

    al_flip_display();
    al_rest(5.0);

    al_destroy_font(gFont);
    al_destroy_display(display);
}
```

Compile that. And it works!



But it's wrong. Very, very wrong.

Note all the `new` keywords. That's memory allocations of the class `Circle` and `Rectangle`. When you `new` a class, you get back a pointer to the newly allocated object on the heap. If you remember earlier, I also said that anything you allocate from the heap, you have to clean up. So right now, we're leaking 5 `Circle` and 5 `Rectangle` objects.

How do we clean them up? We delete them, using the `delete` keyword. We end up with syntax that looks like this: `delete shapes[0];`. And we've deleted the first object we allocated! But there are 9 more to go:

```
// changes to make ...
for (int index = 0; index < 10; index++)
{
    shapes[index]->Draw();
}

al_flip_display();
al_rest(5.0);

// Add this!
delete shapes[0];
delete shapes[1];
delete shapes[2];
delete shapes[3];
delete shapes[4];
delete shapes[5];
delete shapes[6];
delete shapes[7];
delete shapes[8];
delete shapes[9];
```

And we now are no longer leaking memory!

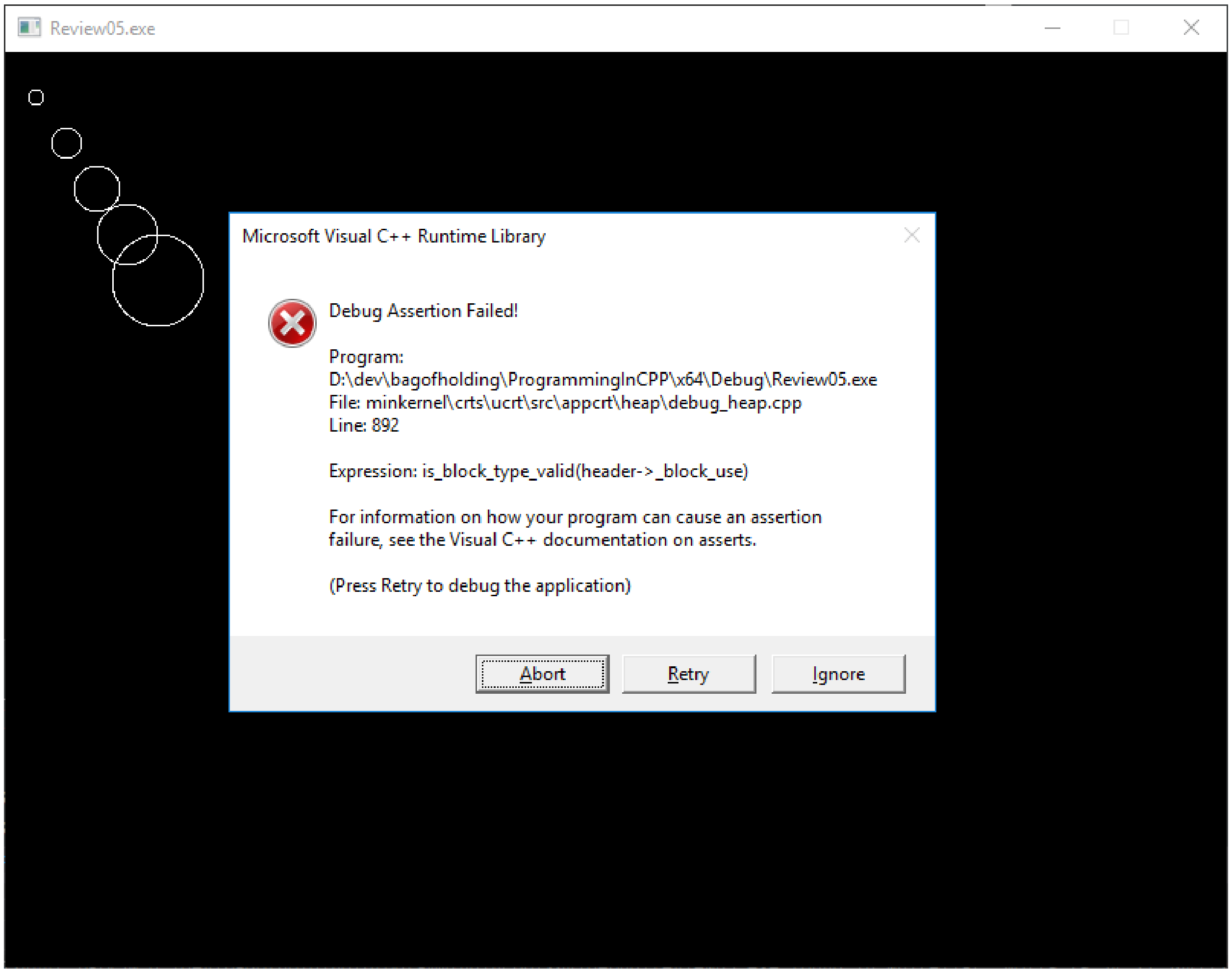
But that's pretty hokey to have to track and delete objects like that. What if we wanted to allocate an arbitrary number of shapes? That current strategy isn't going to work. So what do we do? C++ comes with an `operator delete[]` C++ new operator link

> operator delete[] is a regular function that can be called explicitly just as any other function. But in C++, delete[] is an operator with a very specific behavior: An expression with the delete[] operator, first calls the appropriate destructors for each element in the array (if these are of a class type), and then calls an array deallocation function.

Well that sure sounds like an appropriate replacement now, doesn't it? Let's try it:

```
for (int index = 0; index < 10; index++)
{
    shapes[index]->Draw();
}

al_flip_display();
al_rest(5.0);

// use this instead!
delete[] shapes;
```

That compiles no problem. Let's run it!



```
Debug Assertion Failed!

Program:
D:\dev\bagofholding\ProgrammingInCPP\x64\Debug\Review05.exe
File: minkernel\crts\ucrt\src\appcrt\heap\debug_heap.cpp
Line: 892

Expression: is_block_type_valid(header->_block_use)

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

(Press Retry to debug the application)
```

`Expression: is_block_type_valid(header->_block_use)` ... what in the world does that mean. You can go ahead and google that if you want to, but I don't think it'll give you back anything good.

So what's going on. The definition from the cpluscplus website says that we're deleting the array!

But are we? Check out the code again, but this time with an eye on how we create the array:

```
Shape* shapes[10];
```

What does that mean, exactly? Reading it from right to left:

We're creating a 10 element array of `Shape` pointers (that's what the `*` means).

OK, we're creating an array of pointers. And that's what we're trying to delete, are the pointers in the array!

But hang on a minute there, cowboy. Check out the definition of what `delete[]` actually does:

> An expression with the delete[] operator, first calls the appropriate destructors for each element in the array (if these are of a class type), *and then calls an array deallocation function*.

But that's what we're doing there. We are deleting the array that we created there. Or are we?

Here's how we create the array:

```
Shape* shapes[10];
```

That's an array of 10 `Shape` pointers. So `delete []` has got to work.

Let me ask you a question. What's the difference between these two lines:

```
Shape* shapes[10];
int numbers[10];
```

One of those lines creates an array of `Shape` pointers. The other is an array of `int``s. And where do these arrays live? If you're first response is "on the stack", they you win a no-prize! Arrays of this sort are allocated on the stack. You cannot deallocate (``delete`) from items created on the stack! And that's what we're seeing in the error. What we *have* allocated on the heap are the objects the slots in the `shapes` array are pointing to. So we *do* have to iterate over all the elements in the list to delete them. We *can* clean up the code to be a little less icky:

```
for (int index = 0; index < 10; index++)
{
    delete shapes[index];
}
```

That solves the problem, but it still leaves us with "how do we use *delete[]*". Well, we actually have to create the array on the heap.

```
// Shape* shapes[10];
Shape** shapes = new Shape*[10];  // This allocates a block of memory big enough for␣
↪10 shape pointers

// snip

al_flip_display();
al_rest(5.0);

delete[] shapes;  // <== New bit of code

al_destroy_font(gFont);
al_destroy_display(display);
```

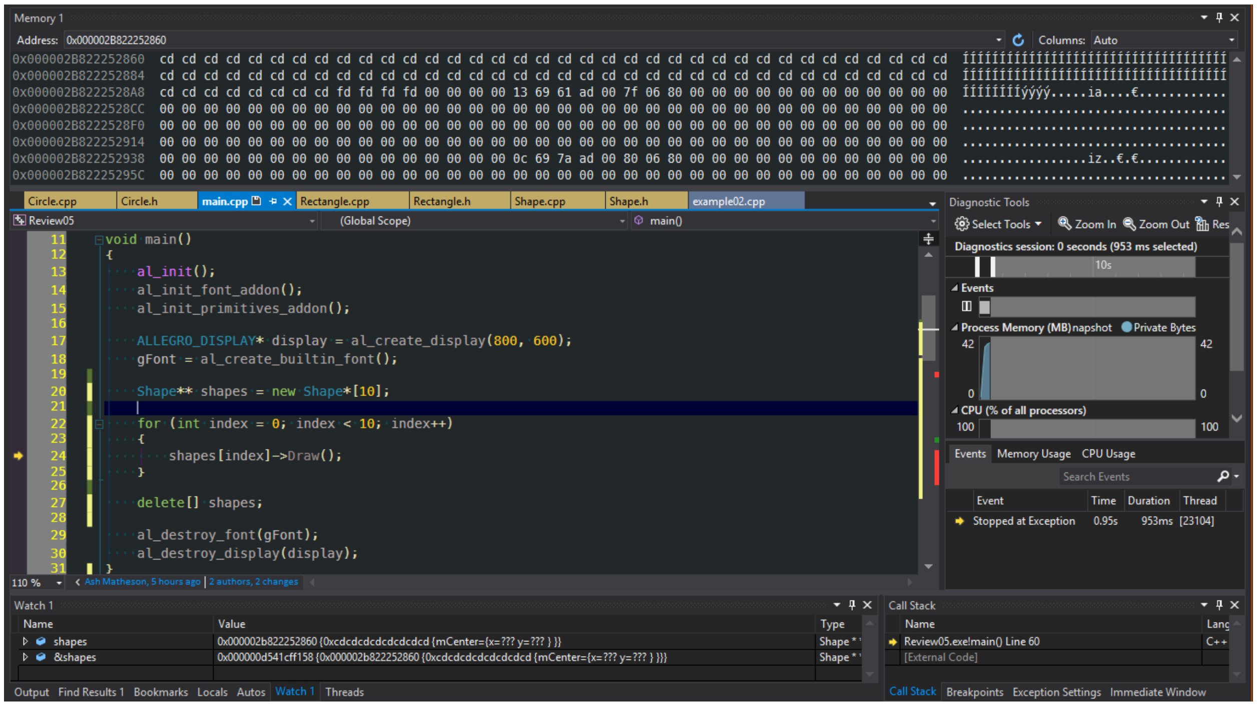
Let me ask you a question ...

What would happen if my `main` function looked like this:

```
Shape** shapes = new Shape*[10];

for (int index = 0; index < 10; index++)
{
    shapes[index]->Draw();
}

delete[] shapes;
```

Hopefully you'd expect that not to work. First off, we've not actually put anything into that array. But what's in that memory block?

Have I got a picture for you:

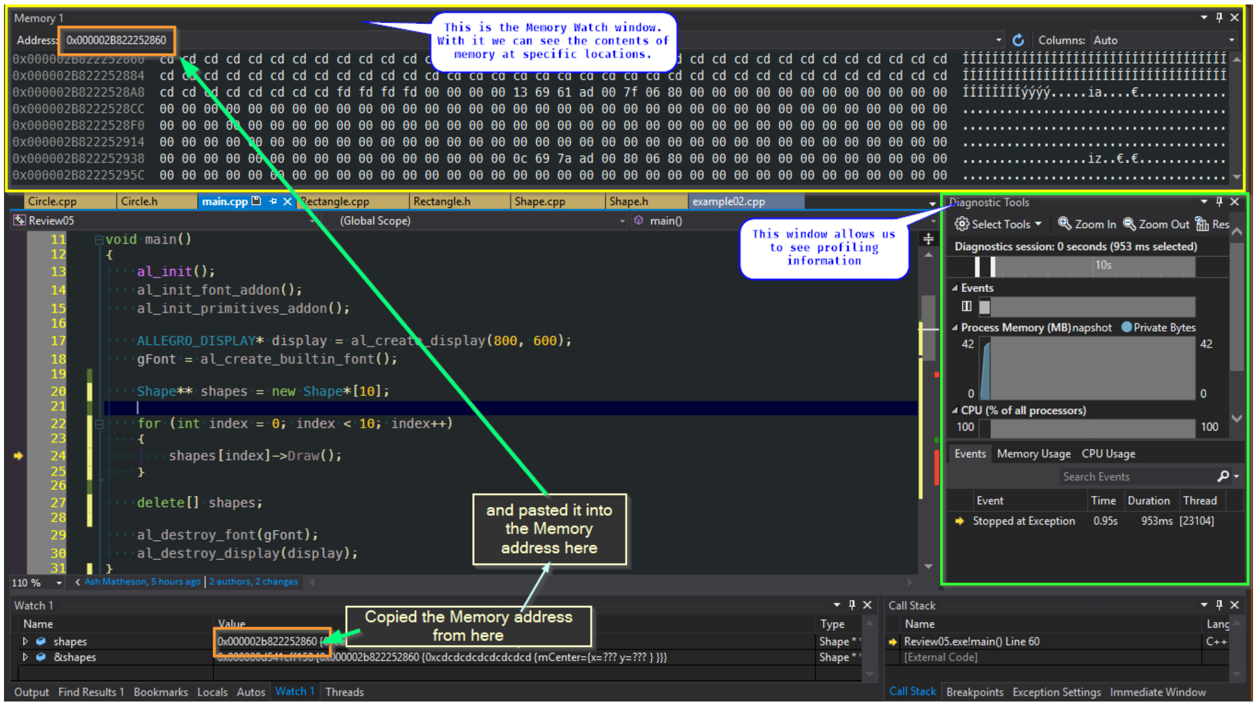


