INITIALIZING OPENGL

A Sadistic Approach to Modern OpenGL

Intentionally left blank.

Intentionally left blank.

Initializing OpenGL

A Sadistic Approach to Modern OpenGL

T h o m p s o n L e e

Intentionally left blank.

(This is not your typically common programming book.)

Thanks to everyone for your support.

Written in 2018. Published on Github.

TABLE OF CONTENTS

Preface

Yeah! Nothing in your face!

This is an introductory book to creating boilerplate OpenGL programming code in C and C++, with a lot of pictures. It will not cover concepts such as C and C++ primer, algorithms, data structures, operating systems, and many others that does not relate to initializing OpenGL. It also will not cover any subjects mentioning POSIX, UNIX-based systems, and the subjective ideologies of what platforms to choose, except when we are dealing with programming language interoperability issues in C and C++.

What this book will cover are concepts about initializing OpenGL on a Windows platform, but it is written in a matter that a typical reader will attempt their best to understand. It is not suggested that the reader should only focus on what the book will teach, and the author is encouraging the readers to go out on an adventure on the Internet to find the references and citations this book has glossed over. All the references and citations are listed at the end of this book.

This is a poorly written, informal book, only written by the author. Poorly written, as in, the book structure is off-the-walls. It is most likely to have errors and untrue facts, have too many facts and overload the reader with too much information, and information outside the scope of the author's limited knowledge. Neither should anyone believe one will take this book at face value, and say this book is a professionally-written book that has passed peer reviews and garnered publishing support in the video gaming industry. But if you, the reader, so wished to praise the author for the effort, then we shall oblige and celebrate with a bottle of beer, just imagining the beer resting against a hot sweaty palm.

If you are still reading this preface, we salute you, and will say that you are a charmer. You will be able to power through anything, any obstacles ahead of you, anything that some of us were not able to overcome. Go on, and read ahead.

Oh, and by the way, this book is formatted in the following table:

Style Type	Example	Meaning
Text	Hello world.	Normal text.
Inlined Programming Code	`extern class Foo;`	Inlined C and C++ code styling.
Actual Programming Code	`struct Foo foo = {};`	Actual C and C++ code styling.
File names	`Filename.txt`	A file's name.
Function arguments	`nCmdShow`	A function's parameter.
Constants	`SW_SHOWDEFAULT`	Global constant flag.

Chapter 1

Let Us Start with Windows!

Here we are at the turn of the second decade of the new millennium, with technology bustling at its seams in the gaming space. With the rise in a whole slew of modern practices in getting games out the door, we need to take a hundred steps back and focus on how it all begins.

Now, we shall cut to the chase, step on it for a bit, and let us get started on initializing your very first OpenGL context in Windows, or on a Windows platform.

Installation

Oh yeah, that is a bold introduction. Here is a list of things you need before you even think about jumping ahead:

- A personal computer (Laptop).
- Your graphics processing card (GPU).
- Your Windows operating system of choice running in 64-bit (Windows 10 Pro).
- OpenGL drivers (Download and install them from your hardware vendor).
- Visual Studio 2017 (Community Edition).
- Visual C++ (C++14).
- Windows 10 SDK (Depending on your operating system).

And here is a list of optional things you may or may not need:

- Your favorite build environment.

As of this time of writing, Visual Studio 2017 is the latest edition of the Microsoft Visual Studio suite, Windows 10 is the latest edition of Microsoft Windows family, ATI is bought by AMD, and finally Nvidia and AMD are respectively competing with Nvidia GeForce 2080 TI vs. AMD Radeon RX 580X. This is 2018, ok?

This is where we will see you in 2 to 3 hours once everything is installed. Visual C++ needs to be selected in the Visual Studio 2017 Installer. Windows 10 SDK is a separate installation, so you need to wait until all installations are finished before installing this SDK. All big installations must follow a reboot of your computer. And since you may be using Windows 10 of any flavor (Home, Pro, Enterprise, etc.), you may also require installing and patching updates, so it will take a bit more time in the process. But, the author assures you, once everything is complete, we can continue forth.

Why do we need to install Windows 10 SDK, you may ask?[1] OpenGL is required to use the packaged "`opengl32.lib`" and "`opengl32.dll`," which Microsoft compiled with proprietary source codes to communicate between the application to the hardware, essentially acting as a conduit between *Installable Client Driver* (ICD) and the applications using OpenGL.

If you do happen to skip installing the latest and greatest OpenGL drivers, you will default to a software version of OpenGL 1.1 (on Windows 98, Windows ME, and Windows 2000), or a Direct3D wrapper that supports OpenGL 1.1 (Windows XP and up). None of these options are particularly fast, so installing drivers is always an important step.[2]

However, if you do not have any decent GPUs, well then. Sorry to say, you need to start investing in purchasing a nice computer for yourself. And you should not be continuing to read, but we will not stop you from doing so.

Picking Your Poison

Usually, by the time you have everything set up, this is the moment where you start to choose what programming language to start with. For all intended purposes, and for the dignity of this book, we will be using C and/or C++, whichever you feel like it. You can also go ahead and choose Java, Python, Ruby, Android JNI, Android Kotlin, or any other programming languages that specifically supports C and C++ bindings. In short, any programming languages you wished to choose from.

But let us jump back to C and C++. So, why do we care?

The answer is simple: C and C++ are System Programming Languages, both of which are supported natively by the Windows platform. Technically, it should have been only C, which Windows uses natively, but since C++ is an extension of C, we will allow this. Alright, now stop arguing!

If you are not using C and/or C++, then you must download and install packages and/or libraries for your chosen languages that includes OpenGL bindings. Some of those languages come with OpenGL bindings pre-installed, while others may have separate downloads for you to install. And if you chose to

[1] Another reason not included in the passage is on Github, but the issue is closed. (Github, 2016). Windows SDK will now automatically implicitly link `opengl32.lib` when using MSBuild.
[2] Obtained information from the official Khronos Group OpenGL wiki. (Khronos Group, 2018)

do this, then this book would not be of much use for you, considering the fact the pre-installed packages and/or libraries with OpenGL bindings have already done what this book is about to teach you.

So, choose wisely.

Development Setup

This is not a trick subchapter. We are still in the setting up phase.

The next step you need to do is to choose whether you want to work with a build environment. Build environments come in a variety of shapes and sizes. Here is a list of them as examples:

- Visual Studio Project
- GNU makefile
- CMake
- Microsoft Program Maintenance Utility (NMake)
- Microsoft Windows Command Line Interface

The author is sure you will choose either one of the first 3 options, with the 1st option being the most common one. Otherwise, why would you install Visual Studio 2017 in the first place? Nonetheless, all build environments work fine, as they all do 1 simple task: Executing the compiler with a series of commands and switch flags specified by the user.

And your goal here is to setup your build environment up, so it can link to OpenGL via the Linker. For this book, we will be using the Visual Studio Project as our main build environment.

For starters, please run and execute Visual Studio 2017. (See Figure 1)

While you're staring at Visual Studio 2017 in Figure 1, think about what sort of application type do you prefer: 32-bit applications, or 64-bit applications. Punch this information into your build environment, so your build environment will help you set up the necessary switches and flags to get your preferred application type as the output.

Once Visual Studio 2017 finishes loading, it is time to create a new C++ project. Indeed, every C project in Visual Studio is named "C++ Project," so start getting used to this. Microsoft avoided C for a long while, you know?[3]

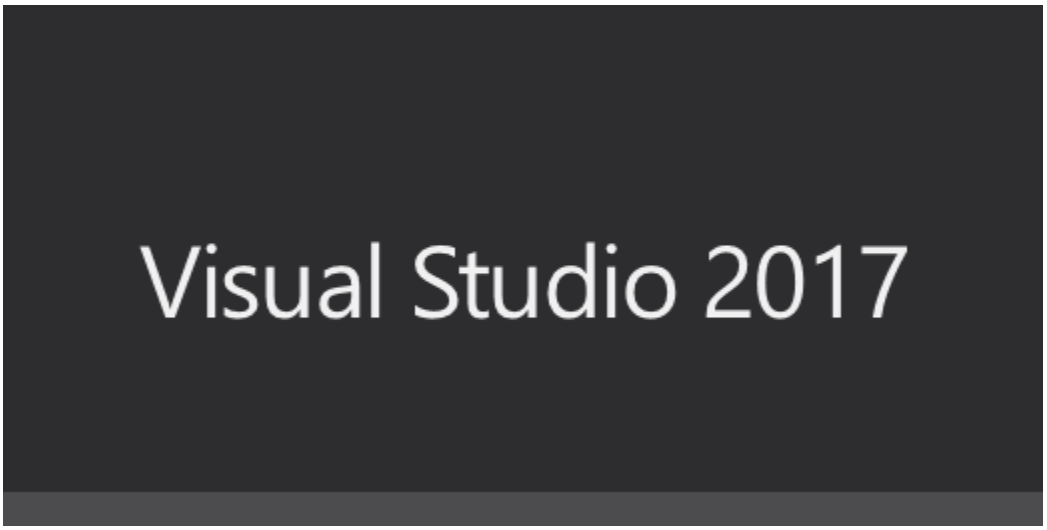

Figure 1: The loading screen of Visual Studio 2017.

The template for the C++ project is to use an Empty Project template. This will make everything simplified for you. (See Figure 3)

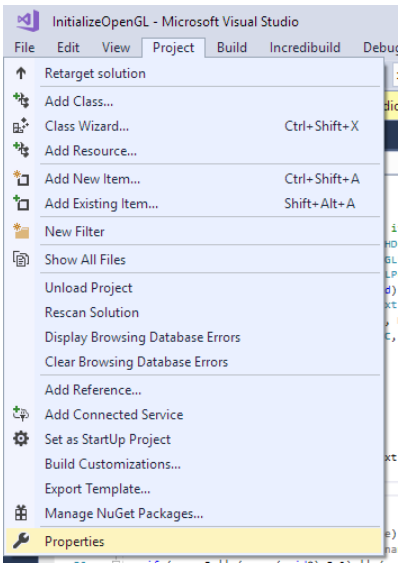

Figure 2: Project Properties.

Once the Empty Project is created, at the top of the Toolbar, click on Project, then Properties down at the bottom. Consult the image on the left, if you need help finding it. (See Figure 2)

Visual Studio 2017 will then open up the Project Property Pages for you, but did you notice something different? It is the selected target platform up at the top that is wrong here. By default, Visual Studio will assign the target platform to target "`Active(x64)`." This is not the right target platform of choice, even though it is indeed running in a 64-bit environment.

You need to change it to be "`All Platforms`." So, go ahead and select the option in the dropdown list. When you select that option, it will automatically show an alert box, telling you the Configuration Page is missing. This is normal, because it needs to generate all Project Property Pages for all platforms. Close the dialog, and reopen the Project Property Pages, and it should be like Figure 4.

[3] Quote from Ars Technica (Bright, 2013): "Microsoft has long avoided supporting C99, the major update to C++'s predecessor that was standardized last millennium, claiming that there was little demand for it among Visual Studio users. This was true, but only to a point; it's true that many Windows developers were not especially interested in C99 because they had no good tooling to support it."

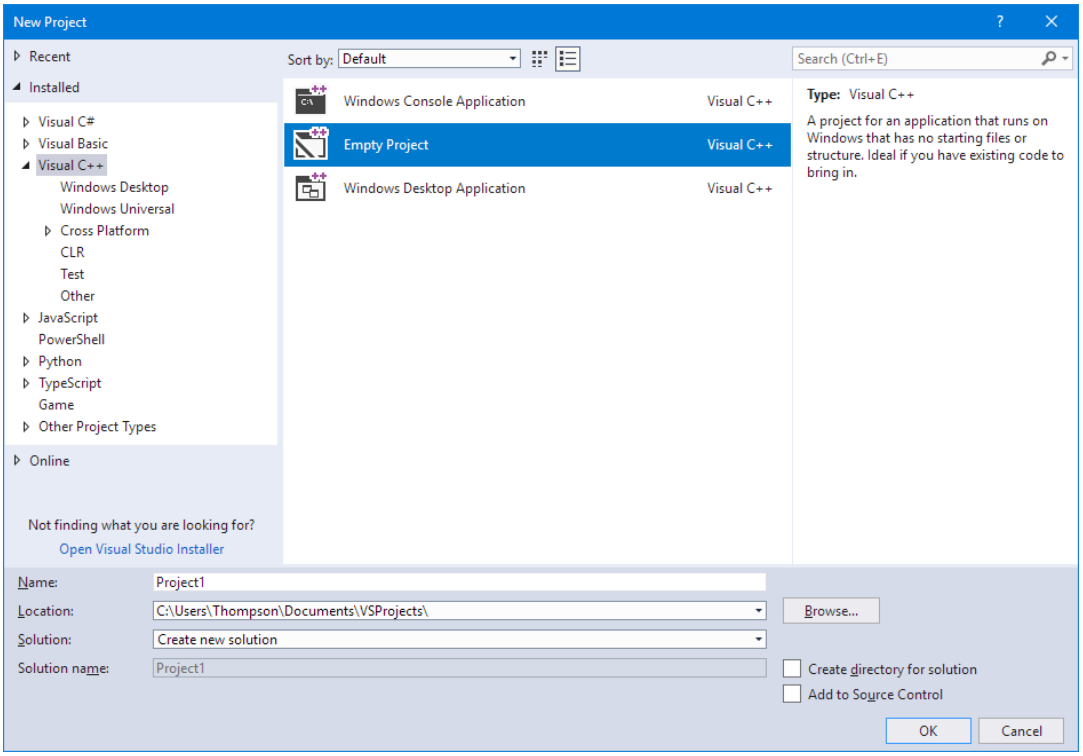

Figure 3: The Visual C++ Empty Project template, chosen for you.