```
Memory 1
Address: 0x000002B822252860                                                                                       Columns: Auto
0x000002B822252860  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
0x000002B822252884  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
0x000002B8222528A8  cd cd cd cd cd cd cd cd fd fd fd fd 00 00 00 00 13 69 61 ad 00 7f 06 80 00 00 00 00 00 00 00 00 00 00 00 00  ÍÍÍÍÍÍÍÍýýýý.....ia....€...........
0x000002B8222528CC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....................................
0x000002B8222528F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....................................
0x000002B822252914  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....................................
0x000002B822252938  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0c 69 7a ad 00 80 06 80 00 00 00 00 00 00 00 00 00 00 00 00  ................iz..€.€...........
0x000002B82225295C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....................................
```

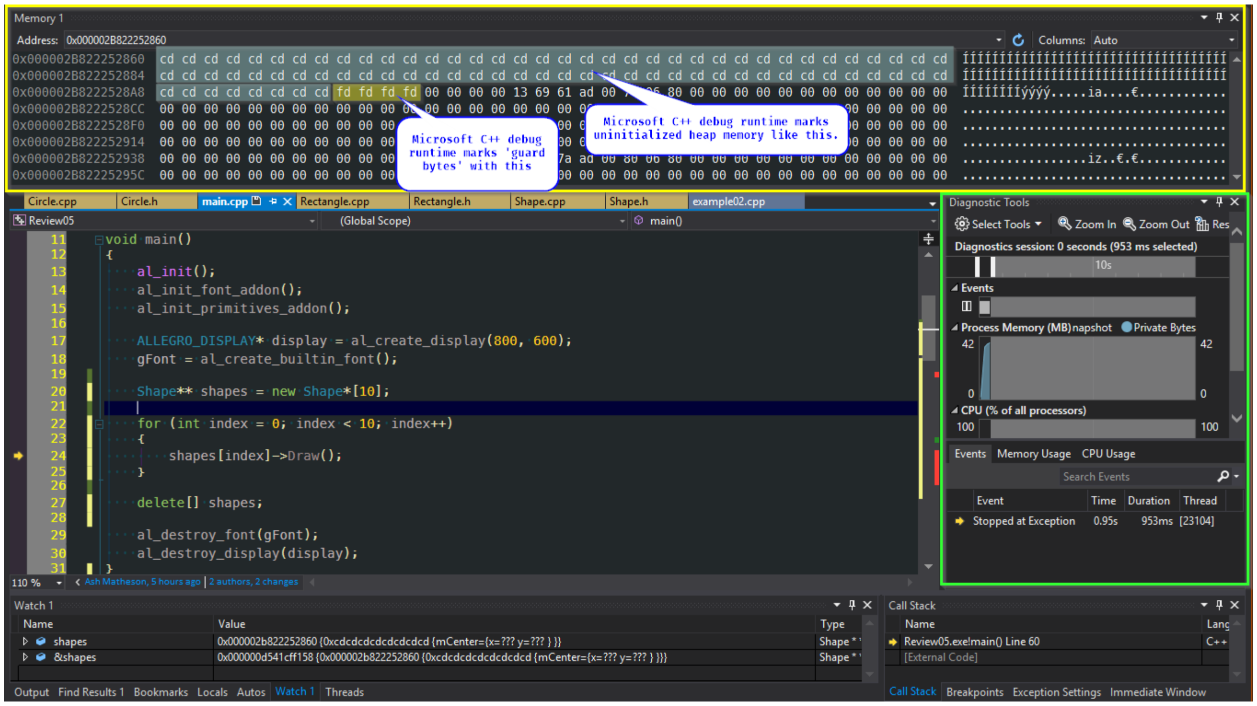
```cpp
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
    }

    delete[] shapes;

    al_destroy_font(gFont);
    al_destroy_display(display);
}
```

```
Watch 1
Name      Value                                                                                 Type
shapes    0x000002b822252860 {0xcdcdcdcdcdcdcdcd {mCenter={x=??? y=??? } }}                     Shape *
&shapes   0x000000d541cff158 {0x000002b822252860 {0xcdcdcdcdcdcdcdcd {mCenter={x=??? y=??? } }}} Shape *

Call Stack
Name                            Lang
Review05.exe!main() Line 60     C++
[External Code]

Diagnostic Tools
Diagnostics session: 0 seconds (953 ms selected)
Event                  Time    Duration  Thread
Stopped at Exception   0.95s   953ms     [23104]
```

Got a lot going on there. So the next image breaks down what we're seeing:


This is the Memory Watch window. With it we can see the contents of memory at specific locations.
This window allows us to see profiling information
and pasted it into the Memory address here
Copied the Memory address from here

So, in the memory watch window, there's a lot of `CD` and `FD` values. What do they mean?


Microsoft C++ debug runtime marks 'guard bytes' with this
Microsoft C++ debug runtime marks uninitialized heap memory like this.

```
Address: 0x000002B822252860
0x000002B822252860  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
0x000002B822252884  cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd cd  ÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ
0x000002B8222528A8  cd cd cd cd cd cd cd cd fd fd fd fd 00 00 00 00 13 69 61 ad 00 [illegible] 06 80 00 00 00 00 00 00 00 00 00 00 00  ÍÍÍÍÍÍÍÍýýýý.....ia....€...........
0x000002B8222528CC  00 00 00 00 00 00 00 00 00 00 00 [illegible] 00 00 00 00 00  ....................................
0x000002B8222528F0  00 00 00 00 00 00 00 00 00 00 00 [illegible] 00 00 00 00 00  ....................................
0x000002B822252914  00 00 00 00 00 00 00 00 00 00 00 [illegible] 00 00 00 00 00  ....................................
0x000002B822252938  00 00 00 00 00 00 00 00 00 00 00 [illegible] 7a ad 00 80 06 80 00 00 00 00 00 00 00 00 00 00 00 00 00  ................iz..€.€...........
0x000002B82225295C  00 00 00 00 00 00 00 00 00 00 00 [illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ....................................
```

```cpp
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
    }

    delete[] shapes;

    al_destroy_font(gFont);
    al_destroy_display(display);
}
```

Name	Value	Type
shapes	0x000002b822252860 {0xcdcdcdcdcdcdcdcd {mCenter={x=??? y=??? } }}	Shape *
&shapes	0x000000d541cff158 {0x000002b822252860 {0xcdcdcdcdcdcdcdcd {mCenter={x=??? y=??? } }}}	Shape *

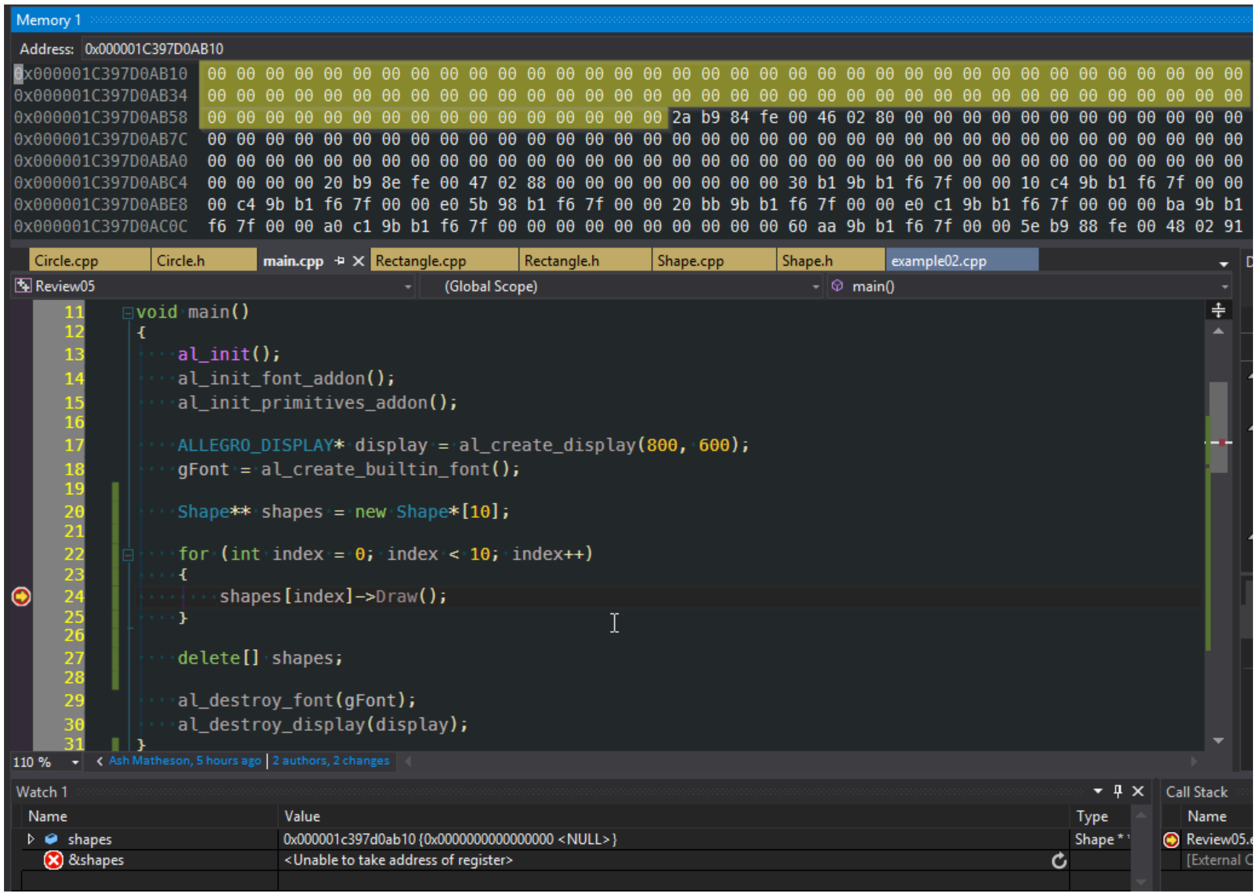
Name	Lang
Review05.exe!main() Line 60	C++
[External Code]	

Event	Time	Duration	Thread
Stopped at Exception	0.95s	953ms	[23104]

Here’s a handy little table to refer to what some common values are:

nobug.org’s CRT Heap Table

That’s cool, but that’s not the whole story. We’re currently running in Debug. What happens when we run in release?



```
Address: 0x000001C397D0AB10
0x000001C397D0AB10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C397D0AB34  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C397D0AB58  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2a b9 84 fe 00 46 02 80 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C397D0AB7C  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C397D0ABA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000001C397D0ABC4  00 00 00 00 20 b9 8e fe 00 47 02 88 00 00 00 00 00 00 00 00 30 b1 9b b1 f6 7f 00 00 10 c4 9b b1 f6 7f 00 00
0x000001C397D0ABE8  00 c4 9b b1 f6 7f 00 00 e0 5b 98 b1 f6 7f 00 00 20 bb 9b b1 f6 7f 00 00 e0 c1 9b b1 f6 7f 00 00 00 ba 9b b1
0x000001C397D0AC0C  f6 7f 00 00 a0 c1 9b b1 f6 7f 00 00 00 00 00 00 00 00 00 00 60 aa 9b b1 f6 7f 00 00 5e b9 88 fe 00 48 02 91
```

```cpp
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
    }

    delete[] shapes;

    al_destroy_font(gFont);
    al_destroy_display(display);
}
```

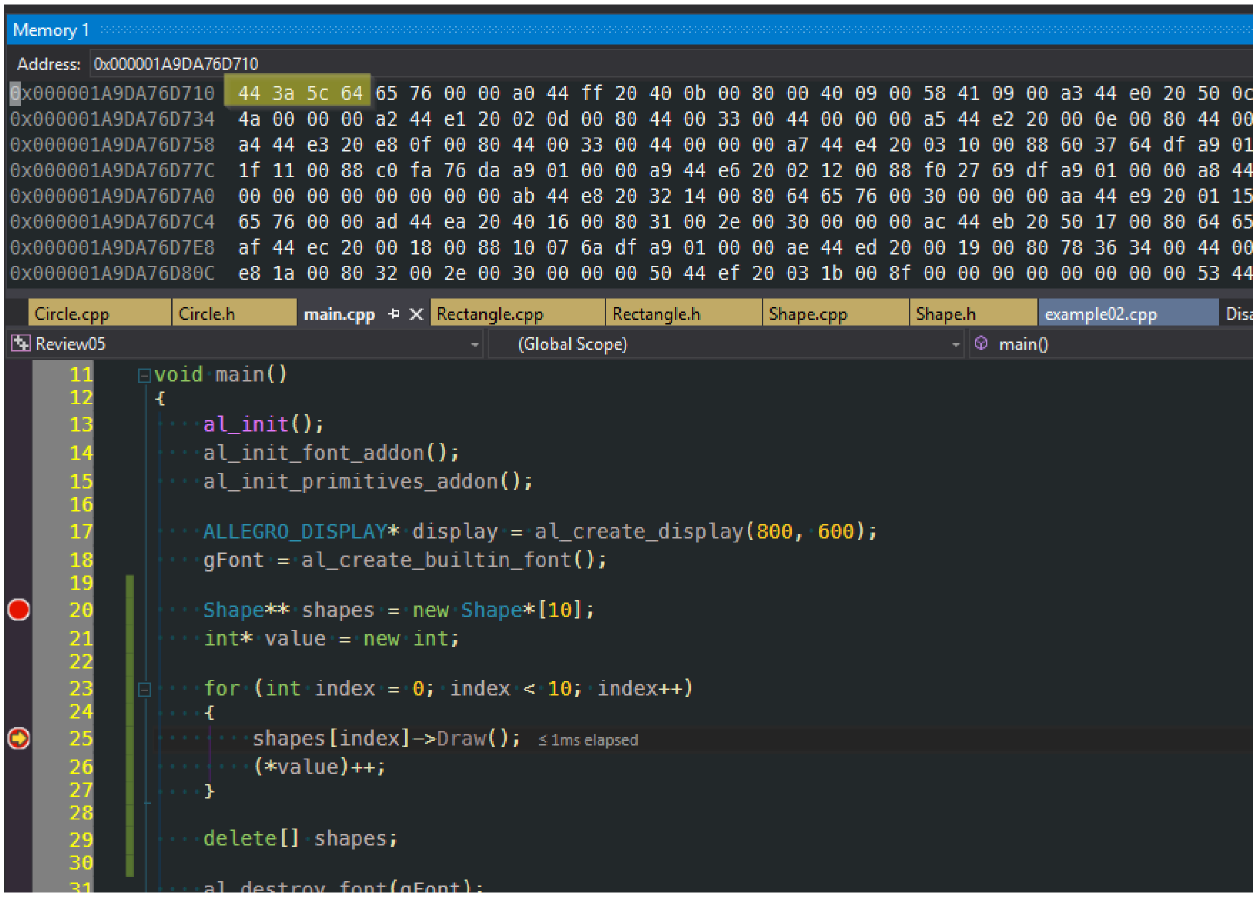
Watch 1

Name	Value	Type
shapes	0x000001c397d0ab10 {0x0000000000000000 <NULL>}	Shape * *
&shapes	<Unable to take address of register>	

Ahh, it's all zeros. That's good. Or is it?

New isn't supposed to 'clear' memory - it should just be grabbing a block of memory that isn't in use and blocking it off. So I would expect random data in that.

Let's try something simple,



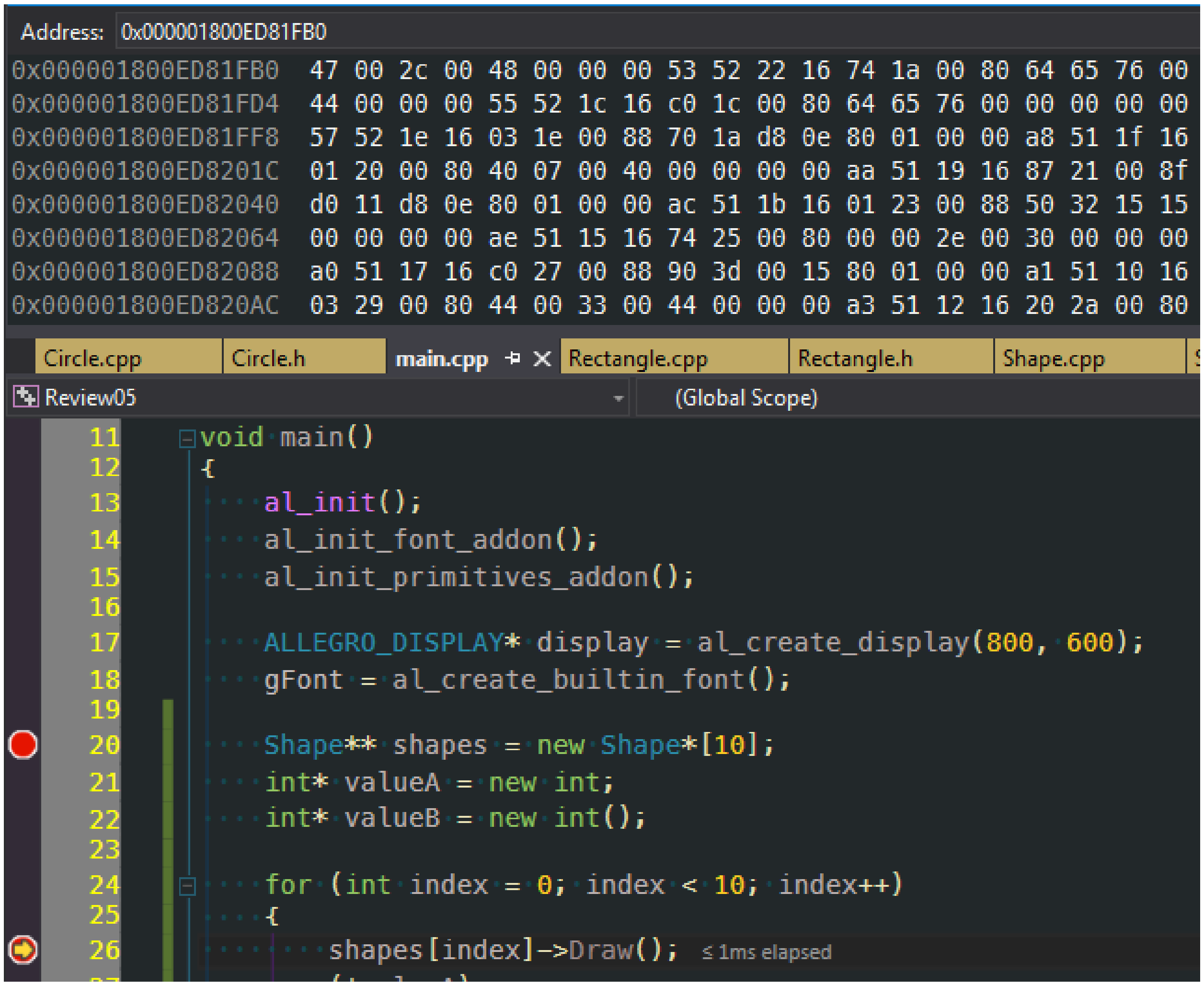
```
0x000001A9DA76D710  44 3a 5c 64 65 76 00 00 a0 44 ff 20 40 0b 00 80 00 40 09 00 58 41 09 00 a3 44 e0 20 50 0c
0x000001A9DA76D734  4a 00 00 00 a2 44 e1 20 02 0d 00 80 44 00 33 00 44 00 00 00 a5 44 e2 20 00 0e 00 80 44 00
0x000001A9DA76D758  a4 44 e3 20 e8 0f 00 80 44 00 33 00 44 00 00 00 a7 44 e4 20 03 10 00 88 60 37 64 df a9 01
0x000001A9DA76D77C  1f 11 00 88 c0 fa 76 da a9 01 00 00 a9 44 e6 20 02 12 00 88 f0 27 69 df a9 01 00 00 a8 44
0x000001A9DA76D7A0  00 00 00 00 00 00 00 00 ab 44 e8 20 32 14 00 80 64 65 76 00 30 00 00 00 aa 44 e9 20 01 15
0x000001A9DA76D7C4  65 76 00 00 ad 44 ea 20 40 16 00 80 31 00 2e 00 30 00 00 00 ac 44 eb 20 50 17 00 80 64 65
0x000001A9DA76D7E8  af 44 ec 20 00 18 00 88 10 07 6a df a9 01 00 00 ae 44 ed 20 00 19 00 80 78 36 34 00 44 00
0x000001A9DA76D80C  e8 1a 00 80 32 00 2e 00 30 00 00 00 50 44 ef 20 03 1b 00 8f 00 00 00 00 00 00 00 00 53 44
```