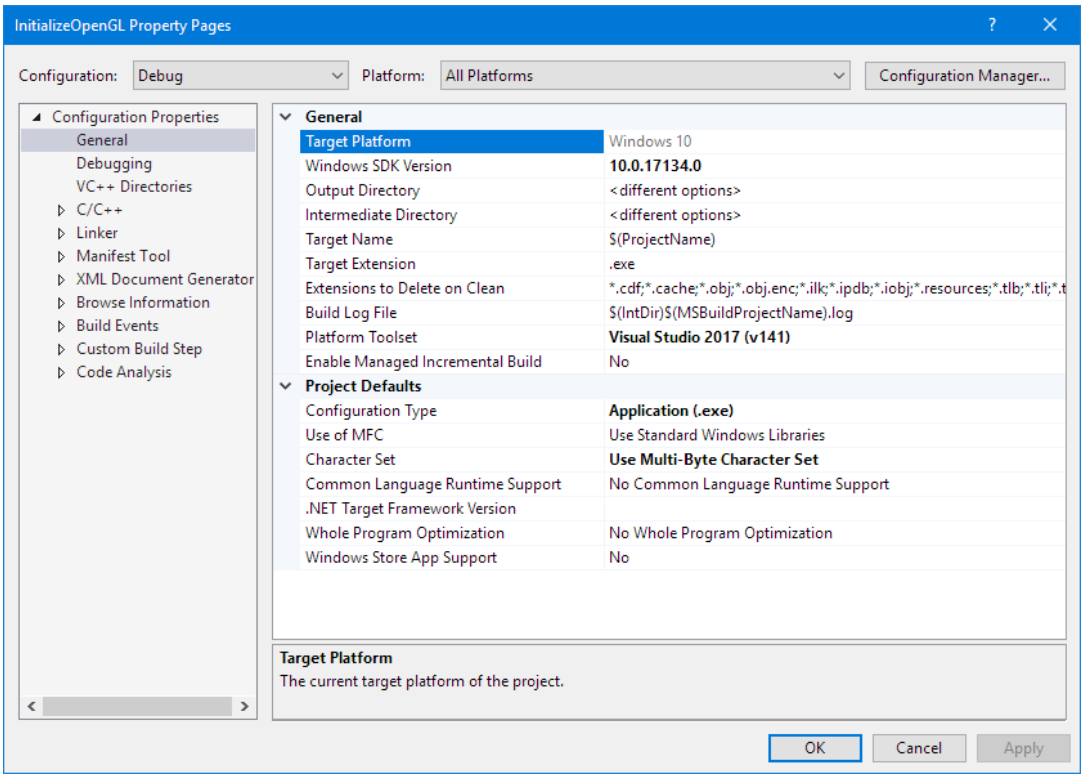

Figure 4: This is what the Project Property Pages should look like.

Okay, so if you feel the last step of alerting you a configuration page is missing is not normal, the author encourages the reader to go report the problem to Microsoft. It is not known why it is happening for some of us.

Next, in the left pane, expand the Linker category node, and select Input. In the right pane, at the top, we need to edit "`Additional Dependencies`." Go ahead and edit the entry, and add "`opengl32.lib`," all in lowercase letters. Finally, click on the Apply button at the bottom of the dialog.

That is all we need to do. We now have completed the "linking" process by setting up Project Property Pages for all platforms, and we have provided a new additional dependencies entry for all platforms to use the OpenGL library file. Under Windows, you need to statically link to a library called `opengl32.lib` (note that you still link to `opengl32.lib` if you're building a 64-bit executable. The "32" part is meaningless). Visual Studio, and most Windows compilers, come with this library.[4]

You're ready to move on, so take a break, drink some lemonade, and play games, before coming back to read the next chapter.

[4] Obtained information from the official OpenGL wiki. (Khronos Group, 2018)

Chapter 2

The Core of this book

You are now coming to terms with Visual Studio. Your mind, your soul, and your motivation now drive you towards the path of understanding what is everything doing behind the scenes. You start to read this paragraph, and you have this sense of curiosity, while being annoyed at the author on why having to put up with all this.

Ease there, my reader, we have to take things slowly. This chapter introduces a lot of OpenGL 4 specific initialization steps, so it makes sense as to why we must digest all of these one by one. The steps are as follows:

- Create a Window context.
- Create a dummy OpenGL context.
- Use dummy context to load OpenGL features newer than OpenGL 1.1.
- Set OpenGL context to use the newer OpenGL versions.

Let us begin the first step, the creation of our window context of our application.

Creation of a Window

One rule of thumb is, before attempting to actually use OpenGL, the program needs to initialize OpenGL. And since OpenGL has many different flavors on different platforms, the initialization process differs between platform to platform. This mainly has to do with the Windows operating system using Direct3D wrappers to wrap OpenGL functions on Windows XP and up. We do not know the reasons why they are doing this, but we are not someone who is knowledgeable on this matter.

Before we delve further, you must know the method of creating an OpenGL context is very platform-specific, and it can also differ for language-specific bindings. If we were to use language-specific bindings, then it is best not to continue further and perhaps use a Window Toolkit for managing this OpenGL context creation task. Commonly known Window Toolkits, designed specifically around creating and managing OpenGL windows, are *freeglut*, *GLFW*, and *GLUT*[5]. Some other Window Toolkits are "multimedia libraries," which not only are designed to create and manage OpenGL windows, but also

[5] *GLUT* is super, super old. Highly recommended not to use it any more in this day and age.

provide input controls, audio, and other tasks useful for “multimedia purposes.” These “multimedia libraries” include *SDL2*, *Allegro 5*, *SFML*, *Ecere SDK*, and more.

The way these Window Toolkits create the OpenGL context is by calling on platform-specific APIs. You may be confused as to why such is the case, so let this book continue to explain. Because OpenGL does not exist until you create an OpenGL context, the OpenGL context creation is not regulated by the OpenGL specifications, but rather it is regulated by platform-specific APIs.

Let us start with the first step, which is to create a handle to a window context, while the handle of the window needs to have its own device context. There are two methods to approach this, the WinMain default entry method, and the Main entry method. Note that the Main entry method will always show a console window in the background, since it is the entry point for console applications. If you do not want to see the console window, use the WinMain entry method instead and set the Subsystem entry in your Project Property Pages to be `/SUBSYSTEM:WINDOWS`. This is set in the System under the Linker node.

The following code snippet is the WinMain method:

```
#include <Windows.h>
#include <cstdio>



LRESULT MyWindowProcess(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
  switch (msg) {
    case WM_CLOSE:
      DestroyWindow(hwnd);
      break;
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    default:
      return DefWindowProc(hWnd, msg, wParam, lParam);
  }
  return 0;
}



int WinMain(HINSTANCE handleCurrentInstance, HINSTANCE handlePreviousInstance, LPSTR
commandLines, int SW_Flag) {
  //Initialize window application.
  WNDCLASSEX windowClass = {};
  windowClass.cbSize = sizeof(windowClass);
  windowClass.lpszClassName = TEXT("My Window Class");
  windowClass.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
  windowClass.lpfnWndProc = MyWindowProcess;
  windowClass.hInstance = handleCurrentInstance;
  windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);

  if (!RegisterClassEx(&windowClass)) {
    char buffer[128];
    std::snprintf(buffer, 128, u8"Window class registration has failed.\r\n");
    OutputDebugStringA(buffer);
    return -2;
  }
```

```
  //Initialize the window instance.
  //This variable will be discussed in the next chapter.
  DWORD requiredWindowStyles = WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_POPUP | WS_BORDER;
  HWND handleWindow = CreateWindowEx(WS_EX_LEFT,
                                     TEXT("My Window Class"),
                                     TEXT("Initialize OpenGL"),
                                     requiredWindowStyles | WS_VISIBLE,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     NULL,
                                     NULL,
                                     handleCurrentInstance,
                                     NULL);
  if (!handleWindow) {
    char buffer[128];
    std::snprintf(buffer, 128, u8"HWND was not created successfully.\r\n");
    OutputDebugStringA(buffer);
    return -3;
  }

  ShowWindow(handleWindow, SW_SHOW);
  UpdateWindow(handleWindow);

  BOOL gotMessage;
  MSG msg;
  while ((gotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && gotMessage != -1)
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  //wParam is the exit code when PostQuitMessage() is called.
  //On 64-bit systems, wParam is 64-bit, but WinMain's function prototype requires
integers. We have to explicitly downcast to get the right value to return.
  return (int) msg.wParam;
}
```

We need to use `std::snprintf()` in order to use the Windows recommended method of outputting messages, by redirecting the message outputs to the Output tab pane via `OutputDebugStringA()` in Visual Studio 2017. Usually, we would use `OutputDebugString()`, but it is best to be explicit about this early on. This code demonstrates how to send messages to the Output tab pane, while being able to use any Unicode characters. All references to Visual Studio 2017 will use "Visual Studio" from here on out.

If you are not sure what these functions are, we highly recommend you to start learning about them on the MSDN Documentations site. They are a great resource in doing Windows programming. We are assuming you know what these functions are, and will leave it to the readers to learn for exercise, as they are out of scope for this book.

But the short gist is that this is how you would set up a very primitive Win32 application for Windows, as it uses all basic knowledge on getting something to show up on the screen.

And, following the code above, this is the Main method:

```
#include <Windows.h>
#include <cstdio>


LRESULT MyWindowProcess(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {
  return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

```
int main() {
  //Gets the hInstance of the current program.
  HINSTANCE handleInstance = GetModuleHandle(NULL);

  //Initialize window application.
  WNDCLASSEX windowClass = {};
  windowClass.cbSize = sizeof(windowClass);
  windowClass.lpszClassName = TEXT("My Window Class");
  windowClass.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
  windowClass.lpfnWndProc = MyWindowProcess;
  windowClass.hInstance = handleInstance;
  windowClass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);

  if (!RegisterClassEx(&windowClass)) {
    char buffer[128];
    std::snprintf(buffer, 128, u8"Window class registration has failed.\r\n");
    std::printf("%s," buffer);
    return -2;
  }

  //Initialize the window instance.
  //This variable will be discussed in the next chapter.
  DWORD requiredWindowStyles = WS_CLIPCHILDREN | WS_CLIPSIBLINGS | WS_POPUP | WS_BORDER;
  HWND handleWindow = CreateWindowEx(WS_EX_LEFT,
                                     TEXT("My Window Class"),
                                     TEXT("Initialize OpenGL"),
                                     requiredWindowStyles | WS_VISIBLE,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     CW_USEDEFAULT,
                                     NULL,
                                     NULL,
                                     handleInstance,
                                     NULL);
  if (!handleWindow) {
    char buffer[128];
    std::snprintf(buffer, 128, u8"HWND was not created successfully.\r\n");
    std::printf("%s," buffer);
    return -3;
  }

  ShowWindow(handleWindow, SW_SHOW);
  UpdateWindow(handleWindow);

  BOOL gotMessage;
  MSG msg;
  while ((gotMessage = GetMessage(&msg, (HWND) NULL, 0, 0)) != 0 && gotMessage != -1)
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  return 0;
}
```

Did you spot any differences in the code? If the programming code is a bit messy, try to type the code down in Visual Studio on your computer, and then reformat the code to your preferred formatting styles. That way, you should be able to read the code much easier than what is shown in the book.

The code above has very minor changes, but if you do see them, the differences are mainly the contents inside the Window Process Callback function (`WindowProc`), the error reporting behavior, and the return values of the entry points. When an error is thrown, the message will be displayed in the background console window, rather than in the Output tab pane in Visual Studio.

In the `WinMain` method, you are required to process `WM_CLOSE` and `WM_DESTROY` explicitly, but in the `main` method, the background console application will handle `WM_CLOSE` and `WM_DESTROY` automatically. You must understand this when initializing OpenGL through either method, and take heed into account how they both are handled differently. When you attempt to implement one of these methods, just keep this advice in mind.

Starting from this point onwards, all of the codes will be the same for both `WinMain` and `main` entry methods, so we will not be displaying the entire source code again. For extra points, look into how to close the background console application window when using the Main entry method, and try to use `FreeConsole()`[6] if it works for you.

A Dummy OpenGL Context

The next step in the list mentioned earlier is to create a dummy OpenGL context for your window context. The word "context" is being thrown around quite often, but it is a legitimate, technical jargon in the world of Windows programming, so this book will try to spare you of ever mentioning this word.

In your window application's initialization code, you probably may have seen the constant, `CS_OWNDC`, being used. It is a Class Style enumeration value, and it tells Windows to allocate a unique device context for each window in the class. A "device context" is a Windows data structure, containing information about the drawing attributes of a device, such as a display or a printer. All drawing calls are made through a device-context object, which encapsulates the Windows APIs for drawing lines, shapes, and text. And since OpenGL is fundamentally a bunch of drawing calls through the Windows API to your graphics card, it makes sense to create a window application with unique device contexts for OpenGL.

Now we can start using the device context that has been allocated for your window application. Your device context contains another data structure called a Pixel Format Descriptor, and its job is to describe the pixel format of a drawing surface your window application will be using when doing drawing, hence the name. Unfortunately, per MSDN documentations, this Pixel Format Descriptor contains a lot of struct members to fill in, so we will not-so-briefly touch on what the requirements for an OpenGL context needs from the Pixel Format Descriptor.

If you want to see the entire Pixel Format Descriptor, a code sample is provided on the next page.

[6] A process can be attached to at most one console. A process can use the `FreeConsole` function to detach itself from its console. If other processes share the console, the console is not destroyed, but the process that called `FreeConsole` cannot refer to it. A console is closed when the last process attached to it terminates or calls `FreeConsole`. (Microsoft, 2018)

What OpenGL needs from the Pixel Format Descriptor are in the following list:

- Certain Pixel Format Descriptor flags
- The format type of your frame buffer to be used for your application.
- The color depth of the frame buffer.
- Number of bits for the depth buffer.
- Number of bits for the stencil buffer.
- Number of auxiliary buffers in the frame buffer.
- Certain struct members are ignored, since they are exclusive to older OpenGL versions.

The last one is important, because a lot of us tend to forget that in Windows programming, backward compatibility made some struct members unused and ignored, for historically valid reasons, and is why it was put into the list, also for the author's own benefit.

The Pixel Format Descriptor

In this section, we take a look at the pixel format descriptor. The code looks like this:

```
PIXELFORMATDESCRIPTOR pfd = {};
// Per MSDN Documentation, provide the size of the pixel format descriptor.
pfd.nSize = sizeof(pfd);
// Per MSDN Documentation, version number is always 1. Struct is never updated at all.
pfd.nVersion = 1;
// Required Flags.
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
// The type of frame buffer. RGBA or color index palette.
pfd.iPixelType = PFD_TYPE_RGBA;
// Color buffer bits count.
pfd.cColorBits = 32;
// Depth buffer bits count.
pfd.cDepthBits = 24;
// Stencil buffer bits count.
pfd.cStencilBits = 8;
```

Some of the values would make you curious, and that is fine. We will go over each struct member and their values.

Starting with flags and their values: The flag, `PFD_DRAW_TO_WINDOW`, is straightforward, as it tells your device context its frame buffer can draw to a window or a device surface. A "device surface," part of the Windows Graphics Device Interface (GDI), is a surface required for drawing and text output, usually referring to your primary surface (on-screen surface) or your display. The other flag, `PFD_SUPPORT_OPENGL`, tells your device context it can support OpenGL specifications that describe the behavior of a rasterization-based rendering system your application can control. This does not signify the application is using hardware-accelerated rendering, nor is doing software rasterizations; It only tells the application it can support this rendering system. The last flag, `PFD_DOUBLEBUFFER`, tells the application the frame buffer is double-buffered, which is also very straightforward.

Next, we have the Pixel Type, which is set to `PFD_TYPE_RGBA`. We can only use either one type or the other, but not simultaneously. `PFD_TYPE_RGBA` is your typical RBGA color hex values, in Big Endian byte order. For `PFD_TYPE_COLORINDEX`, the color-index mode specifies colors in a logical palette with an index to a specific logical-palette entry. Most GDI programs use color-index palettes, but the RGBA mode works better for OpenGL for several effects, such as shading, lighting, fog, and texture mapping. If having the truest color is not critical for your OpenGL application, you might choose to use the color-index mode. Example usage includes creating a topographic map that uses "false color" to emphasize the elevation gradient.

If you want other reasons for choosing color index mode in an OpenGL application, there is a dedicated chapter, Chapter 5, of the *OpenGL Programming Guide: Red Book*, which explains some of the reasons and shows how color-index mode is done. It is under the section, titled, "*Lighting in Color-Index Mode*."[7]

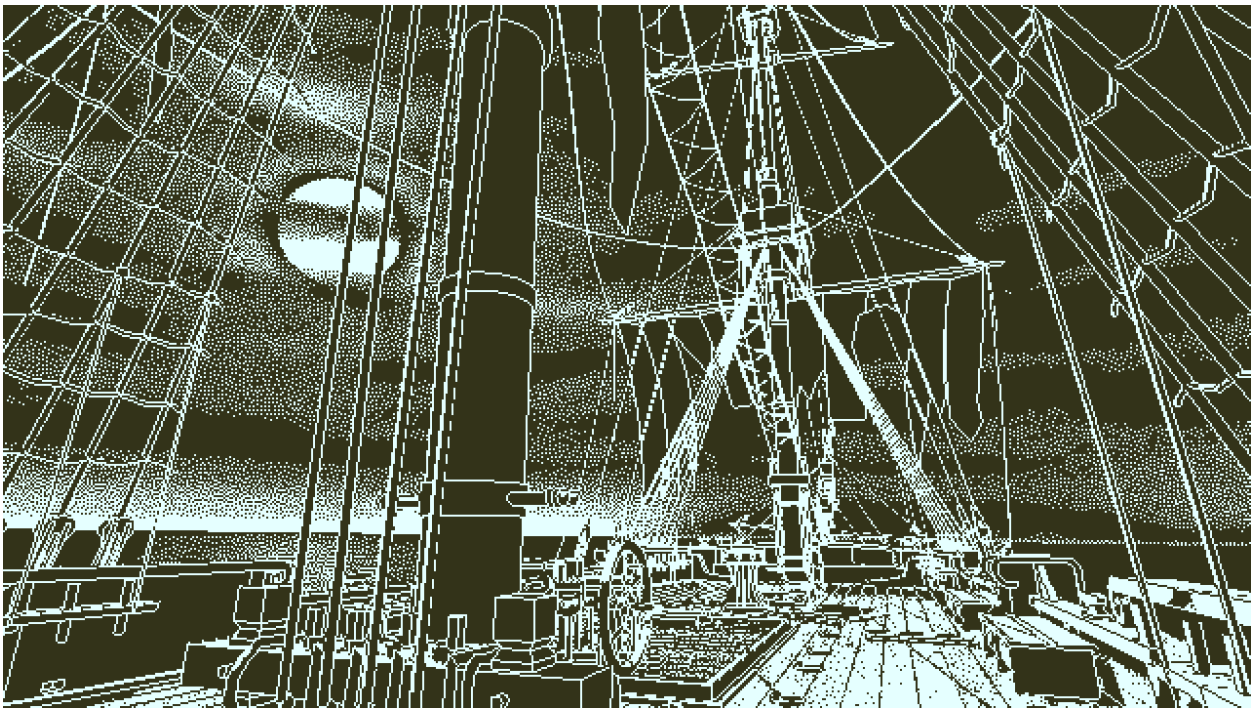

Figure 5: Screenshot of Return of Obra Dinn. A prime example of a game that uses 1-bit color depth.

The next struct member is the color bits. This is the number of bits that we can use to store a color pixel value as a whole number, or in other words, the number of bits used for each color component of a single pixel. Here, we set the value to be 32, or 32 bits, so it can use "True color"[8]. You do not always

[7] As of writing, the author only remembered it to be in Chapter 5. Perhaps, in later editions of the Red Book, the chapter number has been moved around, thus the title of the section is provided, just in case.

[8] "True color" here refers to a 24-bit color component and an optional 8-bit alpha channel. The Alpha value does not have an intrinsic meaning; it only does what the shader that uses it wants to. Usually, Alpha is used as a translucency value, but do not make the mistake of confining your thinking to just that. Alpha means whatever you want it to. (Khronos Group, 2011)

have to set it to 32, because there are many other types of color depth formats you can use that is higher or lower than 32. For instance, your OpenGL application may use "Deep colors," which could be of 30, 36, or 48 bits, allowing your application to process over 1 billion different colors. Another instance, is a very stylized game, *Return of Obra Dinn*, that uses 1-bit colors.

Also, since we have set the color format to use 32 bits, this means we need to use the RGBA format with the correct byte order. Note that RGBA color format is the only format that supports 32 bits for OpenGL, and the byte order should be converted to the byte order OpenGL expects it to be (RGBA format).

The next two values to set are the number of bits for the depth buffer, and the number of bits for the stencil buffer. Unlike the color depth, which is an integer value that contains the color components and alpha values, the depth buffer bits value is referring to the depth precision of the Z-buffer. These two buffers are both typically packed into a 32-bit normalized unsigned integers depth format, and would result in both of them sharing the same memory space. This is where you can sometimes see tutorials and resources mentioning about giving them a value of 24 to represent a 24-bit depth buffer, and a value of 8 for the stencil buffer.

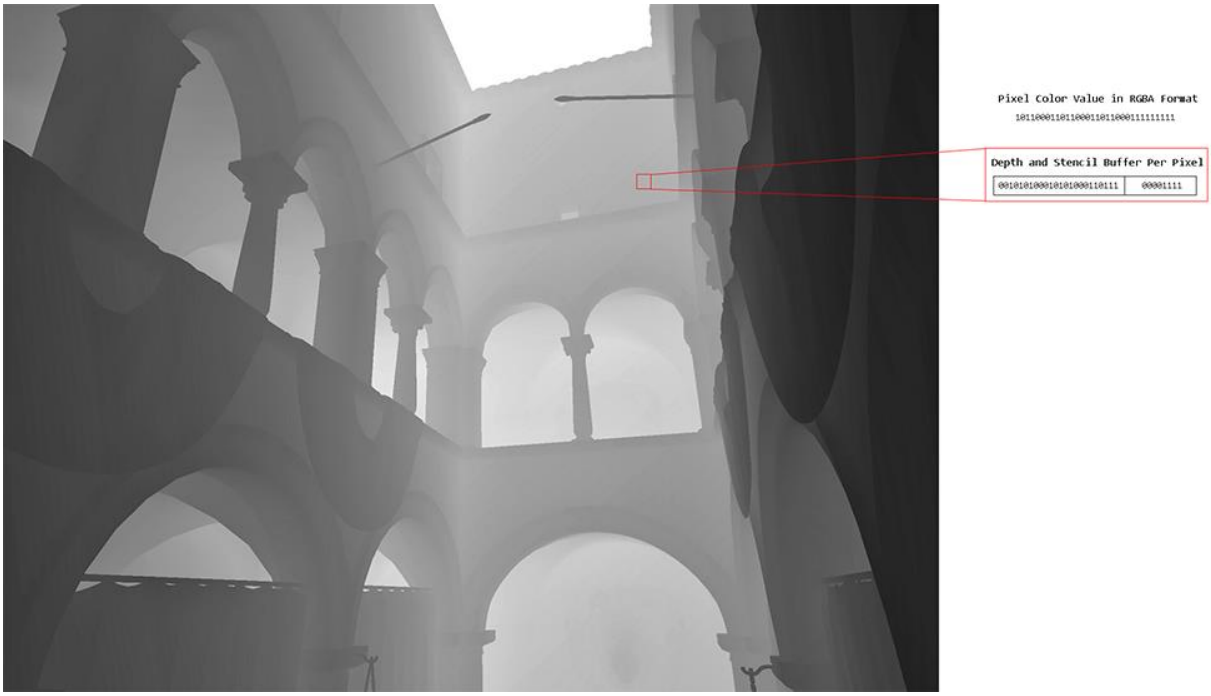

Figure 6: Depth buffer visualization of the Sponza scene (Lampert, 2014). This book added what a depth buffer and stencil buffer look like when viewing at a shared memory. The values are only for visualization, not accurate representations.

OpenGL do support 32-bit floating point values (32-bit floats) in the depth buffer, starting from OpenGL 2.0, introduced as an extension. On modern GPUs, the recommended method is to use a 32-bit float buffer, and an 8-bit unsigned integer buffer to handle high-precision depth tests. The 32-bit float buffer is typically stored as a texture of type `GL_R32F`, which is a single-channel (Red) 32-bit float texture, and then the texture is written into a frame buffer object (FBO), before being passed into a fragment

shader. The stencil buffer is then passed into the fragment shader as a separate buffer, and finally the depth and stencil values are used to calculate the final depth values of each pixel inside the fragment shader.

However, we can choose to use the hardware-supported 24-bit depth buffer and 8-bit stencil buffer for low precision depth tests, and use those buffers to determine the depth values in the fragment shader. In all cases, the 8-bit stencil buffers have no advantages over the other. Due to the range of precision, a 24-bit depth buffer would introduce Z-fighting, artifacts of pixels fighting over one another on the display screen, more often around areas closer to the far plane of your camera frustum, than if you use a 32-bit depth buffer. So, keep this in mind.

In conclusion, the depth buffer bit value is limited to 24 or 32 for practical reasons, and the stencil buffer is limited to multiples of 8. The larger the depth buffer bit values are given, the more graphical resources to measure the precise depths per pixel is needed for the application.

Understanding Pixel Formats

By the time we understand the basic, necessary properties of a pixel format descriptor, it is time to apply the pixel format descriptor by choosing the pixel format. But before we continue, we first need to understand some terminology used when describing pixel formats.

There are two separate definitions used by OpenGL to fully specify a pixel format for the CPU to understand during processing, and this is set in our pixel format descriptor. There is the `format` and the `type`. (Frantzis, 2018)

The `format` specifies the logical order of components in the pixel format. It follows the scheme or syntax:

```
GL_{COMPONENT LIST}
```

Examples are as follows:

```
GL_RGBA  //The pixel format consists of four components in the logical order R, G, B, and A, respectively.
GL_BGRA  //The pixel format consists of four components in the logical order B, G, R, and A, respectively.
GL_RED   //The pixel format consists of a single component, R.
```

The `type` specifies the size and memory representation of the components specified in `format`. It has two forms, non-packed representations and packed representations.

Non-packed representations are ordered in memory in the order specified by `format`. How each component is stored in memory depends on the system's endianness.

The non-packed representations of `type` follow the scheme or syntax:

```
GL_{DATA TYPE}
```

The `{DATA TYPE}` specifies the data type of each component. Examples are as follows:

```
GL_UNSIGNED_BYTE  //Each component is stored as an unsigned 8-bit binary integer. This is
not affected by the system's endianness.
GL_HALF_FLOAT     //Each component is stored as an IEEE-754 16-bit floating-point value.
This is affected by the system's endianness.
GL_FIXED          //Each component is stored as a signed 16.16 two's complement fixed
point binary integer. This is affected by the system's endianness.
```

Packed representations of type follow the scheme or syntax:

```
GL_{DATA TYPE}_{COMPONENT BITS}(_REV)
```

The `{DATA TYPE}` specifies the data type used to hold the pixel data as a whole. The `{COMPONENT BITS}` specify the number of bits each component occupies in the data type, and is read from left to right. The leftmost number corresponds to the size of the component which will occupy the most significant bits of the data type. The rightmost number corresponds to the least significant bits, respectively.

The `(_REV)` is a notation used in the core profile specification, starting in OpenGL 4.5. Types whose token names end with `_REV` "reverse" the component packing order from least to most significant bit locations (Khronos Group, 2017). Meaning that, if the `type` does not have the `_REV` suffix, then the first component occupies the most significant bits and the last component occupies the least significant bits. Visually, the format `GL_XYZW` will be mapped to the data type `0xXYZW`. If the `type` has the `_REV` suffix, then the first component occupies the least significant bits, and the last component occupies the most significant bits. Visually, the format `GL_XYZW` will be mapped to the data type as `0xWZYX`, hence the `_REV` designation, which is short for "reversed."

The memory layout of a packed pixel format depends on the system's endianness, and has nothing to do with whether the component list is in reversed order or not.

Examples are as follows:

```
GL_UNSIGNED_SHORT_5_6_5       //16-bit value. First component: 11-15 bit. Second
component: 5-10 bit. Third component: 0-4 bit.
GL_UNSIGNED_SHORT_5_6_5_REV   //16-bit value. First component: 0-4 bit. Second component:
5-10 bit. Third component: 11-15 bit.
```

By combining a `format` and a `type`, we get a full specification of the pixel format. For packed representations, the number of components in `format` and `type` must match.

For example, the combination `GL_RGBA` with `GL_UNSIGNED_BYTE` specifies a pixel format laid out in memory as the bytes R, G, B, and A respectively, with R at the lowest memory address, and A at the highest memory address. This combination is not affected by the system's endianness.

As another example, the combination `GL_RGB` with `GL_UNSIGNED_SHORT` will have a memory layout that depends on the system's endianness. On Little-Endian systems, the memory layout is stored like so:

R_0 R_1 G_0 G_1 B_0 B_1

On Big-Endian systems, the memory layout is stored like so:

R_1 R_0 G_1 G_0 B_1 B_0

The subscripted numbers, $_0$ and $_1$, represent the first and second bytes of their respective components.

For a completer and more extreme example, the combination `GL_BGRA` with `GL_UNSIGNED_INT_10_10_10_2` is represented as follows, with M for most significant bit, and L for least significant bit:

M L
$B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0G_9G_8G_7G_6G_5G_4G_3G_2G_1G_0R_9R_8R_7R_6R_5R_4R_3R_2R_1R_0A_1A_0$

The memory layout of this combination depends on the system's endianness. On Little-Endian systems, the memory layout is stored like so:

0 1 2 3
M L M L M L M L
$R_5R_4R_3R_2R_1R_0A_1A_0$ $G_3G_2G_1G_0R_9R_8R_7R_6$ $B_1B_0G_9G_8G_7G_6G_5G_4$ $B_9B_8B_7B_6B_5B_4B_3B_2$

On Big-Endian systems, the memory layout is stored like so:

0 1 2 3
M L M L M L M L
$B_9B_8B_7B_6B_5B_4B_3B_2$ $B_1B_0G_9G_8G_7G_6G_5G_4$ $G_3G_2G_1G_0R_9R_8R_7R_6$ $R_5R_4R_3R_2R_1R_0A_1A_0$

So, what would happen if we use the combination `GL_RGBA` with `GL_UNSIGNED_INT_2_10_10_10_REV`? Its representation is as follows:

```
M                                                               L
A1A0B9B8B7B6B5B4B3B2B1B0G9G8G7G6G5G4G3G2G1G0R9R8R7R6R5R4R3R2R1R0
```

The memory layout on Little-Endian systems:

```
0               1               2               3
M             L M             L M             L M             L
R7R6R5R4R3R2R1R0 G5G4G3G2G1G0R9R8 B3B2B1B0G9G8G7G6 A1A0B9B8B7B6B5B4
```

And the memory layout on Big-Endian systems:

```
0               1               2               3
M             L M             L M             L M             L
A1A0B9B8B7B6B5B4 B3B2B1B0G9G8G7G6 G5G4G3G2G1G0R9R8 R7R6R5R4R3R2R1R0
```

And what about a combination that is invalid, such as `GL_RGB` with `GL_UNSIGNED_INT_8_8_8_8`? This combination is invalid because `GL_RGB` is a 3-component `format` and `GL_UNSIGNED_INT_8_8_8_8` is a 4-component packed `type`. Invalid combinations, such as this, will cause your application to emit a segmentation fault when the graphics card is attempting to process the pixel format.

Now that we fully and clearly understand the terminology used to describe pixel format, we can now continue on the subject.

On any Windows platform, Microsoft's implementation of OpenGL uses the Pixel Format Descriptor data structure, `PIXELFORMATDESCRIPTOR`, to convey pixel format data. The structure's members specify the preceding properties and several other pixel formats. A given device context can support several pixel formats. The Windows platform identifies the pixel formats that a device context supports, with consecutive one-based index values (1, 2, 3, 4, ...). A device context can have just one current pixel format, chosen from the set of pixel formats that device context supports. (Microsoft, 2018)

Each window has its own current pixel format in OpenGL on the Windows platform. This means, for example, that an application can simultaneously display RGBA and color-index OpenGL windows, or single-buffered and double-buffered OpenGL windows. This per-window pixel format capability is limited to OpenGL windows on the Windows platform.

On the same topic, do you recall we assigned a variable, `requiredWindowStyles`, when we were creating our window? Since an OpenGL window has its own pixel format, only device contexts retrieved for the client area of an OpenGL window are allowed to draw into the window. Additionally, the window class attribute should not include the `CS_PARENTDC` style, for the same reason that OpenGL applications can only use their own device context, and not the parent window's device context.

It is for the reason mentioned above, where we would need to specify the window styles, `WS_CLIPCHILDREN` and `WS_CLIPSIBLINGS`[9], to prevent any child windows from clipping draw calls over to the parent window, and letting other sibling windows clip the parent window during transitions from fullscreen mode to windowed mode, and vice versa. `WS_POPUP` makes your window become a generic, bare-minimum window, with no Close and Minimize/ Maximize buttons, no title bar, no borders, and no window frame to denote where the application is located and how big it is on your screen. `WS_BORDER` would display the title bar and window frame, but it will not show the Close and Minimize/Maximize buttons.

And the reason why a window can only be set to have only one pixel format, is if a window has multiple pixel formats being set, it will lead to significant complications for the Windows platform service, Desktop Window Manager (DWM), and for multithreaded applications. And therefore, it is not allowed for any window to have more than one pixel formats being set.

The next typical steps are, we obtain the device context, then create a pixel format descriptor, then set the device context's pixel format, and finally create an OpenGL rendering context suitable for that device. We set the pixel format before the rendering context because the rendering context inherits the device context's pixel format.

With our `PIXELFORMATDESCRIPTOR` struct ready, we need to then convert this into a pixel format index, the number that was mentioned earlier, where the Windows platform would identify the pixel formats with.

Choosing and Setting a Pixel Format

It is time for us to choose a pixel format, and then set the chosen pixel format to our device context, based on one of the many pixel formats our device context can support. We can easily do this by first choosing our pixel format, then finally setting the returned pixel format. This is done like so:

```
SetPixelFormat(hDeviceContext, ChoosePixelFormat(hDeviceContext, &pfd), &pfd);
```

The `ChoosePixelFormat` function attempts to match an appropriate pixel format supported by a device context to a given pixel format specification, or in our case, our pixel format descriptor (Microsoft, 2018). If the function succeeds, the return value is a one-based pixel format index that is the closest match to the give pixel format descriptor, and this pixel format index is required for `SetPixelFormat` function to set it

[9] More details about transitioning from fullscreen mode to windowed mode and back, will be covered in a later chapter. Apparently, while doing research on this topic, it seemed no one since 2009 knows the original reason of putting those Window Style flags in, and why it must be done like this in the past, other than referring to what MSDN states in the documentation. OpenGL applications, in the past, were integrated with Microsoft Foundation Classes (MFC), and were allowed to run in single-document interfaces (SDI) or multiple-document interfaces (MDI). `WS_CLIPCHILDREN` and `WS_CLIPSIBLINGS` are used for MDI windows, and the flags do not need to be specified for SDI windows. However, the flags are still set by OpenGL toolkits, and the author's speculation is they are there not because of legacy reasons, but rather Windows Graphics Device Interface (GDI) must know whether the parent window needs to clip over the children and sibling windows, if those were created.

(Microsoft, 2018). If the function fails, it will return a zero, which is conveniently a `NULL`, that `SetPixelFormat` can process, and this is something that we need to handle, to notify to the user that their device context fails to support our necessary pixel format for OpenGL. This is the full error handling code:

```
//Very fancy and professional error handling function code, courtesy from Microsoft.
void ErrorExit(LPCTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM |
                  FORMAT_MESSAGE_IGNORE_INSERTS,
                  NULL,
                  dw,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPTSTR) &lpMsgBuf,
                  0,
                  NULL);

    // Display the error message and exit the process
    lpDisplayBuf = (LPVOID) LocalAlloc(LMEM_ZEROINIT, (lstrlen((LPCTSTR) lpMsgBuf) +
lstrlen((LPCTSTR) lpszFunction) + 40) * sizeof(TCHAR));
    StringCchPrintf((LPTSTR) lpDisplayBuf, LocalSize(lpDisplayBuf) / sizeof(TCHAR),
TEXT("%s failed with error %d: %s"), lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR) lpDisplayBuf, TEXT("Error"), MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
    ExitProcess(dw);
}

//Choosing and setting the pixel format
if (!SetPixelFormat(hDeviceContext, ChoosePixelFormat(hDeviceContext, &pfd), &pfd)) {
    //If SetPixelFormat() returns FALSE, we process the error.

    //It is up to the reader to choose to use what Microsoft provided for use...
    if (true)
        ErrorExit(TEXT("SetPixelFormat"));
    else {
        //Or we go and use the tried and true, old-fashioned way.
        char buffer[128];
        std::snprintf(buffer, 128, u8"InitializeOpenGL: Pixel Format is not supported by
device context.\r\n");
        OutputDebugStringA(buffer);

        //Return whatever value you wish here.
        return -4;
    }
}
```

And to use this ErrorExit() function, we need to add `#include <strsafe.h>` and the following code that will generate an error on demand. We need to add `<strsafe.h>` is because it involves a Win32 specifically-tailored varargs `printf()` function.[10]