```cpp
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];
    int* value = new int;

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
        (*value)++;
    }

    delete[] shapes;

    al_destroy_font(gFont);
```

So to test our theory, we're allocating a single `int` on the heap. Taking a look at the memory block, we get the result we expect (that is, an uninitialized block of memory).

This is getting frustrating, isn't it?

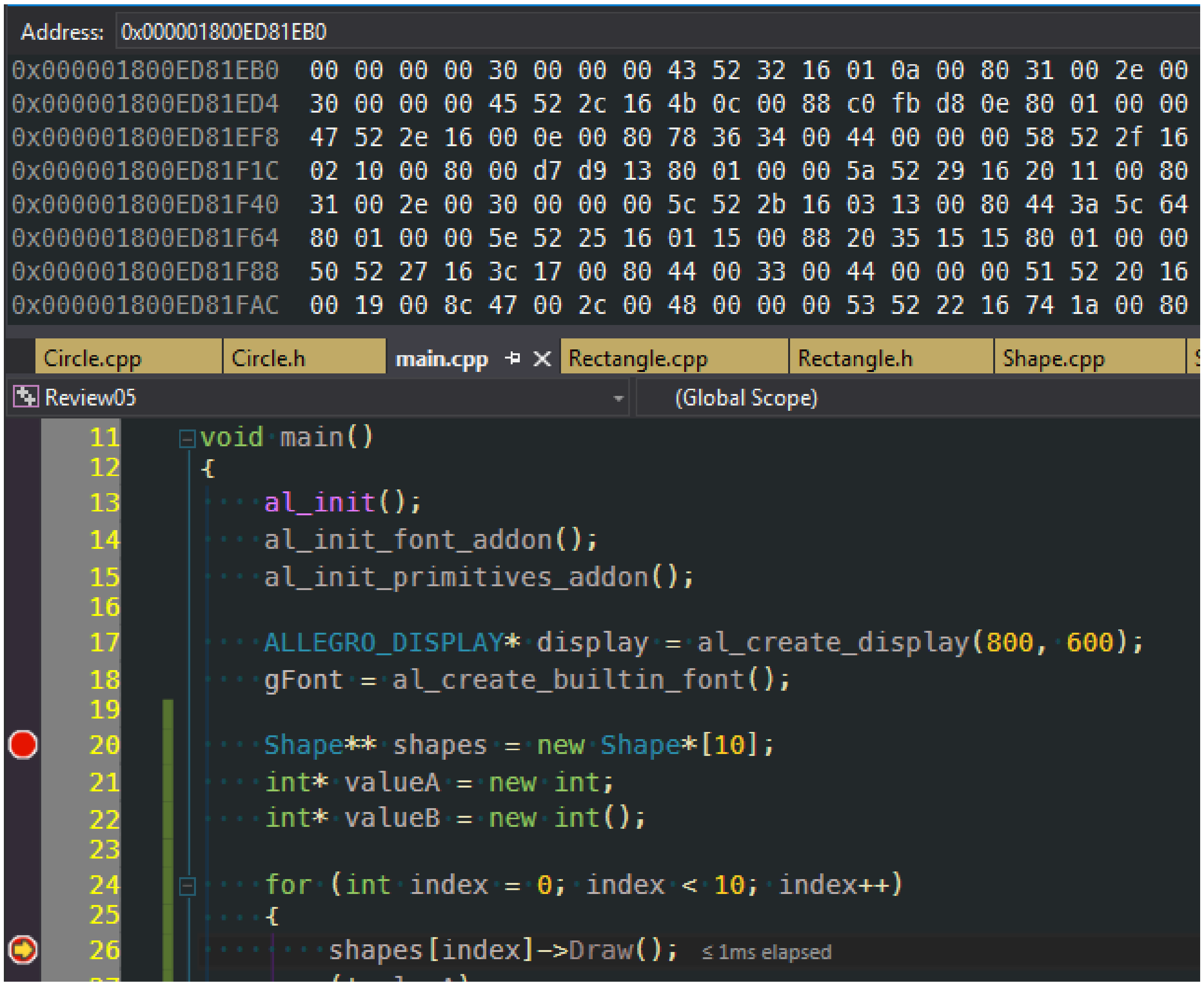
Let's take a look at this:



```
Address: 0x000001800ED81FB0
0x000001800ED81FB0  47 00 2c 00 48 00 00 00 53 52 22 16 74 1a 00 80 64 65 76 00
0x000001800ED81FD4  44 00 00 00 55 52 1c 16 c0 1c 00 80 64 65 76 00 00 00 00 00
0x000001800ED81FF8  57 52 1e 16 03 1e 00 88 70 1a d8 0e 80 01 00 00 a8 51 1f 16
0x000001800ED8201C  01 20 00 80 40 07 00 40 00 00 00 00 aa 51 19 16 87 21 00 8f
0x000001800ED82040  d0 11 d8 0e 80 01 00 00 ac 51 1b 16 01 23 00 88 50 32 15 15
0x000001800ED82064  00 00 00 00 ae 51 15 16 74 25 00 80 00 00 2e 00 30 00 00 00
0x000001800ED82088  a0 51 17 16 c0 27 00 88 90 3d 00 15 80 01 00 00 a1 51 10 16
0x000001800ED820AC  03 29 00 80 44 00 33 00 44 00 00 00 a3 51 12 16 20 2a 00 80
```

```
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];
    int* valueA = new int;
    int* valueB = new int();

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw(); ≤ 1ms elapsed
```

We're seeing in that slide that `int* valueA = new int;` doesn't clear memory. However ...



```
Address: 0x000001800ED81EB0
0x000001800ED81EB0  00 00 00 00 30 00 00 00 43 52 32 16 01 0a 00 80 31 00 2e 00
0x000001800ED81ED4  30 00 00 00 45 52 2c 16 4b 0c 00 88 c0 fb d8 0e 80 01 00 00
0x000001800ED81EF8  47 52 2e 16 00 0e 00 80 78 36 34 00 44 00 00 00 58 52 2f 16
0x000001800ED81F1C  02 10 00 80 00 d7 d9 13 80 01 00 00 5a 52 29 16 20 11 00 80
0x000001800ED81F40  31 00 2e 00 30 00 00 00 5c 52 2b 16 03 13 00 80 44 3a 5c 64
0x000001800ED81F64  80 01 00 00 5e 52 25 16 01 15 00 88 20 35 15 15 80 01 00 00
0x000001800ED81F88  50 52 27 16 3c 17 00 80 44 00 33 00 44 00 00 00 51 52 20 16
0x000001800ED81FAC  00 19 00 8c 47 00 2c 00 48 00 00 00 53 52 22 16 74 1a 00 80
```

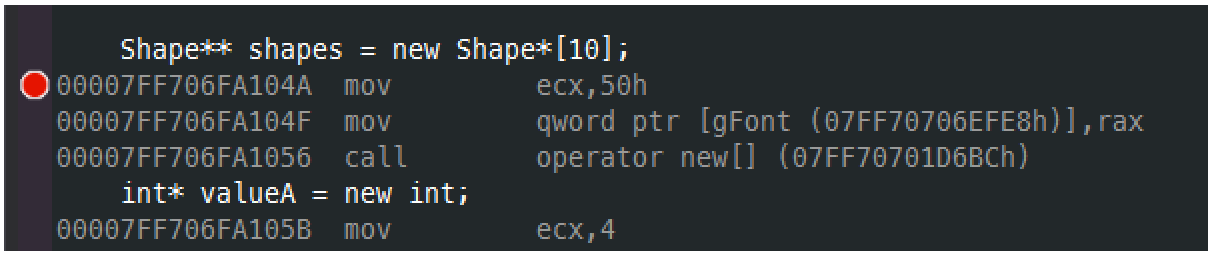
```
void main()
{
    al_init();
    al_init_font_addon();
    al_init_primitives_addon();

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new Shape*[10];
    int* valueA = new int;
    int* valueB = new int();

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();  ≤ 1ms elapsed
```

In this case, `int* valueB = new int();` *does* clear the block of memory! From Stack Overflow here we can see that the C++ 11 standard it will clear the allocated block to 0.

BUT THAT'S NOT WHAT WE'RE DOING IN OUR ARRAY!

I'm going to add in the `()` to our `Shape` array creation:

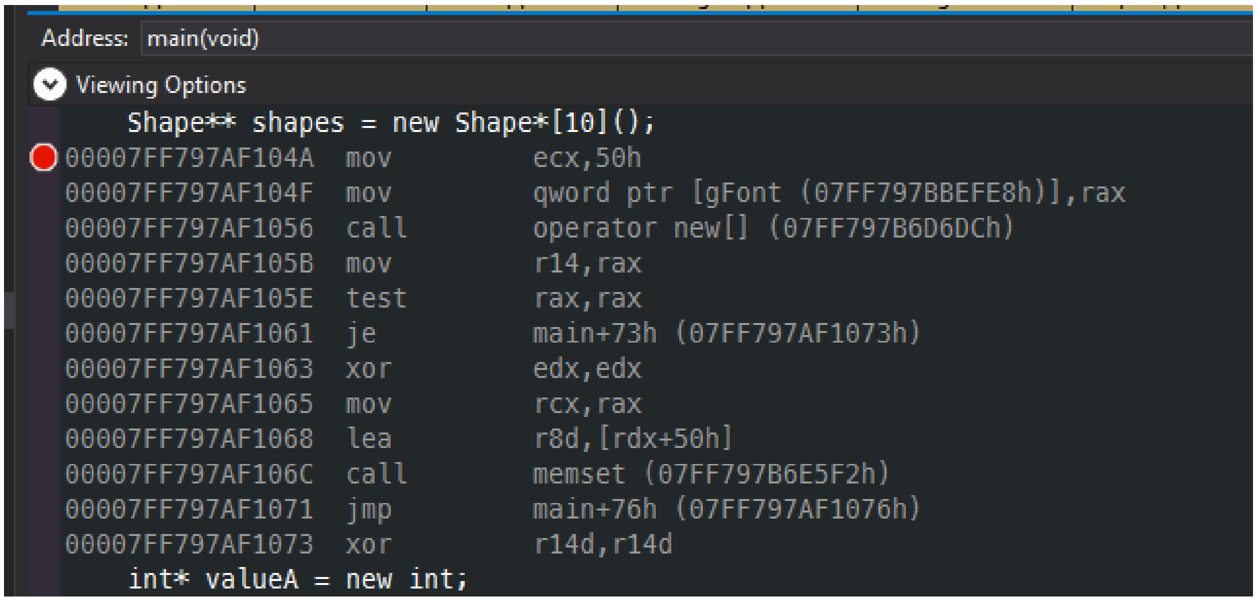
eg: it was:



```
    Shape** shapes = new Shape*[10];
00007FF706FA104A  mov         ecx,50h
00007FF706FA104F  mov         qword ptr [gFont (07FF70706EFE8h)],rax
00007FF706FA1056  call        operator new[] (07FF70701D6BCh)
    int* valueA = new int;
00007FF706FA105B  mov         ecx,4
```

three lines of assembly. All it does is call operator `new[]`.

VS:


Address: main(void)

Viewing Options