```
#include <strsafe.h>

void main()
{
	// Generate an error or do something wrong
	if (!GetProcessId(NULL))
		//We invoke the ErrorExit() function like so:
		ErrorExit(TEXT("GetProcessId"));
}
```

Just a friendly reminder, you can only choose either the Microsoft's error handling code snippet, or a very basic error handling code, never both. It is encapsulated with an `if` statement with an always `true` condition, because the author thinks what Microsoft has provided is already official enough for the task at hand.

With the pixel formats finally out of the way, it is time for us to dive into the usage of our own dummy context, to create our actual OpenGL context.

Initialization of OpenGL

This next step requires the use of our `hDeviceContext`, which holds our pixel format we have chosen in the previous section. What we need to do here, is we want to be able to "grab" out the necessary functions to create the OpenGL context. But before we do this, we need to first "grab" out the functions we need to start the "grabbing" process. Did you get it?

It is quite simple.

[10] This specifically-tailored `printf` function is made to avoid a security vulnerability, widely known in the programming circles as the "Uncontrolled String Format Buffer Overflow Vulnerability," introduced as part of the K&R C standard specifications. That is when the C programming language included the standard input and output library header, and introduces the `sprintf` and its variable argument variants that were added into the common standard header `<stdio.h>`. The history itself is murky on when the vulnerability was discovered, as information security reports were not properly documented at that time. More information can be found in the Appendix, where the author documents the findings there.

The proper procedure for "grabbing" out the necessary function is to invoke the call to `LoadLibrary()`, a function which is used to load up a dynamically linked library object, or DLL object for short. But which one of the `LoadLibrary()` function do we use?

Per Microsoft's documentation, there are 4 known `LoadLibrary` functions, `LoadLibraryA`, `LoadLibraryExA`, `LoadLibraryW`, and `LoadLibraryExW`. Each of the function differs slightly in that the parameters to be passed in depends on the encoding of the string representation, as well as what optional information should the library need to know before we attempt to load the DLL object. We'll go over only 2 of them briefly, the `LoadLibraryA` and `LoadLibraryExA`. The `A` versions are for codes where the encoded strings are ANSI strings. ANSI strings are referred to officially as the Windows Codepage 1252 strings, and they are strings that uses characters from the extension of the ASCII character set. The `W` versions are for code where the string encodings are multi-byte characters (or wide characters), or when encoded as UTF-16 LE.

There is a whole slew of information added to the Appendix, and it includes relevant information on how to convert strings from multi-byte characters (UTF-16 LE, or wide characters) to the universally recommended UTF-8 format, and back.

sadf

Reminder

Write code on using ChoosePixelFormat

Write code on using SetPixelFormat

Bookmark

https://www.khronos.org/opengl/wiki/Creating_an_OpenGL_Context_(WGL)

https://docs.microsoft.com/en-us/windows/desktop/api/wingdi/nf-wingdi-choosepixelformat

https://blogs.msdn.microsoft.com/oldnewthing/20030723-00/?p=43073/ (Program template)

https://blogs.msdn.microsoft.com/oldnewthing/20100412-00/?p=14353 (Toggling Fullscreen/Window)

https://blogs.msdn.microsoft.com/oldnewthing/20050505-04/?p=35703/ (Hiding taskbar when in fullscreen)

References

B. W. Kernighan, D. M. (1978). *The C programming language.* Upper Saddle River, New Jersey, USA: Prentice-Hall, Inc. Retrieved October 20, 2018

Betoseha. (2015, April 20). *CJKV variant glyphs.png*. Retrieved from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:CJKV_variant_glyphs.png

Bright, P. (2013, June 28). *C99 acknowledged at last as Microsoft lays out its path to C++14*. Retrieved from Ars Technica: https://arstechnica.com/information-technology/2013/06/c99-acknowledged-at-last-as-microsoft-lays-out-its-path-to-c14/

Frantzis, A. (2018, July 13). *OpenGL pixel formats*. Retrieved from Pixel Format Guide: https://afrantzis.com/pixel-format-guide/opengl.html

Github. (2016, September 29). *Installing opengl with newer Windows SDK*. Retrieved from Github: https://github.com/Microsoft/vcpkg/issues/99

IETF. (2003, November). UTF-8, a transformation format of ISO 10646. *Request for Comments: 3629* . Montreal, Quebec, Canada.

ISO. (1993, January 21). UCS Transformation Format One (UTF-1). *ISO/IEC 10646, 1st*. ISO/IEC. Retrieved from https://web.archive.org/web/20150318032101/http://kikaku.itscj.ipsj.or.jp/ISO-IR/178.pdf

Khronos Group. (2011, October 17). *Image Format*. Retrieved September 16, 2018, from OpenGL Wiki: https://www.khronos.org/opengl/wiki/Image_Format#Color_formats

Khronos Group. (2017). *The OpenGL Graphics System: A Specification Version 4.5 (Core Profile).* (M. Segal, & K. Akeley, Eds.) Khronos Group Inc., The. Retrieved from https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf

Khronos Group. (2018, September 8). *Getting Started.* (D. Photon, Editor) Retrieved September 8, 2018, from OpenGL Wiki: https://www.khronos.org/opengl/wiki/Getting_Started#Windows

Lampert, G. R. (2014, January 26). *Visualizing the Depth Buffer*. Retrieved from Lampert's game dev journal: http://glampert.com/2014/01-26/visualizing-the-depth-buffer/

Microsoft. (2018, August 29). *ChoosePixelFormat function*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/windows/desktop/api/wingdi/nf-wingdi-choosepixelformat

Microsoft. (2018, July 11). *FreeConsole function.* Retrieved from MSDN Documentations: https://docs.microsoft.com/en-us/windows/console/freeconsole

Microsoft. (2018, August 29). *Microsoft Docs*. Retrieved from SetPixelFormat function: https://docs.microsoft.com/en-us/windows/desktop/api/wingdi/nf-wingdi-setpixelformat

Microsoft. (2018, May 30). *Pixel Formats*. Retrieved from Microsoft Documentations: https://docs.microsoft.com/en-us/windows/desktop/opengl/pixel-formats

Microsoft. (2018, August 28). *SetWindowPos function.* Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-setwindowpos

Microsoft. (2018, August 28). *ShowWindow function*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-showwindow

Microsoft. (2018, May 30). *Windows Coding Conventions*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/windows/desktop/learnwin32/windows-coding-conventions

Unicode Consortium. (2015, November 18). *Unicode History Corner*. Retrieved October 20, 2018, from Unicode: https://www.unicode.org/history/

Unicode Consortium. (2017). *ISO/IEC 10646:2017*. Mountain View, CA: Unicode Consortium.

W3Techs. (2018, October 28). *Usage of character encodings broken down by ranking*. Retrieved from Web Technology Surveys: https://w3techs.com/technologies/cross/character_encoding/ranking

Wikipedia. (2002, March 13). *Hungarian notation*. Retrieved September 30, 2018, from Wikipedia: https://en.wikipedia.org/wiki/Hungarian_notation

Appendix

Things You Can Learn!

The Appendix is devoted to research findings and notes gathered for the creation of this book. There are no citations nor references in this section, hence the Reference table is in a previous section. All findings in this section are in no way considered as the "official" nor "recommended" methods to understand how the code works. This section is randomly listed out of order of importance and chronological order. Most of the information are found in the MSDN, with very few of them listed in the footnotes.

And yes, this disclaimer is there so the book can put you in ease. Now let us continue.

A Hungarian Notation Trick

Hungarian Notation is an identifier naming convention in computer programming, in which a name of a variable or function indicates its intended purpose. It is composed with a series of initials or characters in variables, functions, structs, and other API platform-centric data types. In Windows Win32 C programming, it is widely known Microsoft used Hungarian Notation for naming them structs and variables, and is scattered throughout their API functions and parameter or argument names. This section is devoted to the Windows Coding Conventions. (Microsoft, 2018)

Obviously, the solution is to look up the names of the API functions on the MSDN Documentations, and then scan and search for the answer, but that will not teach you how to utilize Hungarian Notations well. This little section is devoted to teaching you the basics of "Deriving the Hungarian Notations." But before we get to there, we need to learn about what the Hungarian Notation is composed of. We can learn how developers in the past were able to utilize Hungarian Notations. There is no need to learn about the history of Hungarian Notations and the rise and fall of usage in programming languages through the ages.[11]

For example, you may have seen the use of defining or assigning a string variable with the type, `LPSTR`. LP refers to "Long Pointer," STR refers to "STRing."

Another example is the variable name, `pwsz`, all in lowercase Hungarian Notation. P stands for Pointer. W stands for Wide Chars, which are characters with the length of 2 bytes per character, and mostly correlates to the multi-byte characters. SZ stands for Strings, pronounced as STRINGZ. You will see this and similar naming schemes more often with temporary variables. An example is given on the next page.

[11] Wikipedia has an entire article dedicated to Hungarian Notations. (Wikipedia, 2002)

```
LPCWSTR pwsz = NULL;
```

The lowercase letters here are identifiers in Hungarian Notation form, and it is followed by an identifier in UpperCase notation. With conjunction, the entire name is a camelCase identifier, and we can then use Hungarian Notations as "clues" on what the "identifier" is supposed to do.

The secret trick to quickly learning Hungarian Notation is its mnemonics, which is very obvious to any person who has read a single page of MSDN Documentations about Win32 programming. But, the mnemonics is not there for show, rather it is useful to know as a way to decipher constants, variables, enumerations, and structs, to Win32 API functions.

For example, we have a function, `AdjustWindowRect(LPRECT, DWORD, BOOL)`, that takes in a `LPRECT`, a `DWORD`, and a `BOOL`. When we look at the function's prototype, we can see the parameter list is named as such:

```
AdjustWindowRect(LPRECT lpRect, DWORD dwStyle, BOOL bMenu)
```

To explain the parameter types, `LPRECT` refers to a pointer to a `RECT` struct data, `DWORD` refers to a "double word" or a 32-bit unsigned binary integer[12], and a `BOOL` integer value signifying `TRUE` or `FALSE`. To explain the parameter names, `lpRect` refers to "Long Pointer To `RECT`," `dwStyle` refers to "Double Word of Window Styles," and `bMenu` refers to "`BOOL` flag for Menu."

We can immediately see `lpRect` is a pointer to a `RECT`, but we would probably never know what `dwStyle` and `bMenu` is. The trick here is to ignore the Hungarian Notation, and think what the remaining words would match up with the mnemonics. This is how we break the problems down.

"Style" would make you think it has to do with Window Styles, because it is the only place where "Styles" as enumerations would make sense. Why is "dw" added to the "Style"? Since "dw" is `DWORD`, and `DWORD` are usually related to 32-bit unsigned integer constants, something closely resembles a constant and it describes a "style" would point to Window Styles.

"Menu" would make you think about System Menu, which is basically an old way of saying the Menu Bars, or the Window Toolbars. Why is "b" added to the "Menu"? Usually, we think of "b" as `BOOL`, and whenever `BOOL` is used, it is mostly there to tell us an object is enabled or disabled, or the object exists or does not exist. So, the "Menu" may be enabled or disabled, or the fact it can exist or be gone.

By the end of this, we can then learn to derive our findings in our head, and start to look for the necessary information. In this case, it is one or many of Window Styles, and whether we want to include the Window Toolbar into calculation to adjust the `RECT` struct passed in.

Another thing about Hungarian Notation that also uses the method of deriving information explained above, is figuring out how to pinpoint which struct tags are used and whom they belong to. Here is another example, using multiple functions, `SetWindowPos`, and `ShowWindow`.

[12] This information is found in the Microsoft API and Reference Catalog, in the Open Specifications subsection, under Protocols, categorized in Windows Protocols, subcategorized inside Windows References, and labeled as [MS-DTYP] Windows Data Types.

Taken from MSDN, `SetWindowPos` function changes the size, position, and Z-ordering of a child window, a popup, or a top-level window. These windows are ordered according to their appearance on the screen. The topmost window receives the highest rank and is the first window in the Z-order (Microsoft, 2018). `ShowWindow` function sets the specified window's Show state (Microsoft, 2018).

The common Hungarian Notation used for both of these functions all start with "SW." For `SetWindowPos`, all enumerations and constants associated with `SetWindowPos` begins with "SWP," and for `ShowWindow`, all similar data associated with `ShowWindow` begins with "SW." Microsoft took advantage of their proprietary feature in their Visual Studio suite, IntelliSense, by allowing those who write Hungarian Notation form to easily sort the associated identifier names together, creating a coherent and convenient way to easily see what the options are.

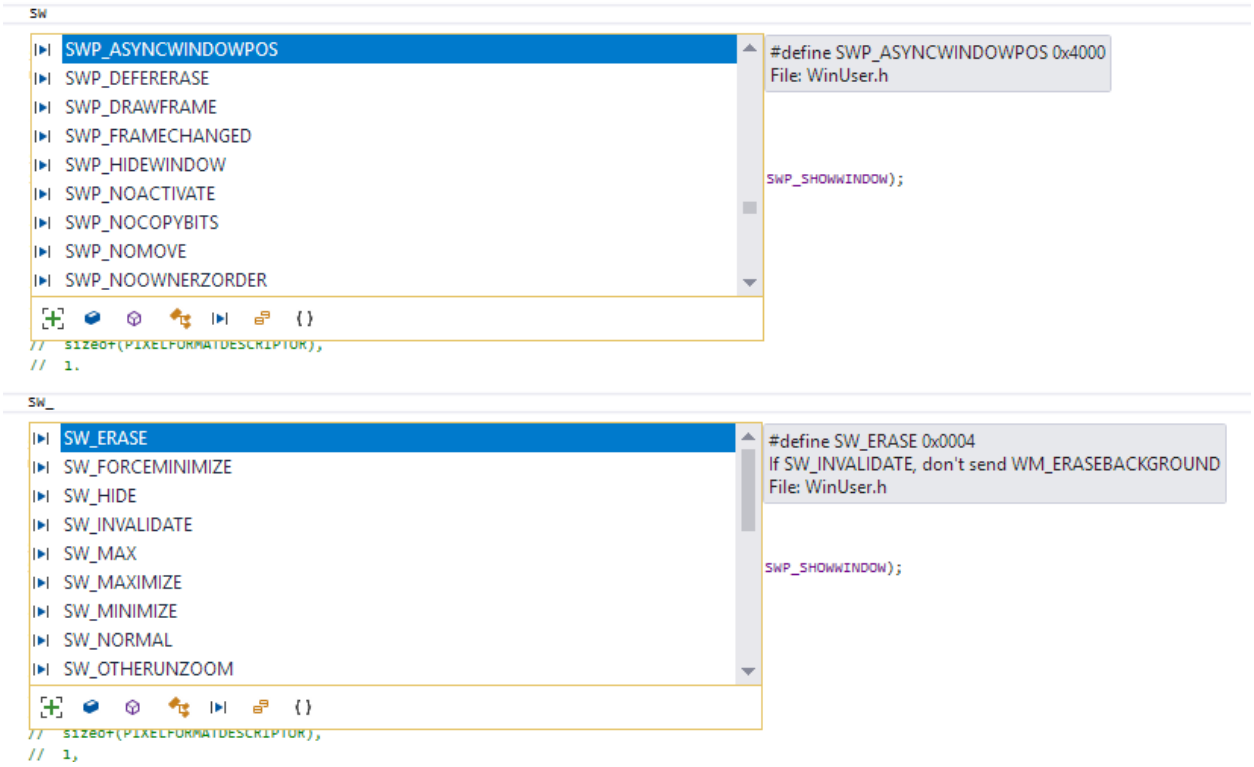

Figure 7: Using IntelliSense, associated constants are easily sorted by alphabetical order, making them easy to search and navigate through. Top section is for `SetWindowPos` function. Bottom section is for `ShowWindow` function. The only difference is having an underscore typed out, marking the end of the Hungarian Notation.

One can imagine this to be somewhat useful, but still requiring rote memorization of the mnemonics used to label the constants, enumerations, and struct tags. This is dependent on skills and experience, and as such, there is no guarantee that this will work. It is, as such, merely a "trick."

Documented Quirks

This section is dedicated to learning hacks, quirks, workarounds, and weird undefined behaviors a programmer would normally never suspect the code to be the case. All of the unusual behaviors we do not normally expect will be called “quirks” for this section. These quirks are usually found when reading the Remarks section in some of the documentation articles provided by Microsoft. Note that it is not just Microsoft, but also it could be from many other vendors and manufacturers, who would provide code access to certain behaviors, either for good intentions, unintentionally allowing code to execute past its intended scopes, or it could be some oversight that is hard to fix.

All known quirks will be listed in this section. It is not a substitute for answering problems that may arise while the reader is attempting to learn how to initialize OpenGL, but at least this book will provide some level of assistance. At least.

ShowWindow Getting Ignored on First Invocation

This is what MSDN Documentation has to say in regards to the `ShowWindow` function, which is used when initializing and creating a window [Microsoft, 2018]:

> The first time an application calls `ShowWindow`, it should use the `WinMain` function's `nCmdShow` parameter as its `nCmdShowparameter`. Subsequent calls to `ShowWindow` must use one of the values in the given list, instead of the one specified by the `WinMain` function's `nCmdShow` parameter.

This remark essentially means, when the first `ShowWindow` function is called, it considers the fact the parent of the window may be setting up some startup information to be passed to the child. The child window will have to obey the parent window’s intention, and then becomes defiant by choosing what to do after that.

This behavior is designed for the following situations, per Microsoft’s remarks:

> - Applications create their main window by calling `CreateWindow` with the `WS_VISIBLE` flag set.
> - Applications create their main window by calling `CreateWindow` with the `WS_VISIBLE` flag cleared, and later call `ShowWindow` with the `SW_SHOW` flag set to make it visible.

In our window initialization code in the beginning of Chapter 2, we actually use `WS_VISIBLE` in our window class initialization. It was more about the author just wanting to rush ahead, and get to the point as quick as possible. Not that big of a deal to begin with, when all it takes is knowing you have to call on `ShowWindow` function again after the first function invocation, as mentioned below:

> As noted in the discussion of the `nCmdShow` parameter, the `nCmdShow` value is ignored in the first call to `ShowWindow` if the program that launched the application specifies startup information in the structure. In this case, `ShowWindow` uses the information specified in the `STARTUPINFO` structure to show the window. On subsequent calls, the application must call `ShowWindow` with `nCmdShow` set to `SW_SHOWDEFAULT` to use the startup information provided by the program that launched the application.

And that is there is to it.

The “sprintf” History

This is in no way an attempt at rewriting history, but rather this is merely documenting the findings on the history of the sprintf by utilizing Google search. It has led to a slew of rabbit holes, such that it may be necessary to document this in an Appendix for future references.

The history all begin with the introduction of the C programming language specification, published by Brian Kernighan and Dennis Ritchie on February 22, 1978 (B. W. Kernighan, 1978). At the time of writing, the authors of K&R C specifications wrote their codes and examples in the book, The K&R C programming language, were written for 16-bit minicomputers, such as the PDP-11 mini-computers sold by Digital Equipment Coporation (DEC). As such, the modern C language is influenced by the design of the PDP-11 in its entirely, shaping the foundations for Intel’s CP/M, the MS-DOS, and finally the x86 architecture as a whole.

The specifications we are referring to is the first edition, and not the second edition that included the ANSI C standards. In this book, it introduces many standard syntaxes and data structures, of which in particular, the C standard input and output functions, specifically mentioned in Chapter 7, *Input and Output*.

In Chapter 7, Section 5, the function `sprintf` was briefly mentioned, and as follows it, is the description for this function:

> The functions `scanf` and `printf` have siblings called `sscanf` and `sprintf` which perform the corresponding conversions, but operate on a string instead of a file. The general format is `sprintf(string, control, arg1, atg2, ...)`, and `sscanf(string, control, arg1, arg2, ...)`.

Furthermore, it goes into detail what exactly it should do, in the following passage:

> `sprintf` formats the arguments in `arg1`, `arg2`, etc., according to control as before, but places the result in `string` instead of on the standard output. Of course, `string` had better be big enough to receive the result. As an example, if `name` is a character array and `n` is an integer, then `sprintf(name, "temp%d," n);` creates a string of the form "temp*nnn*" in `name`, where *nnn* is the value of `n`.

It is apparent in the passage above that it is "advised by the programmer's discretion" to provide a big enough buffer to receive the result from the `sprintf` function. And you may ask, what would happen if the programmer did not specify a big enough buffer to receive the result? The answer is that the code would result in a buffer overflow if it had been executed.

How should one write the code to create a buffer overflow? Here is an example:

```
//Executing sprintf buffer overflow
char nonTerminating[1] = { (char) 0x52 }; //0x52 is the ASCII code for the letter 'R'.
char buffer[1] = { 1 };
char buffer2[20] = "Hello world.";
sprintf(buffer, "%s," nonTerminating);
printf("%s," nonTerminating);
```

And the final output is shown on the next page, along with a screenshot of the debugger reading the memory buffer in the background. Adding `\n` to the strings do not affect the final output.

To explain what is going on, we first allocate a memory buffer of type `char` and the size of the buffer takes up 1 `char` long. This `char` variable size is determined by the specifications listed in the K&R C standards, in the following table on the next page. The table is copied from Chapter 2, *Types, Operators, and Expressions.*

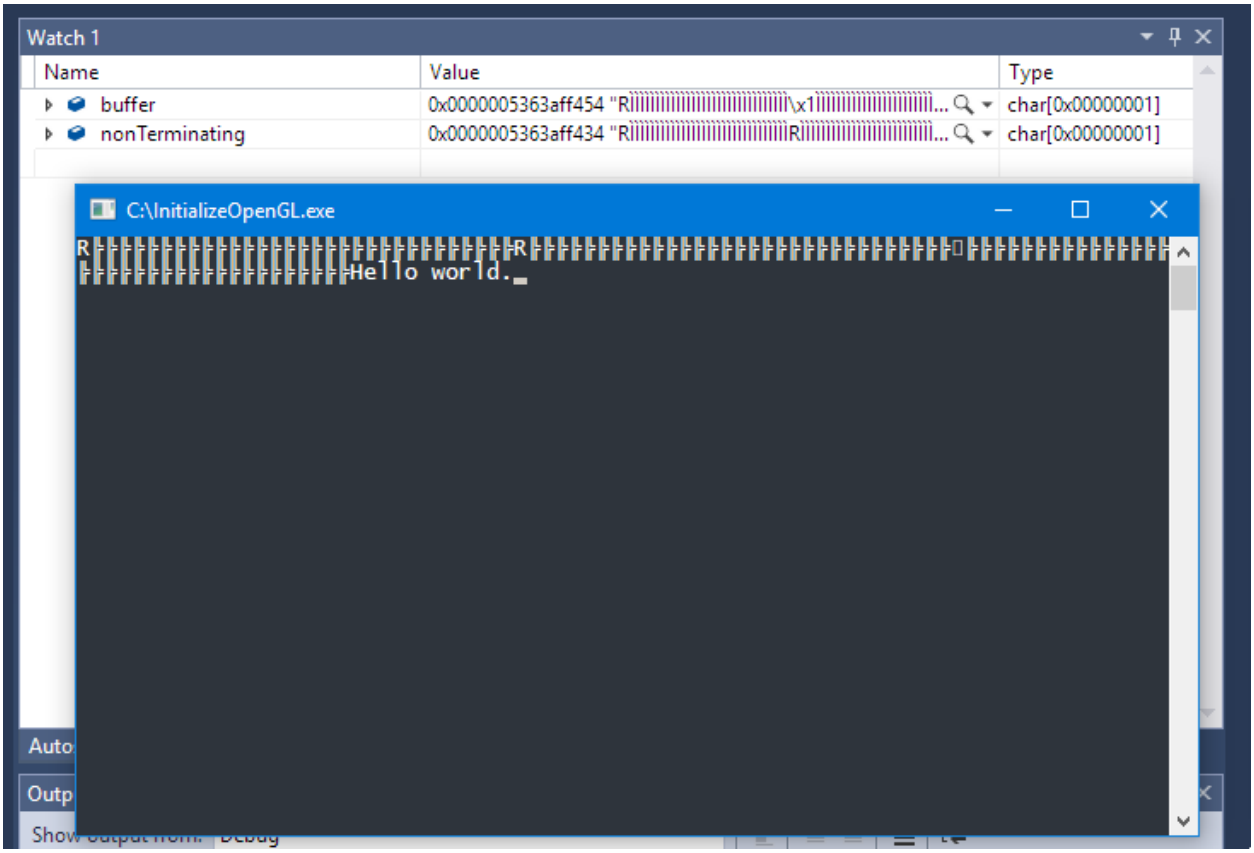

Figure 8: A simple stack buffer overflow. The debugger shows the contents of each variable, and the console output shows the contents of the memory overflow in full, reaching all the way until it hits a NULL character, or a zero. The string value, "Hello world.," is stored in the memory stack as another variable's data, and by stack buffer overflowing over, we can get a glimpse of that variable's data. Imagine what would this be if the data stored on the memory stack is sensitive information.

Table 1: The official sizes of each variable types in K&R C.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
Encoding	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16 bits	36 bits	32 bits	32 bits
short	16 bits	36 bits	16 bits	16 bits
long	32 bits	36 bits	32 bits	32 bits
float	32 bits	36 bits	32 bits	32 bits
double	64 bits	72 bits	64 bits	64 bits

This memory stack buffer overflow vulnerability existed since the introduction of K&R C and its C language specifications in 1972 when it introduces the standard input and output library, also known to all as both the standard I/O library and the `<stdio.h>` standard header. And of course, it exists even today. At first, it was considered a programming bug between the periods of 1972 to 1989, mainly because computers back then were not commonplace in typical households. Then it was elevated to a critical security vulnerability starting around 1989 and through the entire 90's, when personal computers are sold to consumers.

Because it is part of the C language standard, all C compilers and C-based programs and applications are all exposed to this vulnerability by default. Microsoft and other vendors who worked on C compilers has made it explicit to use the safer alternatives, such as `snprintf`, `sprintf_s`, etc., over `sprintf`. In Microsoft's case, they even provided to the developers who use the Win32 API library functions with built-in support for safe string operations, such as `StringCchPrintf`, `_wsprintf`, `_wsnprintf`, etc.

And here we are.

Byte Ordering (Part 1/3)

Unicode support is a feature that has made quite a few programmers and game developers some headache in the past. Those who have never dealt with localizations support in video games would jump away from putting the effort in at first. But once they dive right in, their perspectives will have changed for the better of their experiences in programming development in general.

There are three sections to get your program to have Unicode support: Memory byte order of each characters in a string, Understanding Unicode, and How Unicode characters are parsed in C/C++ code.

All three sections will help ease the high learning curve, which is necessary to prepare the readers for supporting Unicode in their games. The materials and information gathered in this section were taken from the official sources, Internet forums, and old mailing lists, but have cleaned up so it is more informative than vague message posts on those sites.

Just to put it out there, byte ordering is ***NOT ALWAYS*** determined by knowing in advance what the targeted platform's memory byte order is, and how to process the byte orders natively. For example, since this book is devoted to initializing OpenGL applications on any Windows operating system, we cannot assume Windows uses Little-Endian as the primary memory byte order, especially when Microsoft introduced Windows 10 Mobile, which is developed for the ARM 32-bit architecture. And we know for certainty the ARM 32-bit architecture supports Bi-Endian (Big/Little Endian), meaning it can support Big-Endian and Little-Endian byte orders natively.

It may be pushing it too far, but going by the assumption here, one would ask to themselves, how do we determine the byte orders on any given target platforms at runtime?

The answer is by using C-style casting, and/or C++'s type punning technique, which involves using the keyword, `reinterpret_cast<>`. Speaking of `reinterpret_cast<>`, unfortunately, as of writing this book, no one has found the official answer on when the `reinterpret_cast<>` was introduced, along with the `*_cast<>` family. One hint supports the theory that it was introduced from the early days of C++, starting with the C++ 2.0, known as the C++89, the C++ standard specifications's designated name before C++11 released in 1989. It suggests the `*_cast<>` family is formally introduced, when they started supporting new methods of type casting.

The type that we are to cast for type punning is a `uint16_t`, which is a forward-compatible unsigned short integer type. Short integers, or shorts, consist of enough bit fields for us to determine the byte order on any given modern platforms, and is the smallest variable length that we can use for this without wasting any memory space.

Below demonstrates the process for determining the byte order. Explanations will follow in order from top to bottom.

For C projects, use the following:

```
#include <stdint.h>    //Recommended for C projects

static int CStyle_IsLittleEndian() {
	uint16_t x = 0x0001;
	uint8_t* p = (uint8_t*) &x;
	return (int) *p != 0;
}
```

For C++ projects, use the following:

```
#include <cstdint>     //Recommended for C++ projects

static bool IsLittleEndian() {
	uint16_t x = 0x0001;
	uint8_t* p = reinterpret_cast<uint8_t*>(&x);
	return *p != 0;
}
```

In C, there is no keyword for Boolean values, so it must be casted to an integer type, for the function to successfully compile without errors. This is only for guaranteed execution.

In C++, the `reinterpret_cast<>` functions basically the same as a C-style casting. It is just more explicit to the developers for readability and verbosity.

Either way, the end result evaluates whether the pointer address value is 0x00 or 0x01 when it is explicitly casted from an integer value to a pointer. On Little-Endian systems, this function will return a truthy value, and vice versa for Big-Endian systems. How this works has to do with how Little-Endian and Big-Endian systems stores pointer addresses in the memory layout space.

For a memory layout on a Big-Endian and Little-Endian systems, the pointer addresses are stored in such a way that the 0x00 and 0x01 are positioned differently, as shown below.

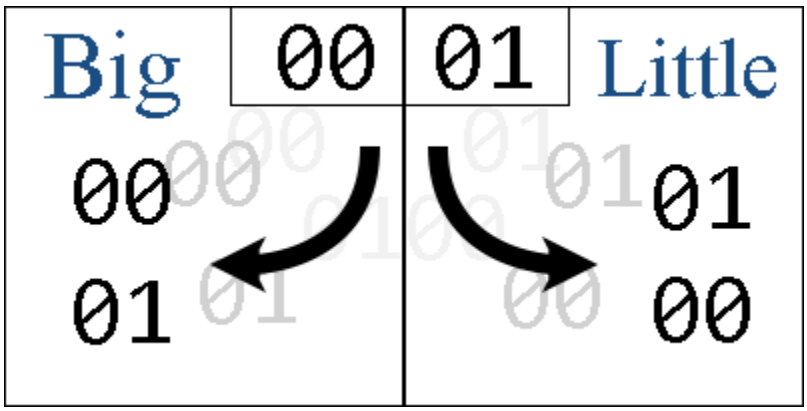

Figure 9: This image simplifies the conversion process from an integer to either Big-Endian or Little-Endian. By rotating the integer value's position to a vertical column, you can see what the memory stack will read like when reading from top to bottom. On Big-Endian systems, the integer values are rotated clockwise, while on Little-Endian systems, the values are rotated counter-clockwise.

With the values now set, we then fetch for the first low byte of the pointer address, by downcasting the pointer address from however long the size of the pointer addresses are to a `uint8_t` integer value. Downcasting the value will lose precision, and as such it will only preserve the first low byte of the original value. In other words, it is taking only the top byte value from the vertical columns in Figure 9. The top byte is the low byte in the memory stack. The closer to the bottom of the stack, the higher the memory address value gets.

Now that we have our low bytes, we then check to see if the low byte is 0x00 or 0x01. If it is 0x00, then it is a Big-Endian system. If it is a 0x01, then it is a Little-Endian system. Knowing the process of determining byte orders in place, we can use it to handle Unicode strings, and have Unicode support natively in our applications.

A Unicode Primer (Part 2/3)

This section will provide some context about Unicode, with all information gathered from the official Unicode Consortium site (Unicode Consortium, 2015). Prior to Windows 2000, UCS-2 is the default Unicode encoding. But after UTF-16 was ratified, Windows then converted from UCS-2 to use UTF-16 for backward compatibility reasons, up to today. Let us learn what Unicode is all about.

What is Unicode?

Unicode, the shortened abbreviation for “Universal Coded Character Set” (UCS), is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world’s writing systems. This standard is maintained by the Unicode Consortium, a non-profit organization that maintains and publishes the Unicode Standard, a character encoding scheme made with the intention of widespread adoption in the internationalization and localization of software applications. In this section, we are to implement said character encoding scheme for our software application for localization support.

The Unicode Standard defines the information of a Unicode character, namely the Unicode Transformation Formats (UTF), based on their encoding bit count: UTF-8, UTF-16, UTF-32, and several others. The most commonly used formats are UTF-8, UTF-32, and UCS-2, a precursor of UTF-16.

UCS-2

To briefly explain what UCS-2 is, this scheme uses a single “code value” containing one or more “code points” assigned to the “code space” between 0 and 65,535 for each character, and allows 2 bytes, or 1 16-bit word, to represent that value. Thus, the “2” in UCS-2 refers to the “2-byte encoding” scheme. For a time, this was known as “the Unicode,” and is supported by many programs to this day. This terminology is also why in the SDL companion library, SDL_ttf, there are function names with the text “UNICODE” in them, such as `TTF_RenderUNICODE_Solid`, `TTF_RenderUNICODE_Shaded`, etc. Reason is, these particular functions are built to support the so-called “Unicode,” the UCS-2 text encoding.

A “code value” is defined as one or more numbers representing a “code point.” A “code point” or “code position” is any of the numerical values that make the “code space.” Many code points represent single individual characters, but they can also have other meanings, such as for formatting. This “code space” ranging from 0 to 65,535 is known as the basic multilingual plane, or BMP for short. “Code points” can labeled with a “U” followed by the code point. As an example, U+2312 represents (ARC ⌒).

For the example of “code point” and “code space,” we will use the ASCII encoding scheme, the Extended ASCII scheme, and Unicode for comparisons. The ASCII character encoding scheme comprises of 128 “code points” in the “code space” ranging from 0x00 to 0x7F. The Extended ASCII scheme comprises of 256 “code points” in the “code space” ranging from 0x00 to 0xFF. The Unicode comprises of 1,114,112 “code points” in the “code space” ranging from 0x000000 to 0x10FFFF. The Unicode “code space” is divided into 17 planes, consisting of 1 BMP and 16 supplementary planes. Each plane is comprised of 65,536 code points, and hence we get a total of 17 × 65,536 = 1,114,112 code points.

UTF-16

UTF-16 evolved out from UCS-2 into a superset of UCS-2, mainly due to geographical and technical reasons, respectively explained on the subsequent pages.

In the East Asian region, the use of Han characters is a common feature in written Chinese (as *hanzi*), Japanese (as *kanji* and *kana*), and Korean (as *hanja*) languages. Occasionally, Vietnamese (as *hán tự* or *chữ nôm*) is included, since Vietnamese historically use Han characters in some contexts. All of these Han characters, including regional and historical variants, are collectively known as the CJK (or CJKV, with the V for Vietnamese) characters. The typefaces of modern Chinese, Japanese, Korean, and sometimes Vietnamese typically use the CJKV variants of the same given Han character, such as the one shown below.

Figure 10: The many variants of the same Han character, 次 (pronounced phonetically English as "tsɪ") using the regional commonly-used font family. Note the slight differences between the stroke lengths and angles of 二 and 冫 on the left-hand side, as well as the distance between 人 and 𠂉 (component of 刀) (Betoseha, 2015). From left to right in order, Traditional Chinese, Simplified Chinese, Korean, Vietnamese, and Japanese.

At the time of the formulation of Unicode, China, Japan, North and South Korea, and the South East Asian regions made a joint effort in creating a consistent encoding scheme to map multiple character sets into a single set of unified characters. This attempt, made to unify regional and historical variants of the Han characters together by considering each given Han character as different glyphs representing the same "grapheme" or "orthographic unit," is known as the "Han unification," or Unihan, to represent the resulting character set repertoire.

Unihan, or the Unihan Database, is maintained by the Unicode Consortium, which provides information about all the unified Han characters encoded in the Unicode Standard, including mappings to various national and industry standards, indices into standard dictionaries, encoded character variants, pronunciations in various languages, and an English Unicode definition of a given character.

One possible rationale behind the Han unification is the desire to limit the size of the full Unicode character set. If each of the CJKV characters are represented by discrete ideograms, the Unicode character set may approach or exceed 100,000 characters, even though the number of Han characters in the CJKV character set required for everyday literacy in any of the three languages are likely to be under 3,000 in total. Since Version 1 of Unicode, the UCS-2, was designed to fit a given character into 16 bits, out of the possible 65,536 code values, only 20,940 code values (or 32% of the code space) were reserved for these CJKV characters.

In the field of computer science and linguistics, there exists a dedicated subject about glyphs, graphemes, sememes, and ideograms. A glyph refers to a visual representation of a given character. A grapheme is the smallest abstract unit of meaning in a writing system, and are all recognized as the same unit of meaning by those with reading and writing knowledge of said writing system, even when it has many possible glyph expressions and/or "abstract characters." A sememe is an "abstract character"

assigned as a grapheme, representing a semantic language unit of meaning. An ideogram is a graphic symbol that represents an idea or concept, independent of any particular language.

An abstract character is a character that does not necessary correspond to what a user thinks of as a “character,” but rather it is a character by itself that can also be imposed onto other multiple Unicode abstract characters. For example, the abstract character U+0061, a, (LATIN SMALL LETTER A) combined with another abstract character U+030A, ◌̊ (COMBINING RING ABOVE) creates the combination, å. This combination might be understood by a user as a single grapheme, while it being composed of multiple Unicode abstract characters.

With the creation of Unihan, it brought forth considerable controversy, mostly in the Han characters used in Japanese, stemming from the fact that Unicode encodes Han characters as a whole, rather than as “per glyphs.,” and not considering the major and minor differences in regional Han characters.

One such major example is the Han characters for "one,” 壹, 壱, and 一, which all are used differently in Chinese, Japanese, and Korean. Many people think that the three versions of “one” should be encoded differently. One such minor example is a line stroke difference between the four-strokes glyph variant for the Han character “grass,” ⺾, and the three-strokes glyph variant, ⺿. The former is widely adopted in Traditional Chinese character glyphs, while the latter is widely adopted in Simplified Chinese, Japanese, and Korean character glyphs.

To deal with the use of different graphemes for the same Unihan sememe, Unicode added the concept of variation selectors, first introduced in Version 3.2 to allow up to 16 variant selectors, and supplemented in Version 4.0 to allow up to 256 variant selectors. The latter is being used only with CJKV Unified Ideograms to specify a variant glyph form registered in the Ideographic Variation Database.

Before Unihan was completed, Windows operating systems before Windows 2000 used to adopt the UCS-2 by default, and at the time Unicode had never had any problems with localizations prior to this. During Unihan, along with the additional 14,500 composite characters added for compatibility with legacy character sets, it was apparent the 2-byte encoding method would not be sufficient enough to map out all the characters.

Thus, UTF-16 was born. It uses a single 16-bit “code unit” to encode the most common 63,000 characters. A pair of 16-bit “code units,” also called surrogates, are used to encode over 1 million less commonly used characters.

There are two types of “code units”, single 16-bit code units that are mapped to the “code points” in the BMP, and pairs of “code units” that are encoded “code points” from the other supplementary planes. The single 16-bit code units corresponding to the code points in the BMP are numerically equal to the code points in the BMP in UCS-2, and hence is why UTF-16 is the superset of UCS-2. The pairs of code units, also known as “surrogate pairs,” are a bit more complicated, and the schemes to encode and decode the surrogate pairs is shown on the next page. The schemes do not apply to single code units.

From a technical standpoint, any one code unit will belong in one of these four ranges:

- High (leading) surrogates (0xD800 ~ 0xDBFF)
- Low (trailing) surrogates (0xDC00 ~ 0xDFFF)
- Valid low BMP characters (0x0000 ~ 0xD7FF)
- Valid high BMP characters (0xE000 ~ 0xFFFF)

Table 2: A simplified surrogate pair encoding scheme.

High-Level Step by Step Instructions	Example 1	Example 2
Give any Unicode character.	𝗞 (0x1D75F)	🍑 (0x1F351)
Subtract by 0x10000.	0xD75F	0x0F351
Divide by 0x400.	0x0035 0x035F	0x003C 0x0351
Add 0xD800 to quotient (high surrogate).	0xD835 0x035F	0xD83C 0x0351
Add 0xDC00 to the remainder (low surrogate).	0xD835 0xDF5F	0xD83C 0xDF51

Table 3: A simplified surrogate pair decoding scheme.

High-Level Step by Step Instructions	Example 1	Example 2
Give any encoded UTF-16 character.	0xD835 0xDF5F	0xD83C 0xDF51
Subtract 0xD800 from high surrogate.	0x0035 0xDF5F	0x003C 0xDF51
Multiply high surrogate by 0x400.	0xD400 0xDF5F	0xF000 0xDF51
Subtract 0xDC00 from low surrogate.	0xD400 0x035F	0xF000 0x0351
Add high and low surrogates.	0xD75F	0xF351