```
    Shape** shapes = new Shape*[10]();
00007FF797AF104A  mov         ecx,50h
00007FF797AF104F  mov         qword ptr [gFont (07FF797BBEFE8h)],rax
00007FF797AF1056  call        operator new[] (07FF797B6D6DCh)
00007FF797AF105B  mov         r14,rax
00007FF797AF105E  test        rax,rax
00007FF797AF1061  je          main+73h (07FF797AF1073h)
00007FF797AF1063  xor         edx,edx
00007FF797AF1065  mov         rcx,rax
00007FF797AF1068  lea         r8d,[rdx+50h]
00007FF797AF106C  call        memset (07FF797B6E5F2h)
00007FF797AF1071  jmp         main+76h (07FF797AF1076h)
00007FF797AF1073  xor         r14d,r14d
    int* valueA = new int;
```

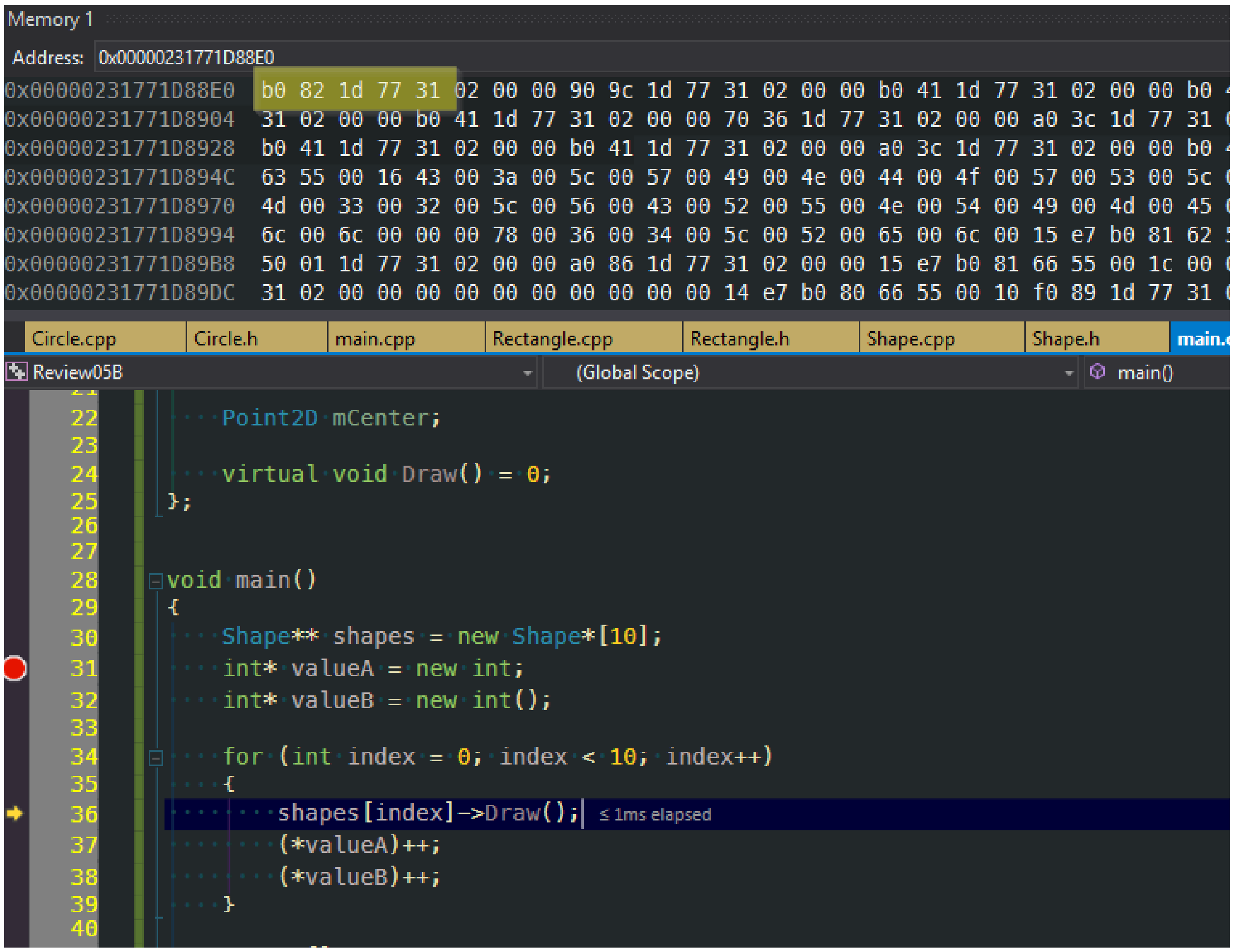
And it does significantly more work! The call to `memset` should make you think that it's setting the block to all zeros.

So ... Is that's what's happening here? Is C++ aliasing our call to this? Well, there's one more test we can do. We write a quick, sample console app to test this all again! With that in mind, I've writting `Review05B`.

And here's what it looks like:

```
struct Point2D
{
    float x;
    float y;

    Point2D(float inX, float inY)
    {
        x = inX;
        y = inY;
    }
};

class Shape
{
public:
    Shape();
    Shape(float inX, float inY);

    virtual ~Shape();

    Point2D mCenter;

    virtual void Draw() = 0;
};


void main()
{
    Shape** shapes = new Shape*[10];
    int* valueA = new int;
    int* valueB = new int();
    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
        (*valueA)++;
        (*valueB)++;
    }

    delete[] shapes;
}
```

And our result?



```
Memory 1
Address: 0x00000231771D88E0
0x00000231771D88E0  b0 82 1d 77 31 02 00 00 90 9c 1d 77 31 02 00 00 b0 41 1d 77 31 02 00 00 b0
0x00000231771D8904  31 02 00 00 b0 41 1d 77 31 02 00 00 70 36 1d 77 31 02 00 00 a0 3c 1d 77 31
0x00000231771D8928  b0 41 1d 77 31 02 00 00 b0 41 1d 77 31 02 00 00 a0 3c 1d 77 31 02 00 00 b0
0x00000231771D894C  63 55 00 16 43 00 3a 00 5c 00 57 00 49 00 4e 00 44 00 4f 00 57 00 53 00 5c
0x00000231771D8970  4d 00 33 00 32 00 5c 00 56 00 43 00 52 00 55 00 4e 00 54 00 49 00 4d 00 45
0x00000231771D8994  6c 00 6c 00 00 00 78 00 36 00 34 00 5c 00 52 00 65 00 6c 00 15 e7 b0 81 62
0x00000231771D89B8  50 01 1d 77 31 02 00 00 a0 86 1d 77 31 02 00 00 15 e7 b0 81 66 55 00 1c 00
0x00000231771D89DC  31 02 00 00 00 00 00 00 00 00 00 00 14 e7 b0 80 66 55 00 10 f0 89 1d 77 31
```

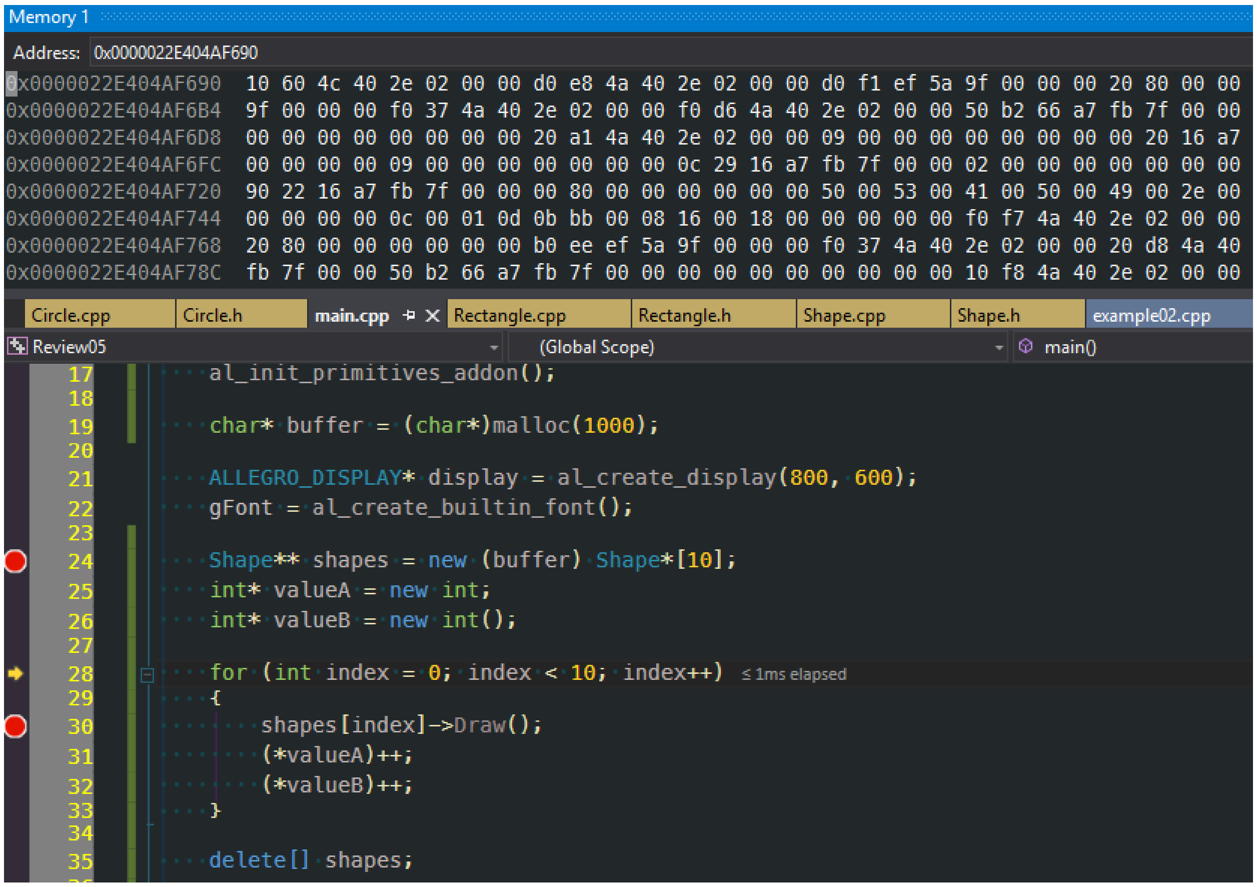
```
    Point2D mCenter;

    virtual void Draw() = 0;
};


void main()
{
    Shape** shapes = new Shape*[10];
    int* valueA = new int;
    int* valueB = new int();

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();  ≤ 1ms elapsed
        (*valueA)++;
        (*valueB)++;
    }
```

OK, so what the heck is going on here? This is what I'm expecting to see. This is the normal behaviour that makes sense to me.

I'm still not sure, but I have some guesses.

1. Allegro has, for some reason, overridden the operator new. This is a highly, highly suspect assumption.
2. It is very possible that I'm "Just getting (un)lucky". `new` is returning me block of memory that are already nulled.

Both situations are possible. I'm starting to lean toward #2, myself. Mostly because of being able to do `placement new` and this bit of code:



```
0x0000022E404AF690  10 60 4c 40 2e 02 00 00 d0 e8 4a 40 2e 02 00 00 d0 f1 ef 5a 9f 00 00 00 20 80 00 00
0x0000022E404AF6B4  9f 00 00 00 f0 37 4a 40 2e 02 00 00 f0 d6 4a 40 2e 02 00 00 50 b2 66 a7 fb 7f 00 00
0x0000022E404AF6D8  00 00 00 00 00 00 00 00 20 a1 4a 40 2e 02 00 00 09 00 00 00 00 00 00 00 00 20 16 a7
0x0000022E404AF6FC  00 00 00 00 09 00 00 00 00 00 00 00 0c 29 16 a7 fb 7f 00 00 02 00 00 00 00 00 00 00
0x0000022E404AF720  90 22 16 a7 fb 7f 00 00 00 80 00 00 00 00 00 00 50 00 53 00 41 00 50 00 49 00 2e 00
0x0000022E404AF744  00 00 00 00 0c 00 01 0d 0b bb 00 08 16 00 18 00 00 00 00 00 f0 f7 4a 40 2e 02 00 00
0x0000022E404AF768  20 80 00 00 00 00 00 00 b0 ee ef 5a 9f 00 00 00 f0 37 4a 40 2e 02 00 00 20 d8 4a 40
0x0000022E404AF78C  fb 7f 00 00 50 b2 66 a7 fb 7f 00 00 00 00 00 00 00 00 00 00 10 f8 4a 40 2e 02 00 00
```

```
    al_init_primitives_addon();

    char* buffer = (char*)malloc(1000);

    ALLEGRO_DISPLAY* display = al_create_display(800, 600);
    gFont = al_create_builtin_font();

    Shape** shapes = new (buffer) Shape*[10];
    int* valueA = new int;
    int* valueB = new int();

    for (int index = 0; index < 10; index++)
    {
        shapes[index]->Draw();
        (*valueA)++;
        (*valueB)++;
    }

    delete[] shapes;
```

Placement `new` allows me to pre-allocate a block of memory and use that as my 'pool' to draw from. In this case I use `malloc` to allocate 1000 bytes and stuff it into `buffer`. Then, using

```
Shape** shapes = new (buffer) Shape*[10];
```

I can pull from that block. And from the image you can see that it's random data (also, the address of `buffer` and `shapes` is the same, but it's not shown in the slide).

Holy crap, that was a long, long review. And I really want to go into more details. But I think that's enough for now.

That was quite the review. And it's debatable that I've covered everything that would be considered 'a review'. But I feel that's enough 'basics' to cover for now. It's very possible I'll come back and add/revise this in the future, but as a starting point I consider the review over.

Pointers in C++

Summary

This is a review of sorts, but I'm going to dig much deeper into memory specifics of the C family of languages. Thus I felt it was far more worthwhile to break it into it's own group of projects.

For the most part, I'll be sticking with Visual Studio, rather than the C++ shell for the code samples; we want to be looking at memory and Visual Studio has much better tools for that.

With that said, I forsee the majority of these projects being console applications. I want these to potentially be portable fairly easily to other compilers/OSes. Making them Windows applications will hamper that. I also will be avoiding using Allegro this time around as I want to keep us focused on memory and pointers.

Why are we looking at pointers?

If you come from other languages like C#, Java, Python, you really haven't had to worry a lot about memory, from a truly lower level standpoint. However, with C++, you *will* be very interested in what's happening with memory.

Why's that? It's pretty simple, actually:

> C++ is horrifically bad at dealing with memory.

What??

If you come from any of those other languages, you will, without a doubt, believe that what I've just written is the gospel. So why is that?

Let's look at an incredibly simple C# example:

```
using System;

public class Program
{
    public class Shape
    {
        public Shape()
        {
            Console.WriteLine("  -> Shape Constructor");
        }

        ~Shape()
        {
            Console.WriteLine("  -> Shape Destructor");
        }

        public void Draw()
        {
            Console.WriteLine("  -> Shape Draw");
        }
    }

    public static void Main()
    {
        Shape A = new Shape();
    }
}
```

.Net Fiddle version

How much are we leaking there? If you said "none", you're correct. C# will automatically Garbage collect `A` when it's no longer being used.

The same thing in C++?

```
// Example program
#include <stdio.h>

class Shape
{
public:
    Shape()
    {
        printf("  -> Shape Constructor\n");
    }

    ~Shape()
    {
        printf("  -> Shape Destructort\n");
    }

    void Draw()
    {
        printf("  -> Drawing a shape\n");
    }
};

int main()
{
    Shape* A = new Shape();
}
```

C++ shell version of leaking candidate

How much are we leaking there? I'll give it to you in a percentage: 100% leaky.

Yes, fixing that would be trivial; just add a `delete A;` before main finishes. But that's no magic bullet. Let me show you something:

```
// Example program
#include <stdio.h>

class Shape
{
public:
    Shape()
    {
        printf("  -> Shape Constructor\n");
    }

    ~Shape()
    {
        printf("  -> Shape Destructort\n");
    }

    void Draw()
    {
        printf("  -> Drawing a shape\n");
    }
};

int main()
{
    Shape* A = new Shape();
    Shape* B = A;

    A->Draw();
    B->Draw();
    delete A;

    B->Draw();
}
```

What happens there?

Drawing a shape - C++ shell

output:

```
-> Shape Constructor
-> Drawing a shape
-> Drawing a shape
-> Shape Destructort
-> Drawing a shape
```

Looks good to me! Next!

Suddenly, a stranger appears next to you, eyeing your computer

"Whoa up there pardner! I think you've got some cattle rustler's in your code there"

Startled, you sit upright in your chair. You could have sworn you locked the front door. Heart racing, you stammer a shocked response.

"Who the hell are you?"

"Why, I'm the Allocator Kidd. And I think your memory's being rustled up but good! Do y'mind if'n I take the reigns there?"

He pulls off his leather cowboy gloves, revealing weathered, leathery hands, with millimeter thick callusses cover the tips of each finger, undoubtably from years of being a keyboard jockey in the wild west.

You stand up, tenatively offering your chair. With a sly grin he nods, taking a seat, roughly pulling the keyboard into his lap. With a side glance he looks at your lighted mouse and mutters to himself

"A mouse. How *quaint*"

He writes two lines of code, smiles and becons you to look at what he's written.

```
int main()
{
    Shape* A = new Shape();
    Shape* B = A;

    printf("How big is a Shape? %lu\n", sizeof(A));
    printf("Is that really how big a Shape is? %lu\n", sizeof(*A));

    A->Draw();
    B->Draw();
    delete A;
    // ...
```

```
-> Shape Constructor
How big is a Shape? 8
Is that really how big a Shape is? 1
-> Drawing a shape
-> Drawing a shape
-> Shape Destructort
-> Drawing a shape
```

"So, what do you think about them apples, Pardner?"

Looking at the code, you come to realize that asking for the `sizeof(A)` is really only asking for the size of the pointer for an instance of a *Shape* object. It makes sense for any pointer on a 64 bit system to be 8 (8*8 = 64 after all).

But why does the `sizeof(*A)` give you a result of 1?

Well, first off, remember that the use of the * symbole acts as a 'de-reference' operation. It essentially takes a pointer and returns the contents at that location, according to the type of the object it's a pointer of. So it's saying that the class `Shape` is only ... one byte big?

Is that the case? You motion to the Data Cowboy to vacate the chair so you can add the following to the code:

```
printf("How big is a Shape? %lu\n", sizeof(A));
printf("Is that really how big a Shape is? %lu\n", sizeof(*A));
printf("So what is the size of a Shape, really? %lu\n", sizeof(Shape));
```

And the results?

```
-> Shape Constructor
How big is a Shape? 8
Is that really how big a Shape is? 1
So what is the size of a Shape, really? 1
-> Drawing a shape
-> Drawing a shape
-> Shape Destructort
-> Drawing a shape
```

But that can't be right! How can a class only be one byte big.

Our Digital Cowpoke sees the look of consternation on your face, motions to the keyboard and begins to type as you move to the side. He adds a bit of code to the bottom of the `Shape` class and and re-runs the code

```
// thar's code above here
    void Draw()
    {
        printf("  -> Drawing a shape with text: [%s]\n", value);
    }

    char value[256];
};

int main()
{
    Shape* A = new Shape();
    Shape* B = A;
    strcpy(A->value, "Here is some text");
    printf("How big is a Shape? %lu\n", sizeof(A));
    printf("Is that really how big a Shape is? %lu\n", sizeof(*A));
    printf("So what is the size of a Shape, really? %lu\n", sizeof(Shape));

    A->Draw();
    B->Draw();

    delete A;
    memset(A, 0, sizeof(Shape));  // Mimicing someone else re-using the memory that␣
↪was just freed

    B->Draw();
}
```

```
-> Shape Constructor
How big is a Shape? 8
Is that really how big a Shape is? 256
So what is the size of a Shape, really? 256
-> Drawing a shape with text: [Here is some text]
-> Drawing a shape with text: [Here is some text]
-> Shape Destructort
-> Drawing a shape with text: []
```

Big Tex's code:

The class had no data.

THE CLASS HAD NO DATA.

Now that it has data, we get a size that looks sane. But that doesn't explain why `sizeof` gave us a value of 1 when there was no data present.

Hopalong Hacker can see where you're going, as he's been there before. He cracks open your browser and feverently types in a URL he's had memorized long before you could hold a mouse.

In Memory of Bjarne

Why is the size of an empty class not zero?

> To ensure that the addresses of two different objects will be different. For the same reason, `new` always returns pointers to distinct objects.

With that, the Marlborough Man-Of-Code tips his hat to you and disappears in a puff of AND-logic.

But there's still something off with this code. Look at the results of calling the `Draw` method.

The problem is this: We don't have a complex enough codebase.

Let's talk about the logic of the codebase first:

- We allocate instances of `Shape` called `A` and assign the pointer `A` to `B`.
  - This just dupicates the pointer; they just point to the same memory block.
- We delete `A`
- We then try to access `B` by calling the `Draw` method. And it works

It'll work because we haven't had the chance for another heap allocation to reclaim that block of memory. But here's the kicker: Because we originally had a class with no data, *IT WOULDN'T MATTER!*

Wait ... why wouldn't it matter? We deleted the instance of the object. If we somehow managed to reclaim that block of memory for something else ... wouldn't that be an issue?

The answer is, how could it? Remember when we measured the size of the class, it was *effectively* zero? That's because we only measure fields in a class when allocating memory. The previous example should have proven that to you, where we allocated a character array 256 bytes long and we ended up seeing that the size of `Shape` was, in fact, 256 bytes.

But that makes no senes! The code has to live with the object instance, right? We create an instance of an object and it creates an instance of the code and data. Right?

Let me ask you a question - does the code change between instances of the same class? I mean, the data does, totally. But *does the code actually change*?

No, it doesn't.

And then, the answer is clear ... Data and Code do **not** live at the same spot in memory.

Here's something to chew on: Anatomy of a program in memory

I really love that article. It's one of the better written pieces on the layout of a program in memory. It doesn't answer all the questions, but it's a great start.

All your code lives in what's called the "Text Segment" of your application's process memory. The data for your class (everything that isn't a function) lives on the Heap. So how does your class access that data? Well, the compiler understands that there is data associated with each class and injects into your code a `this` keyword whenever it accesses a class property.

Remember this bit of code?

```
void Draw()
{
    printf("  -> Drawing a shape with text: [%s]\n", value);
}
```

That's in the `Shape` class. Well, the compiler is actually doing (no pun intended) this:

```
void Draw()
{
    printf("  -> Drawing a shape with text: [%s]\n", this->value);
}
```

Don't believe me? Here you go

Are we to the point yet where you think memory handling in C++ is horrific yet?

OK, so it's not actually horrific. You have a tremendous amount of control over memory in C++. But it's not something trivial you can ignore. Understanding what's going on with memory is incredibly important to writing good/fast/bugfree code.

More stuff to watch out for.

Oh, we are not done yet. Not by a long shot.

In the previous example, let me ask you a question:

"Who owns the instance of shape `A`?"

I mean, consider this, from the last example:

```
int main()
{
    Shape* A = new Shape();
    Shape* B = A;
    strcpy(A->value, "Here is some text");
```

What happens when we assign `A` to `B`? In languages like C#, what the 'thing' that `A` points to has an internal reference count and it increments that when the assignment happens. And that count gets decremented when `A` or `B` get set to null/set to something else/half a dozen other reasons.

But what about C++?

By default, C++ has no concept of ownership. Or reference counting. Unless you add it yourself. But I'm getting way, way ahead of myself. (To be fair, the newest C++ standards adds 'smart pointers' that do this and are considered part of "modern C++". But that's a topic for later.)

At this point, consider that you have to be dilligent of where/how you allocate/assign pointers.

Class layout in memory

What do we know about classes so far?

- The logic (methods) for clases live in the 'text' memory block for a program.
- Data (properties) for classes live either in the Heap or Stack space for a program.
- Classes without any properties have zero byte sizes, but they actually report back a non-zero size (usually but not guaranteed to be 1) when you query their size.
- Classes, via 'hidden compiler magic' have an additional *this* keyword, which maps to the address of the classes instance.

Here's a great breakdown of the `this` keyword on cppreference.com

So, what happens when we start to consider polymorphism in C++? For example, going back to our `Shape`, `Circle` and `Rectangle` classes from Review05? Let's bring them on over and try them out! I'll be removing the Allegro references so that this is just a console application.

Now let's see what we get when we start mucking around with our `Shape` class again. Currently our `Shape` class only has an integer in it, but the class is 4 bytes large. Let's put it back having just a `Point2D` as it's sole property:

```
class ShapeWithPoint
{
public:
    ShapeWithPoint();
    ~ShapeWithPoint();

    void Draw();

    Point2D mCenter;
};
```

I'll add some `printf` code back into main:

printf("What's the size of a ShapeWithPoint? %lun", sizeof(ShapeWithPoint));

and our result:

```
What's the size of a ShapeWithPoint? 8
```

That makes sense, `Point2D` has two floats (4 bytes each).

Now, what happens if we make one of the classes virtual?

```
class ShapeWithVirtual
{
public:
    ShapeWithVirtual();
    virtual ~ShapeWithVirtual();

    void Draw();

    int value;
};
```

All we're doing is making the destructor virtual. It allows derived classes to be delted by deleting a reference to the base class. We'll talk a bit about that later. However, we're talking about class sizes. So, what do we get now?

```
What's the size of a ShapeWithVirtual? 16
```

Wait wait wait. We didn't add any new data to the class! why is it bigger? Why is it 8 bytes bigger? Let's try something - in your code for both `ShapeWithPoint` and `ShapeWithVirtual` replace the `Point2D mCenter;` with a `int value;`.

The results are ...

```
What's the size of a ShapeWithPoint? 4
What's the size of a ShapeWithVirtual? 16
```

What you're seeing is the effect of the V-Table in a class. Going by the 'rule of "the 'largest data type in the class' alignment" we see that we are on 8 byte alignments. Play around with that a bit, if you want to be convinced. Or, if you want to be 100% sure, just use *#pragma pack(1)* like so:

```
#pragma pack(push)
#pragma pack(1)
class ShapeWithPoint
{
public:
    ShapeWithPoint();
    ~ShapeWithPoint();

    void Draw();

    // Point2D mCenter;
    int value;
};

class ShapeWithVirtual
{
public:
    ShapeWithVirtual();
    virtual ~ShapeWithVirtual();

    void Draw();

    // Point2D mCenter;
    int value;
};
#pragma pack(pop)
```

And the output?

```
What's the size of a ShapeWithPoint? 4
What's the size of a ShapeWithVirtual? 12
```

So how does C++ implement polymorphism? Knowing what it is and knowing how it's implemented are two very, very different things.

With our `Shape`, `Circle` and `Rectangle` classes, we've seen that we can keep track of a `Shape` pointer, but put a pointer to an instance of a `Circle` in there and we call the `Draw` method, it will resolve it to the correct object instance's `Draw` call. You know, like we did in `Review05`:

```
VirtualShape** shapes = new VirtualShape*[10];

shapes[0] = new Circle(20.0f, 30.0f, 5.0f);
shapes[1] = new Circle(40.0f, 60.0f, 10.0f);
shapes[2] = new Circle(60.0f, 90.0f, 15.0f);
shapes[3] = new Circle(80.0f, 120.0f, 20.0f);
shapes[4] = new Circle(100.0f, 150.0f, 30.0f);
shapes[5] = new Rectangle(200.0f, 300.0f, 5.0f, 5.0f);
shapes[6] = new Rectangle(220.0f, 330.0f, 10.0f, 10.0f);
shapes[7] = new Rectangle(240.0f, 360.0f, 15.0f, 15.0f);
```

```
shapes[8] = new Rectangle(260.0f, 390.0f, 20.0f, 20.0f);
shapes[9] = new Rectangle(280.0f, 420.0f, 25.0f, 25.0f);

for (int index = 0; index < 10; index++)
{
    shapes[index]->Draw();
}
```

That's polymorphism. But how does it work, under the hood? Each compiler can do it a little differently, but it really comes down to a Virtual Table and a Virtual Table Pointer.

In each instance of a class, a *Virtual Table Pointer* (commonly refered to as a `vptr`) is created - it is a pointer to a table of virtual functions (again, commonly referred to as a `vtable`). `vtable`'s usually contain the following:

- Virtual function dispatch information.
- Offsets to virtual base class subobjects
- RTTI for the object (depending on compiler options).

What we end up getting, when you take a look at something like this:

`shapes[index]->Draw();`

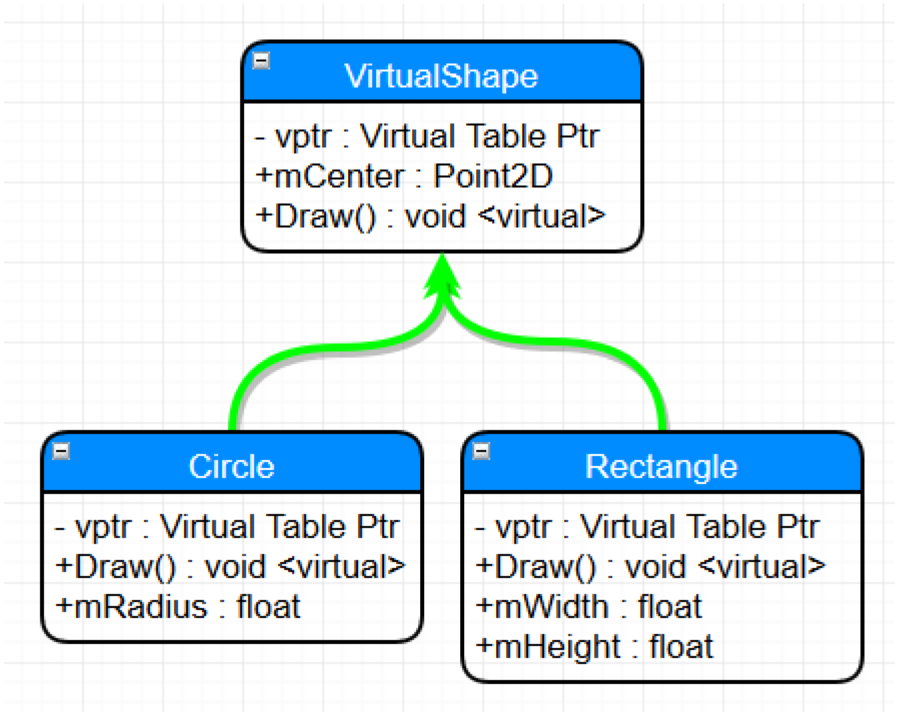
Really ends up being more like this:

`(*shapes[index]->vptr[n])(shapes[index])`

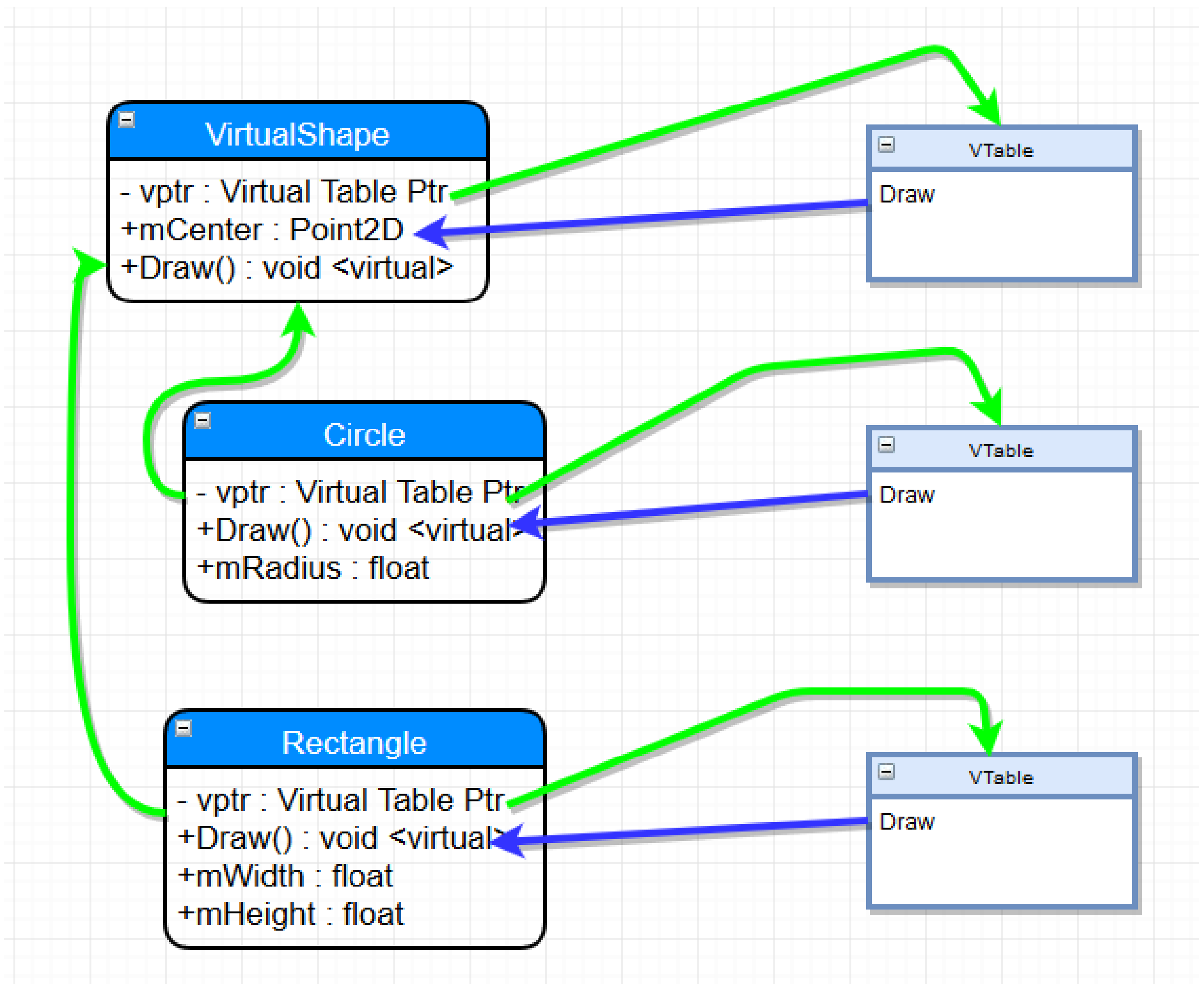
which is a pointer to a function. Not that `vptr[n]` is a slot in the `vtable` at the `n`'th element.

Kind of like this:

So, what's really going on with the compiler is that there is an added virtual function table pointer adding into the defintion of the class:

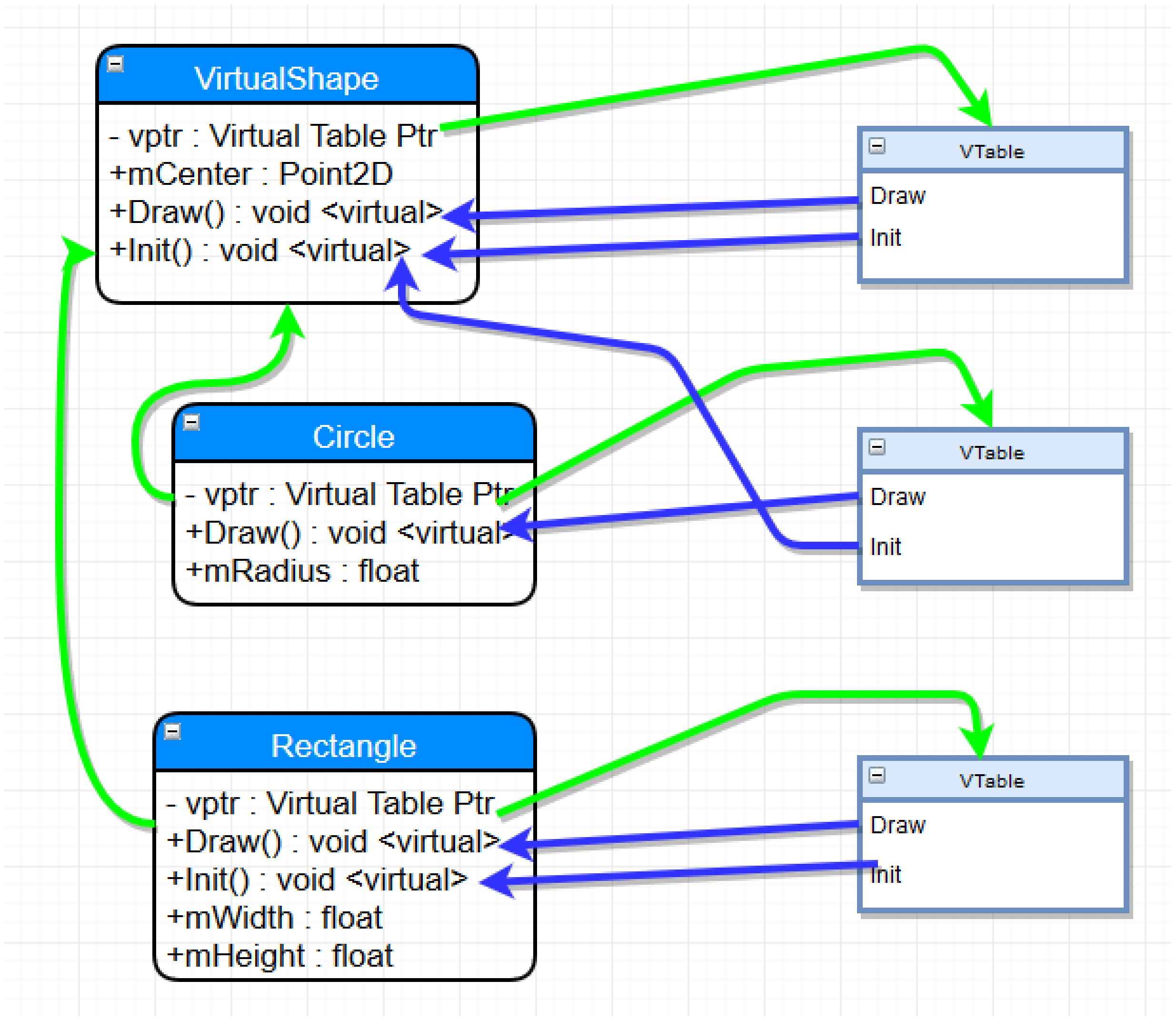


Which then links to the Virtual Function Table for each class, referring to the appropriate virtual function:



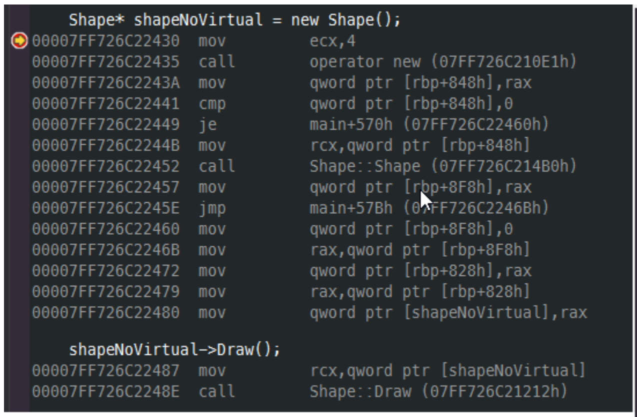
Hopefully the diagram helps - each class ends up with a virtual function table that then links to the appropriate virtual method.

Let's do something a little different - let's add a virtual function into the base and only override it in one of the classes:

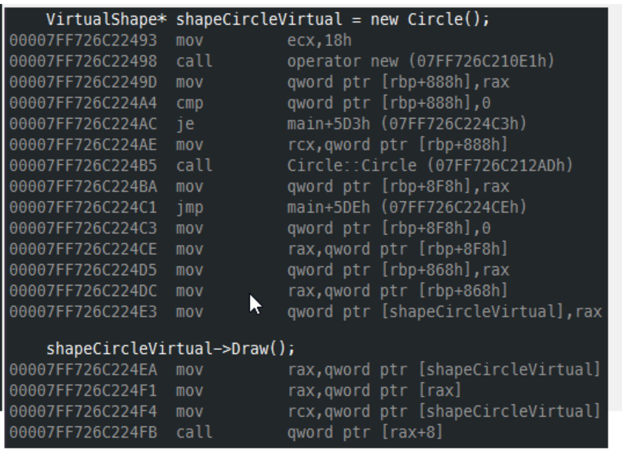


Yes, it can get pretty complex, with vtables pointing across class methods, but that's what inheretence means. And, following the object model, that can be pretty damn groovy.

But it comes with a cost. Let's look at the disassembly difference between Shape (not Virtual) and Circle (virtual)



```
    Shape* shapeNoVirtual = new Shape();
00007FF726C22430  mov         ecx,4
00007FF726C22435  call        operator new (07FF726C210E1h)
00007FF726C2243A  mov         qword ptr [rbp+848h],rax
00007FF726C22441  cmp         qword ptr [rbp+848h],0
00007FF726C22449  je          main+570h (07FF726C22460h)
00007FF726C2244B  mov         rcx,qword ptr [rbp+848h]
00007FF726C22452  call        Shape::Shape (07FF726C214B0h)
00007FF726C22457  mov         qword ptr [rbp+8F8h],rax
00007FF726C2245E  jmp         main+57Bh (07FF726C2246Bh)
00007FF726C22460  mov         qword ptr [rbp+8F8h],0
00007FF726C2246B  mov         rax,qword ptr [rbp+8F8h]
00007FF726C22472  mov         qword ptr [rbp+828h],rax
00007FF726C22479  mov         rax,qword ptr [rbp+828h]
00007FF726C22480  mov         qword ptr [shapeNoVirtual],rax

    shapeNoVirtual->Draw();
00007FF726C22487  mov         rcx,qword ptr [shapeNoVirtual]
00007FF726C2248E  call        Shape::Draw (07FF726C21212h)
```



```
    VirtualShape* shapeCircleVirtual = new Circle();
00007FF726C22493  mov         ecx,18h
00007FF726C22498  call        operator new (07FF726C210E1h)
00007FF726C2249D  mov         qword ptr [rbp+888h],rax
00007FF726C224A4  cmp         qword ptr [rbp+888h],0
00007FF726C224AC  je          main+5D3h (07FF726C224C3h)
00007FF726C224AE  mov         rcx,qword ptr [rbp+888h]
00007FF726C224B5  call        Circle::Circle (07FF726C212ADh)
00007FF726C224BA  mov         qword ptr [rbp+8F8h],rax
00007FF726C224C1  jmp         main+5DEh (07FF726C224CEh)
00007FF726C224C3  mov         qword ptr [rbp+8F8h],0
00007FF726C224CE  mov         rax,qword ptr [rbp+8F8h]
00007FF726C224D5  mov         qword ptr [rbp+868h],rax
00007FF726C224DC  mov         rax,qword ptr [rbp+868h]
00007FF726C224E3  mov         qword ptr [shapeCircleVirtual],rax

    shapeCircleVirtual->Draw();
00007FF726C224EA  mov         rax,qword ptr [shapeCircleVirtual]
00007FF726C224F1  mov         rax,qword ptr [rax]
00007FF726C224F4  mov         rcx,qword ptr [shapeCircleVirtual]
00007FF726C224FB  call        qword ptr [rax+8]
```

Creation of each object is the same, assembly wise. However invoking the `Draw` function on a virtual function incurs a different execution path. Specifically we get a `call` to the `Shape::Draw` method in the non virtualized `Shape` versus the call to an address in the virtualized version.

The cost here is two additional `mov``s to call the virtual ``Draw` method. Yeah, I know, that doesn't sound like much - two additional assembly instructions. Big whoop.

Yet that can add up, being invoked several hundred or several thousand times per frame. Remember, a frame is 1/30th (or 1/60th, or 1/120th) of a second.

But that's not *really* where our code can become slow. One thing C++ compilers do all the time is **inline** code. You may have seen in some C++ code the use of the keyword *inline* - it's kind of the same thing. The goal of the `inline` keyword was to flag the optimizer of the compiler to *inline substitution of a function* rather than invoking a function call. Function calls are slower as they mean creating a new stack, pushing data onto that stack, invoking a jump to that function, returning from that function and peeling results from that function off the stack.

With virtual functions, you can't inline that code base because you can't infer what the code actually is, in the general case. That's it. That's why it can be slow.

For stuff you call a few hundred times a frame, that might not be too bad a trade-off for a simple architecture. But for complex hierarchies, or deep object trees, you might not want to rely on virtual methods.

As a great read on the tradeoffs between polymorphic code and other options, I offer up this fantastic section on Stack Exchange Stack Exchange discussion on virtual functions.

Finding memory leaks

I was going to write a big ol' post about how to find memory leaks. Like, a completely separate project and README.md file and everything.

But there's this video from Adam Welch that pretty much walks you through how to use Visual Studio's memory profiler.

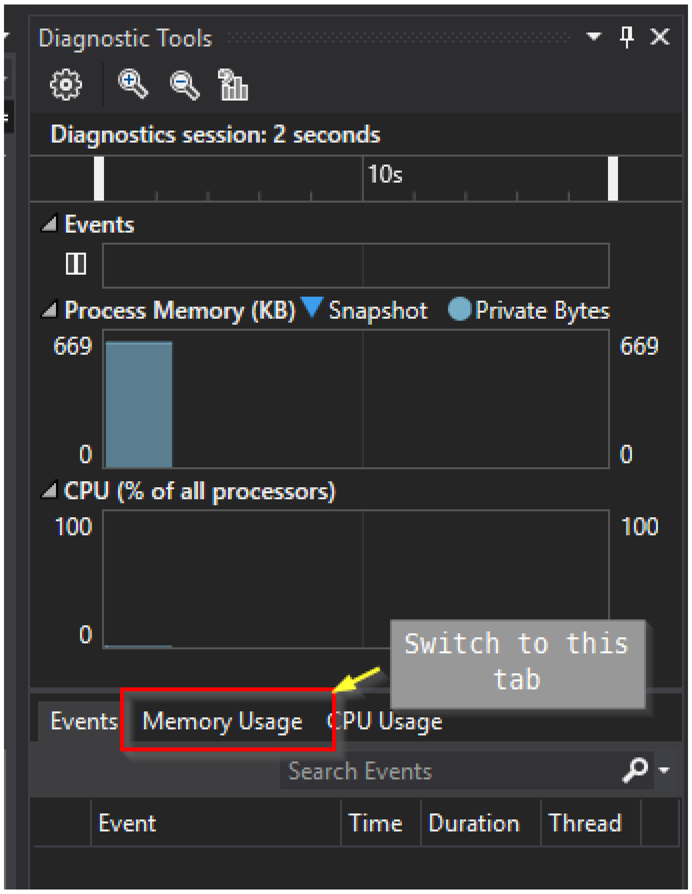
You should watch it. Go. Now.

OK, so how would we go about profiling our app? How do we find memory leaks in the project associated with this codebase?

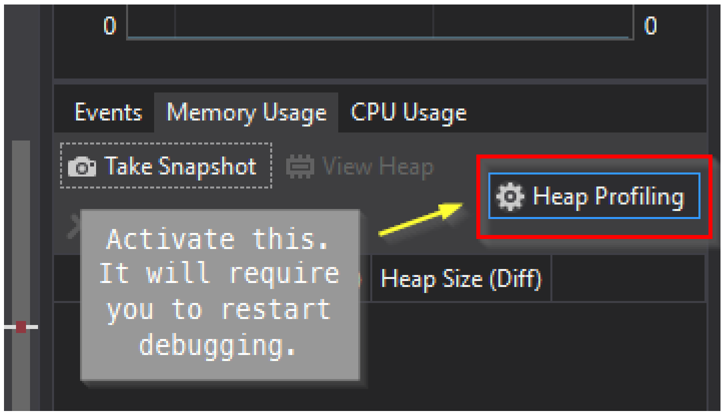
Pretty simple

0. Set up your memory profiling tools

In Debug, run your the `PointerIntro` project. There's a `_getch()` call in the code base that forcibly halts the program. Once you've run it, you'll see this:

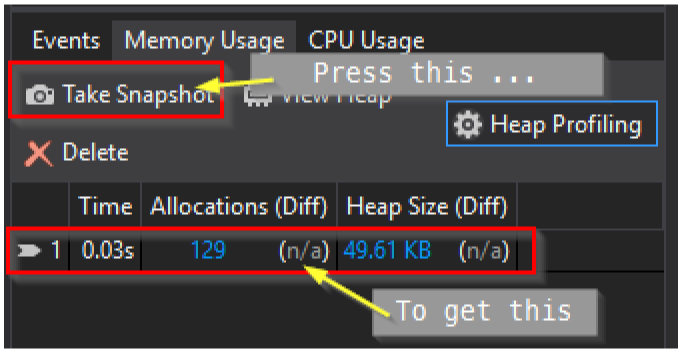


And then ...



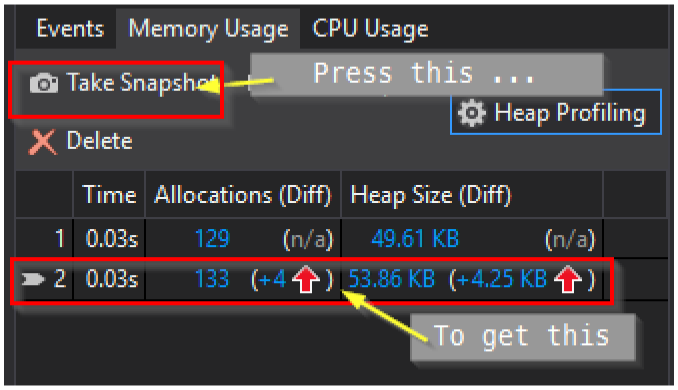
1. Add Breakpoints at the start and end of your program

So, before you go too far, put some breakpoints in your code. Preferably at the start and just before you exit the app. Then re-run the applicaiton in the debugger. When you hit your first breakpoint ...



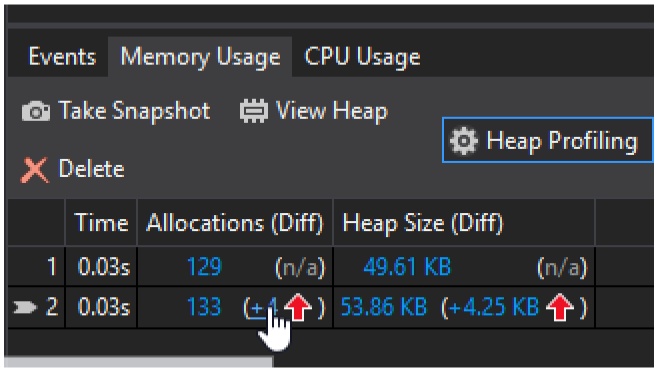
	Time	Allocations (Diff)		Heap Size (Diff)	
1	0.03s	129	(n/a)	49.61 KB	(n/a)

Continue debugging and hit your next breakpoint ...



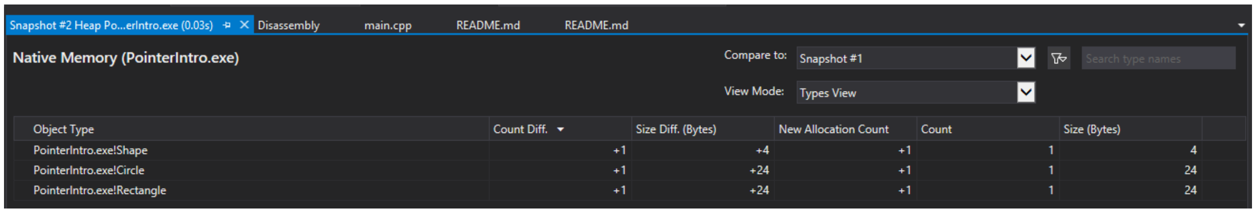
	Time	Allocations (Diff)		Heap Size (Diff)	
1	0.03s	129	(n/a)	49.61 KB	(n/a)
2	0.03s	133	(+4)	53.86 KB	(+4.25 KB)

2. See where your allocations are



	Time	Allocations (Diff)		Heap Size (Diff)	
1	0.03s	129	(n/a)	49.61 KB	(n/a)
2	0.03s	133	(+4)	53.86 KB	(+4.25 KB)

As you can see, we end up with a delta of allocation. And it's going in the wrong direction (that is, upwards). If you click on any of the higlighted links, you then get to see some more detail:



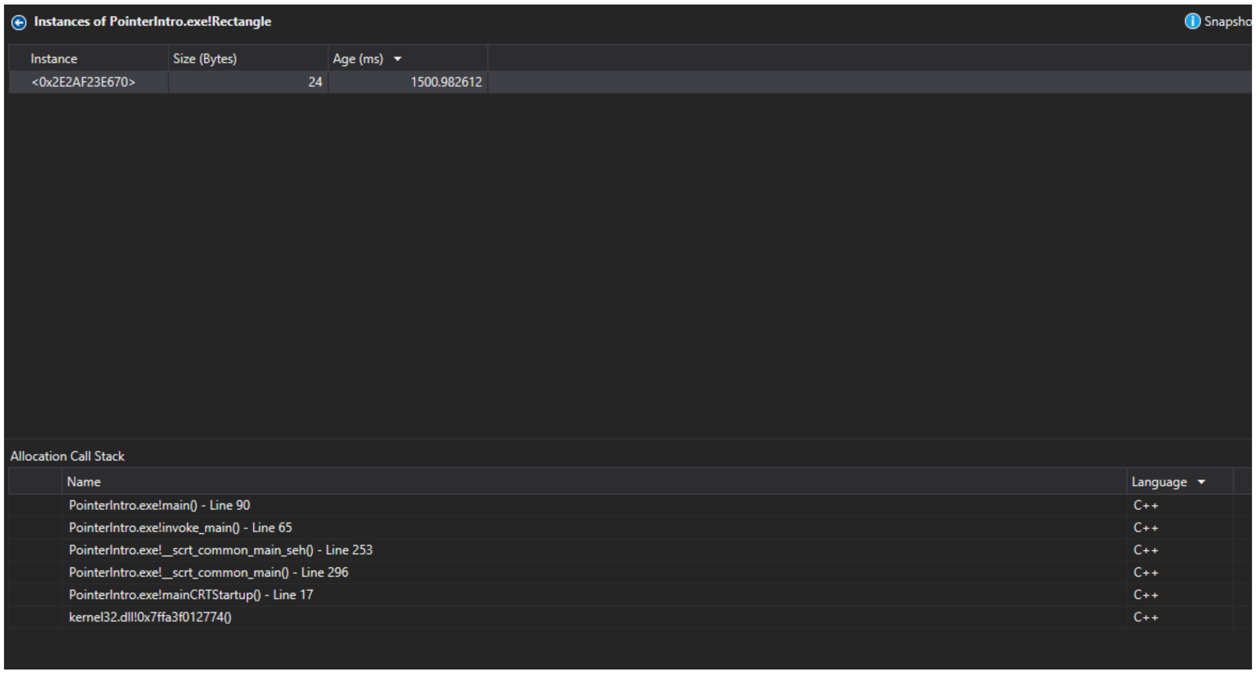
Native Memory (PointerIntro.exe)

Compare to: Snapshot #1

View Mode: Types View

Object Type	Count Diff.	Size Diff. (Bytes)	New Allocation Count	Count	Size (Bytes)
PointerIntro.exe!Shape	+1	+4	+1	1	4
PointerIntro.exe!Circle	+1	+24	+1	1	24
PointerIntro.exe!Rectangle	+1	+24	+1	1	24

Double clicking on one of the entries takes us a little deeper.



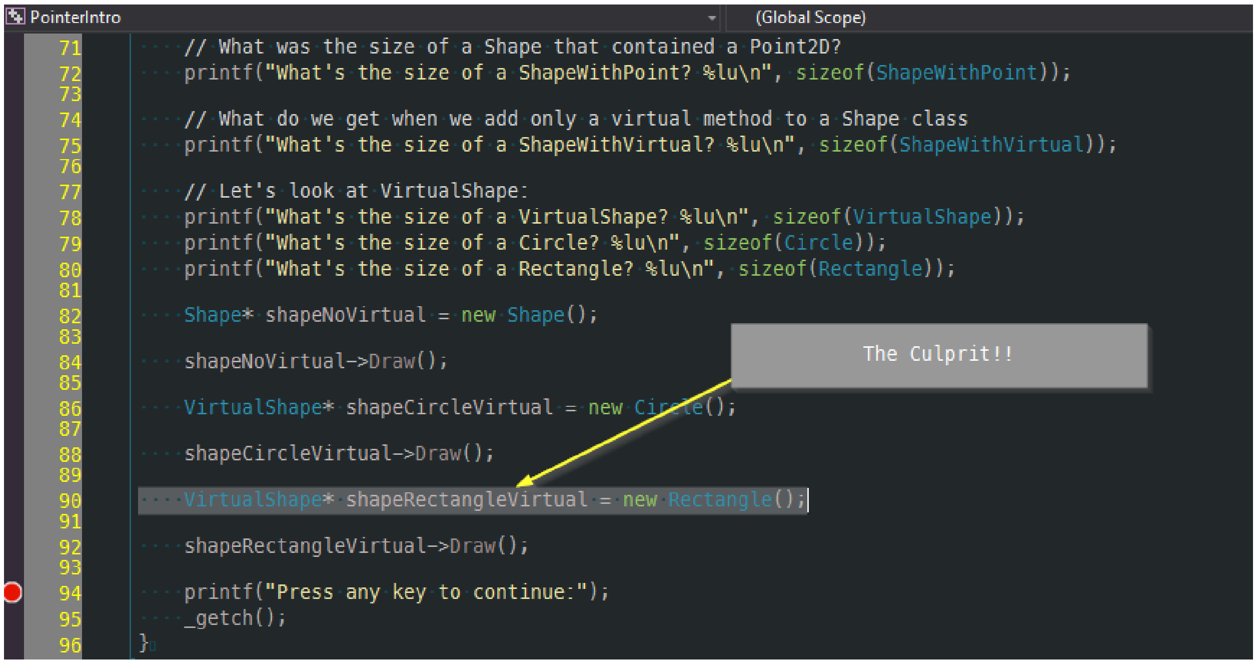
Instances of PointerIntro.exe!Rectangle

Instance	Size (Bytes)	Age (ms)
<0x2E2AF23E670>	24	1500.982612

Allocation Call Stack

Name	Language
PointerIntro.exe!main() - Line 90	C++
PointerIntro.exe!invoke_main() - Line 65	C++
PointerIntro.exe!__scrt_common_main_seh() - Line 253	C++
PointerIntro.exe!__scrt_common_main() - Line 296	C++
PointerIntro.exe!mainCRTStartup() - Line 17	C++
kernel32.dll!0x7ffa3f012774()	C++

Double clicking one last time takes us to the allocation in question.



```
71      // What was the size of a Shape that contained a Point2D?
72      printf("What's the size of a ShapeWithPoint? %lu\n", sizeof(ShapeWithPoint));
73
74      // What do we get when we add only a virtual method to a Shape class
75      printf("What's the size of a ShapeWithVirtual? %lu\n", sizeof(ShapeWithVirtual));
76
77      // Let's look at VirtualShape:
78      printf("What's the size of a VirtualShape? %lu\n", sizeof(VirtualShape));
79      printf("What's the size of a Circle? %lu\n", sizeof(Circle));
80      printf("What's the size of a Rectangle? %lu\n", sizeof(Rectangle));
81
82      Shape* shapeNoVirtual = new Shape();
83
84      shapeNoVirtual->Draw();
85
86      VirtualShape* shapeCircleVirtual = new Circle();
87
88      shapeCircleVirtual->Draw();
89
90      VirtualShape* shapeRectangleVirtual = new Rectangle();
91
92      shapeRectangleVirtual->Draw();
93
94      printf("Press any key to continue:");
95      _getch();
96  }
```

Debugging memory leaks has never been so easy!

Summary

Well, again that was a longish bit of writing. It's not trivial, but it's good stuff to have under your belt. And it's important to understand this stuff, even if you're not digging into the guts of low level optimization every day (I know I don't).

Hope that helps out!

Some references:

- C++ Virtual functions

Intermediate C++ 01 - Templates

Summary

I am not a master of Templates in C++. This is by design on my part. I personally find that they have very limited use. That's not a commonly shared opinion, to be honest with the reader, so I'm taking it upon myself to see if it's just my own personal prejudice or if it's something that my gut check has been accurate about.

What are Templates in C++

Templates are a way of performing operations on 'generic' data types. That's pretty much it.

So, for example, say we wanted to create a function like so:

```
anytype Min(anytype valueA, anytype valueB)
{
    return valueA < valueB ? valueA : valueB;
}
```

So if we run this function on a `float` or `int` or `double` or even `char` data type, it will just work, without having to create specialized versions of the same code. Without templates, you'd have to do either:

```
float Min(float valueA, float valueB)
{
    return valueA < valueB ? valueA : valueB;
}

double Min(double valueA, double valueB)
{
    return valueA < valueB ? valueA : valueB;
}

int Min(int valueA, int valueB)
{
    return valueA < valueB ? valueA : valueB;
}

char Min(char valueA, char valueB)
{
    return valueA < valueB ? valueA : valueB;
}
```

or we do it as a macro:

```
#define MIN(valueA, valueB) \
    ((valueA < valueB) ? valueA : valueB)
```

Here's the above in C++

So, how would we do that in a 'template' way? It's actually very, very easy:

```
template<typename T>
T Min(T valueA, T valueB)
{
    return valueA < valueB ? valueA : valueB;
}
```

In practice, here's the version templated

OK, so ... what's the problem with that? I mean, that looks pretty sweet, right?

Well, let's see what it looks like when it's compiled. First off, what does it look like as functions?

```
main:
 .LFB1494:
    push rbp
    mov rbp, rsp
    push rbx
    sub rsp, 24
    mov esi, OFFSET FLAT:.LC0
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char␣
↪const*)
    mov esi, 8
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov esi, OFFSET FLAT:.LC0
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char␣
↪const*)
    mov rbx, rax
    mov esi, 8
    mov edi, 10
    call Min(int, int)
    mov esi, eax
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov esi, OFFSET FLAT:.LC1
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char␣
↪const*)
    mov rbx, rax
    movsd xmm0, QWORD PTR .LC2[rip]
    mov rax, QWORD PTR .LC3[rip]
    movapd xmm1, xmm0
    mov QWORD PTR [rbp-24], rax
    movsd xmm0, QWORD PTR [rbp-24]
    call Min(double, double)
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(double)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov esi, OFFSET FLAT:.LC4
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char␣
↪const*)
    mov rbx, rax
    movss xmm1, DWORD PTR .LC5[rip]
    movss xmm0, DWORD PTR .LC6[rip]
    call Min(float, float)
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(float)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov esi, OFFSET FLAT:.LC7
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char␣
↪const*)
    mov rbx, rax
    mov esi, 97
    mov edi, 65
    call Min(char, char)
    movsx eax, al
    mov esi, eax
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<<
↪<std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov eax, 0
    add rsp, 24
    pop rbx
    pop rbp
    ret
```

To break it down, let's look at this:

```
call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
```

```
mov rbx, rax
mov esi, 8
mov edi, 10
call Min(int, int)  <<== we simply invoke the function
mov esi, eax
mov rdi, rbx
call std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

Here's the assembly for this

What happens when we use a template version?

```
main:
 .LFB1491:
   push rbp
   mov rbp, rsp
   push rbx
   sub rsp, 24
   mov esi, OFFSET FLAT:.LC0
   mov edi, OFFSET FLAT:std::cout
   call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
   mov rbx, rax
   mov esi, 8
   mov edi, 10
   call int Min<int>(int, int)
   mov esi, eax
   mov rdi, rbx
   call std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
   mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
   mov rdi, rax
   call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
   mov esi, OFFSET FLAT:.LC1
   mov edi, OFFSET FLAT:std::cout
   call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
   mov rbx, rax
   movsd xmm0, QWORD PTR .LC2[rip]
   mov rax, QWORD PTR .LC3[rip]
   movapd xmm1, xmm0
   mov QWORD PTR [rbp-24], rax
   movsd xmm0, QWORD PTR [rbp-24]
   call double Min<double>(double, double)
   mov rdi, rbx
   call std::basic_ostream<char, std::char_traits<char> >::operator<<(double)
   mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
   mov rdi, rax
   call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
   mov esi, OFFSET FLAT:.LC4
   mov edi, OFFSET FLAT:std::cout
   call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
```

```
    mov rbx, rax
    movss xmm1, DWORD PTR .LC5[rip]
    movss xmm0, DWORD PTR .LC6[rip]
    call float Min<float>(float, float)
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(float)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov esi, OFFSET FLAT:.LC7
    mov edi, OFFSET FLAT:std::cout
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
    mov rbx, rax
    mov esi, 97
    mov edi, 65
    call char Min<char>(char, char)
    movsx eax, al
    mov esi, eax
    mov rdi, rbx
    call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char)
    mov esi, OFFSET FLAT:std::basic_ostream<char, std::char_traits<char> >& std::endl
↪<char, std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&)
    mov rdi, rax
    call std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_
↪ostream<char, std::char_traits<char> >& (*)(std::basic_ostream<char, std::char_
↪traits<char> >&))
    mov eax, 0
    add rsp, 24
    pop rbx
    pop rbp
    ret
```

Here's what the assembly for that looks like

Again, let's look at a smaller section:

```
call std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_
↪traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)
mov rbx, rax
mov esi, 8
mov edi, 10
call int Min<int>(int, int)
mov esi, eax
mov rdi, rbx
call std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
```

That looks ... surprisingly similar.

Let's compare side by side

```
call std::basic_ostream<char,                          | call std::basic_ostream<char,
    std::char_traits<char> >&                          |     std::char_traits<char> >&
    std::operator<< <std::char_traits<char> >          |     std::operator<< <std::char_
↪traits<char> >
```

```
    (std::basic_ostream<char,                          |     (std::basic_ostream<char,
    std::char_traits<char> >&, char const*)            |     std::char_traits<char> >&,␣
↪char const*)
mov rbx, rax                                           | mov rbx, rax
mov esi, 8                                             | mov esi, 8
mov edi, 10                                            | mov edi, 10
call Min(int, int)                                     | call int Min<int>(int, int)
mov esi, eax                                           | mov esi, eax
mov rdi, rbx                                           | mov rdi, rbx
call std::basic_ostream<char,                          | call std::basic_ostream<char,
    std::char_traits<char> >::operator<<(int)          |     std::char_traits<char> >
↪::operator<<(int)
```

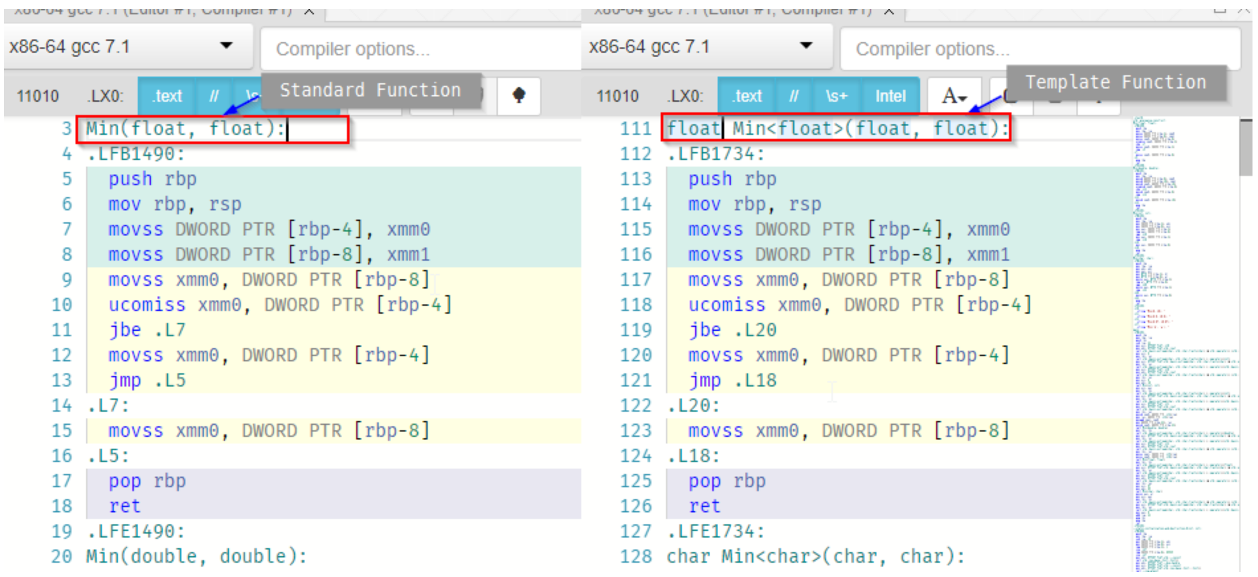
That is ... nearly identical. The difference is here:

```
call Min(int, int)                                     | call int Min<int>(int, int)
```

So ... what does `call int Min<int>(int, int)` mean?

It means the same thing as `call Min(int, int)` - the `Min(int, int)` is the function signature. In the `call int Min<int>(int, int)` example, the `int Min<int>(int, int)` is the function signature. So it is doing a call into a function. This was not what I was expecting, as I thought it would inline the assembly at the call site. - I've always been under the assumption that templates lead to code bloat. I may have been under a false assumption there.

I mean, let's look at the code:



```
Min(float, float):
.LFB1490:
  push rbp
  mov rbp, rsp
  movss DWORD PTR [rbp-4], xmm0
  movss DWORD PTR [rbp-8], xmm1
  movss xmm0, DWORD PTR [rbp-8]
  ucomiss xmm0, DWORD PTR [rbp-4]
  jbe .L7
  movss xmm0, DWORD PTR [rbp-4]
  jmp .L5
.L7:
  movss xmm0, DWORD PTR [rbp-8]
.L5:
  pop rbp
  ret
.LFE1490:
Min(double, double):
```

```
float Min<float>(float, float):
.LFB1734:
  push rbp
  mov rbp, rsp
  movss DWORD PTR [rbp-4], xmm0
  movss DWORD PTR [rbp-8], xmm1
  movss xmm0, DWORD PTR [rbp-8]
  ucomiss xmm0, DWORD PTR [rbp-4]
  jbe .L20
  movss xmm0, DWORD PTR [rbp-4]
  jmp .L18
.L20:
  movss xmm0, DWORD PTR [rbp-8]
.L18:
  pop rbp
  ret
.LFE1734:
char Min<char>(char, char):
```

Those are identical. So that's one theory out the window.

OK, so what else bugs me about template functions. Well, there's the whole "if you don't invoke it, you don't compile it" part of templates. By that I mean, if you don't actually call a function template, it doesn't actually get compiled.

For instance ...

```
// Example program
#include <iostream>
#include <string>

template<typename T>
```

```
T Min(T valueA, T valueB)
{
    return valueA < valueB ? valueA : valueB;
}

template<typename T>
T Max(T valueA, T valueB)
{
    return valueA > valueB ? valueA : valueB;
}

int main()
{
std::cout << "Min(8, 10): " << Min(10, 8) << std::endl;
std::cout << "Min(8.0, 10.0): " << Min(10.0, 8.0) << std::endl;
std::cout << "Min(8.0f, 10.0f): " << Min(10.0f, 8.0f) << std::endl;
std::cout << "Min('A', 'a'): " << Min('A', 'a') << std::endl;
}
```

if you check out the Code explorer version here, you'll see that the template code isn't actually compiled into the exe. But isn't that a good thing? I mean, that's less bloat!

Template code that isn't used doesn't get compiled. That means you can introduce a compiler error and it won't get caught unless you utilize the function! So, let's take a look at the same example, but with an error introduced into the `Max` function. Let's rename `return valueA > valueB ? valueA : valueB;` to `return valueC > valueB ? valueA : valueB;` and see the compiler happily compile it without error:

And the results are ... not what I expected!!

The hells? That failed to compile! Am I nuts? That's never called. So that shouldn't have been compiled!

Goes off to try it with the Microsoft compiler

OK, I'm not crazy. In Microsoft Visual C++ (2015), that compiles without error. OR WARNING!

Don't beleve me? Go and add

```
template<typename T>
T Max(T valueA, T valueB)
{
    return valueA > valueB ? valueA : valueB;
}
```

into any of the existing code we've built in Visual C++. I'll wait.

This illustrates another issue - don't assume all compilers handle templates the same way. I now want to see what this is going to look like compiled with the Microsoft tools. Let's do that.

So, it takes a bit of jigging to get it to work correctly. I'll build a review at some point in time to talk about it. However, we end up with this for the assembly (in Debug) for the template:

```
--- d:\dev\bagofholding\programmingincpp\templates01\main.cpp ------------------
// Example program
#include <iostream>
#include <string>

template<typename T>
T Min(T valueA, T valueB)
{
00007FF6566622B0  mov          dword ptr [rsp+10h],edx
00007FF6566622B4  mov          dword ptr [rsp+8],ecx
00007FF6566622B8  push         rbp
00007FF6566622B9  push         rdi
00007FF6566622BA  sub          rsp,0D8h
00007FF6566622C1  mov          rbp,rsp
00007FF6566622C4  mov          rdi,rsp
00007FF6566622C7  mov          ecx,36h
00007FF6566622CC  mov          eax,0CCCCCCCCh
00007FF6566622D1  rep stos     dword ptr [rdi]
00007FF6566622D3  mov          ecx,dword ptr [rsp+0F8h]
    return valueA < valueB ? valueA : valueB;
00007FF6566622DA  mov          eax,dword ptr [valueB]
00007FF6566622E0  cmp          dword ptr [valueA],eax
00007FF6566622E6  jge          Min<int>+46h (07FF6566622F6h)
00007FF6566622E8  mov          eax,dword ptr [valueA]
00007FF6566622EE  mov          dword ptr [rbp+0C0h],eax
00007FF6566622F4  jmp          Min<int>+52h (07FF656662302h)
00007FF6566622F6  mov          eax,dword ptr [valueB]
00007FF6566622FC  mov          dword ptr [rbp+0C0h],eax
00007FF656662302  mov          eax,dword ptr [rbp+0C0h]
}
```

And it's invoked like so:

```
int main()
{
00007FF656662B70  push         rbp
00007FF656662B72  push         rdi
00007FF656662B73  sub          rsp,0F8h
00007FF656662B7A  lea          rbp,[rsp+20h]
00007FF656662B7F  mov          rdi,rsp
00007FF656662B82  mov          ecx,3Eh
00007FF656662B87  mov          eax,0CCCCCCCCh
00007FF656662B8C  rep stos     dword ptr [rdi]
    std::cout << "Min(8, 10): " << Min(10, 8) << std::endl;
00007FF656662B8E  mov          edx,8
00007FF656662B93  mov          ecx,0Ah
00007FF656662B98  call         Min<int> (07FF656661055h)  // <<== This is the address␣
↪of the function to call (07FF656661055h)
00007FF656662B9D  mov          dword ptr [rbp+0C0h],eax
00007FF656662BA3  lea          rdx,[string "Min(8, 10): " (07FF65666AD98h)]
00007FF656662BAA  mov          rcx,qword ptr [__imp_std::cout (07FF656671150h)]
00007FF656662BB1  call         std::operator<<<std::char_traits<char> >␣
↪(07FF65666113Bh)
00007FF656662BB6  mov          ecx,dword ptr [rbp+0C0h]
    std::cout << "Min(8, 10): " << Min(10, 8) << std::endl;
00007FF656662BBC  mov          edx,ecx
00007FF656662BBE  mov          rcx,rax
00007FF656662BC1  call         qword ptr [__imp_std::basic_ostream<char,std::char_
↪traits<char> >::operator<< (07FF656671178h)]
00007FF656662BC7  lea          rdx,[std::endl<char,std::char_traits<char> >␣
↪(07FF6566610B9h)]
00007FF656662BCE  mov          rcx,rax
00007FF656662BD1  call         qword ptr [__imp_std::basic_ostream<char,std::char_
↪traits<char> >::operator<< (07FF656671180h)]
```

You can find the source for this in the `Templates01.vcxproj` project.