Add 0x10000.	0x1D75F	0x1F351

Knowing the 4 code unit ranges, we can see all of the ranges are mutually exclusive from one another as disjoint sets, therefore it is not possible for a high or low surrogate to match a valid BMP character, and is not possible for two adjacent code units to look like a legal surrogate pair. As a whole, UTF-16 is considered to be "self-synchronizing" on 16-bit words. By "self-synchronizing", given any starting code unit, a character can be determined without examining earlier code units by seeing which range the code unit falls into.

This "self-synchronization" feature is advantageous over many earlier multi-byte encoding schemes, such as Shift JIS, Big5, and TRON encoding. Without "self-synchronization," when parsing an encoded string, it must be re-parsed from the start of the string. However, this also means, if one byte is lost or the traversing process starts at a random byte, UTF-16 would no longer become "self-synchronizing."

Encoding and decoding UTF-16 surrogate pairs, even with self-synchronization, would not be complete without knowing the byte orders and endianness of the computer architecture of the targeted platforms. Since UCS-2 and UTF-16 both produce a sequence of 16-bit code units, the order of the bytes is therefore crucial. To assist in recognizing the byte order of the code units, UTF-16 allows a Byte Order Mark (BOM), a code point with the value 0xFEFF to precede the first actual UTF-16 code value.

This value is useful in two ways. First, this Unicode character is a "zero-width non-breaking space character" (ZWNBSP, or U+FEFF for short), so it will not take up any space when visually displayed to the users. Second, if the endian architecture of the decoder matches that of the encoder, the decoder should be able to detect the value 0xFEFF. Otherwise, an opposite-endian decoder will detect the Unicode reserved non-character value 0xFFFE, which is mainly used in conjunction with the BOM. With

this non-character U+FFFE being an incorrectly decoded BOM, it provides a hint to the decoder to perform byte order swapping for each of the subsequent UTF-16 code points to follow.

Note that per RFC 2781, if the BOM is missing, it should be assumed Big-Endian encoding is used. However, Windows uses Little-Endian byte order by default (mainly because the Windows operating systems are running on x86 architectures for a long time), it is assumed that many programs and applications handling Unicode strings also uses Little-Endian by default UTF-16 encoding. And this is why byte ordering is ***NOT ALWAYS*** determined by target platforms, as mentioned in the beginning of the previous section.

To address the byte order issue, the UTF-16 standard allows the byte order to be explicitly stated by specifying the endianness type as the suffix of the UTF-16 name, namely "UTF-16BE" for Big-Endian encoding type, and "UTF-16LE" for Little-Endian encoding type. When the byte order is explicitly stated, the BOM is not supposed to be prepended to the text, and the BOM character U+FEFF at the beginning of the text should be handled and displayed as a ZWNBSP character. But in practice, most applications ignore the BOM in all cases for simplicity, and this is why only some text editors and integrated development environments (IDEs) would display BOMs correctly.

UCS-4 and UTF-32

UCS-4 is the first widely known, least used, and the easiest Unicode standard out of the many commonly known standards. UCS-4 is the original ISO 10646 standard that defines a 32-bit encoding form, in which each code point in the UCS is represented by a 31-bit value between 0x0000 0000 and 0x7FFF FFFF, with the sign bit (32nd bit) left unused and is always zero. In other words, the identity mapping for UCS-4 is essentially the encoded 32-bit code point's integer value.

Table 4: Comparison between UCS code points and UCS-4.

Code Point	UCS-4
U+0000 0020	00 00 00 20
U+0000 FEFF	00 00 FE FF
U+0031 2B4E	00 31 2B 4E
U+7FFF FFFF	7F FF FF FF

However, it was later restricted by the Internet Engineering Task Force (IETF) with the aptly named "Request for Comments No. 3629" (RFC 3629), restricting the Unicode standard to match the constraints of UTF-16 encoding, explicitly prohibiting code points greater than U+10FFFF, as well as the high and low surrogates U+D800 through U+DFFF. This limited subset of the restriction from U+000000 to U+10FFFF is defined as the UTF-32 standard, referencing the use of 32-bits encoding scheme, while differing itself from UCS-4 at the time of making.

Later versions of the original ISO standard, UCS-4, removed the reserved private use of the range 0xE00000 to 0xFFFFFF and 0x6000 0000 to 0x7FFF FFFF. With the removal of the reserved ranges and the clauses in the Principles and Procedures document of ISO/ICE JTC 1/SC 2 (the IETF's Working

Group 2) listed in the [Unicode Consortium, 2017], which states that “all future assignments of Unicode code points will be constrained to the Unicode range U+000000 to U+10FFFF,” these restrictions make the UCS-4 and UTF-32 essentially identical. As a result, this effectively makes the same rules for mapping UCS-4 code points to the 32-bit code point’s integer value the same for UTF-32.

UTF-1

UTF-1 is one of the lesser known Unicode standards, designed to transform the entire UCS-4 into a stream of bytes. However, its design does not provide self-synchronization, making the operations of searching for substrings and error recovery difficult. It reuses the ASCII printing characters for multi-byte encodings, making it unsuitable for certain uses, such as POSIX-based filenames. Not only is it difficult to use, it is also slow to encode and decode due to its use of division and multiplication by a number which is not a power of 2. In the end, it did not gain much traction, but it did lay down the foundations for UTF-8.

How does it lay out the foundation? We need to look at what UTF-1 does to Unicode characters. By design, UTF-1 is a multi-byte encoding, where a single Unicode code point can be encoded in one, two, three, or five bytes. There is no UTF-1 encoded in four bytes, because that is reserved for other encoding formats. All code points in the ASCII range (0x0 ~ 0x9F) is encoded as one byte for U+0000 through U+009F. This is not the same as hex values 0x0000 through 0x009F.

UTF-1 does not use the control characters and the space character in multi-byte encodings, as the bytes from 0x00 through 0x20 and 0x7F through 0x9F are reserved to represent the corresponding code points. The control characters are marked as C0 and C1 control codes. “C0 control codes” are a set of defined codes in the range 0x00 through 0x1F, and the “C1 control codes” are in the range 0x80 through 0x9F. All of these limitations are part of the attempt allow the use of 66 protected characters to be compatible with ISO 2022.

ISO 2022 is another older standard to encode and decode a character code structure into a variable width encoding, using up to two bytes to correspond to a single character. No specific implementation has to implement all the standard, meaning the conformance level and the supported character sets are defined by the implementation. This ISO 2022 standard is where the many different character sets encoding implementation is dependent upon, with the character sets encoding standards including GB 2312, JIS 0201, JIS 0213, KS 1001, CNS 11643, and others. In short, this ISO 2022 is the precursor to the Unicode standard, ISO 10646.

UTF-1 uses the arithmetic calculation, “modulo 190.” What this means is, out of all 256 possible single-byte values that may represent a character code point, 66 of the single-byte values are protected by ISO 2022. Therefore, the character code point needs to be encoded with a value that is not one of the 66 protected character code points. Thus, 256 – 66 = 190. We can only use 190 possible single-byte values to encode a character code point. Considering the reserved byte value ranges of the C0 and C1 control codes, the remainder of the modulo operation is added with an offset of 0x21. And if the remainder of the value is greater than or equal to 0x7F, it is rounded up to the next available, non-protected byte value.

Table 5 on the next page compares the encoded UTF-1 code points with its successor, UTF-8. This table also shows how the “modulo 190” in UTF-1 is used.

Based on the standard specifications for UTF-1 [ISO, 1993], the outline of the UTF-1 encoding algorithm is as follows:

- A Unicode character from code points U+0000 0000 to U+0000 009F is mapped to the corresponding octet from 00 to 9F.
- A Unicode character from code points U+0000 00A0 to U+0000 00FF is mapped to a sequence of two octets, with the first octet being A0, and the second octet in the range A0 to FF.
- A Unicode character from code points U+0000 0100 to U+0000 4015 is mapped to a sequence of two octets, with the first octet in the range from A1 to F5, and the second octet having 190 values in the range 21 to 7E or the range A0 to FF.
- A Unicode character from code points U+0000 4016 to U+0003 8E2D is mapped to a sequence of three octets, with the first octet in the range from F6 to FB, and the other octets in the range 21 to 7E or the range A0 to FF.
- A Unicode character at code points U+0003 0E2E or larger is mapped to a sequence of five octets, with the first octet in the range from FC to FF, and the other octets in the range 21 to 7E or the range A0 to FF.
- Four-octet sequences are not used, since it would maximize (or use up all the values too soon) the number of characters that can use the three-octet form.

In terms of applied high-level C pseudo-code, this is how a Unicode character is determined:

1. All numbers are in hexadecimal format. “T” is an octet function. “U” is the inverse of “T.” “Z” is a byte value. “%” is a modulo operation. “;” is an octet delimiter/separator.
2. T(Z) is a Unicode character defined for Z = 00 ~ FF, such that:

Z = 00 ~ 5D	T(Z) = Z + 21
Z = 5E ~ BD	T(Z) = Z + 42
Z = BE ~ DE	T(Z) = Z – BE
Z = DF ~ FF	T(Z) = Z – 60

3. U(Z) is the inverse of T(Z). That means, U(T(Z)) = Z, and T(U(Z)) = Z.

Z = 00 ~ 20	U(Z) = Z + BE
Z = 21 ~ 7E	U(Z) = Z – 21
Z = 7F ~ 9F	U(Z) = Z + 60
Z = A0 ~ FF	U(Z) = Z - 42

Table 5: Encoding comparison between UTF-8 and UTF-1.

Code point	Integer (Decimal)	UTF-8	UTF-1
U+0000	0	00	00
U+0020	32	20	20
U+0021	33	21	21
U+007F	127	7F	7F
U+0080	128	C2 80	80
U+009F	159	C2 9F	9F
U+00A0	160	C2 A0	A0 A0
U+00BF	191	C2 BF	A0 BF
U+00C0	192	C3 80	A0 C0
U+00FF	255	C3 BF	A0 FF
U+0100	256	C4 80	A1 21
U+015D	249	C5 9D	A1 7E
U+015E	350	C5 9E	A1 A0
U+01BD	445	C6 BD	A1 FF
U+01BE	446	C6 BE	A2 21
U+07FF	2047	DF BF	AA 72
U+0800	2048	E0 A0 80	AA 73
U+0FFF	4095	E0 BF BF	B5 48
U+1000	4096	E1 80 80	B5 49
U+4015	16,405	E4 80 95	F5 FF
U+4016	16,406	E4 80 96	F6 21 21
U+D7FF	55,295	ED 9F BF	F7 2F C3
U+E000	57,344	EE 80 80	F7 3A 79
U+F8FF	63,743	EF A3 BF	F7 5C 3C
U+FDD0	64,976	EF B7 90	F7 62 BA
U+FDEF	65,007	EF B7 AF	F7 62 D9
U+FEFF	65,279	EF BB BF	F7 64 4C
U+FFFD	65,533	EF BF BD	F7 65 AD
U+FFFE	65,534	EF BF BE	F7 65 AE
U+FFFF	65,535	EF BF BF	F7 65 AF
U+10000	65,536	F0 90 80 80	F7 65 B0
U+38E2D	233,005	F0 B8 B8 AD	FB FF FF
U+38E2E	233,006	F0 B8 B8 AE	FC 21 21 21 21
U+FFFFF	1,048,575	F3 BF BF BF	FC 21 37 B2 7A
U+100000	1,048,576	F4 80 80 80	FC 21 37 B2 7B
U+10FFFF	1,114,111	F4 8F BF BF	FC 21 39 6E 6C
U+7FFFFFFF	2,147,483,647	FD BF BF BF BF BF	FD BD 2B B9 40

The UTF-1 standard suggested to use a table lookup or other methods to offset its inherent inefficiency. Here, we use a table lookup for converting from UCS to UTF-1, and vice versa.

Table 6: From UCS to UTF-1 format.

Condition = Code Points Range	**Mapped to UTF-1 octets**
X = 0000 0000 ~ 0000 009F	X;
X = 0000 00A0 ~ 0000 00FF	A0; X
X = 0000 0100 ~ 0000 4015 Y = X – 0000 0100	A1 + (Y / BE); T(Y % BE);
X = 0000 4016 ~ 0003 8E2D Y = X – 0000 4016	F6 + (Y / BE^2); T((Y / BE) % BE); T(Y % BE);
X = 0003 8E2E ~ 7FFF FFFF Y = X – 0003 8E2E	FC + (Y / BE^4); T((Y / BE^3) % BE); T((Y / BE^2) % BE); T((Y / BE) % BE); T(Y % BE);

Table 7: From UTF-1 format to UCS.

Condition = UTF-1 octets	**Mapped to Unicode Code Points**
X = 00 ~ 9F;	X
X = A0; Y;	Y
X = A1 ~ F5; Y;	(X – A1) × BE + U(Y) + 0000 0100
X = F6 ~ FB; Y; Z;	(X – F6) × BE^2 + U(Y) × BE + U(Z) + 0000 4016
X = FC ~ FF; Y; Z; V; W;	(X – FC) × BE^4 + U(Y) × BE^3 + U(Z) × BE^2 + U(V) × BE + U(W) + 0003 8E2E

UTF-8

As the UTF-1 was drafted, people realized the encoding format is not adequate enough due to performance reasons and other problems, with the biggest problem was not having a clear separation between ASCII and non-ASCII characters, making UTF-1 non-backwards compatible. New UTF-1 tools would be backwards compatible with the ASCII-encoded text, but UTF-1 encoded text could confuse existing tools expecting ASCII or extended ASCII. This is because UTF-1 encoded text may include continuation bytes in the range 0x21 ~ 0x7E, meaning for something else in ASCII. For instance, the Unix path directory separator, '/' is 0x2F in ASCII, but it may be a continuation byte for a Unicode character, for code points U+0100 and up.

Back to the drawing board, a new standard [IETF, 2003] was drafted to replace UTF-1. This new standard would inherit the UTF-1 encoding sequences. The standard also included other important features in UTF-8, especially with backwards compatibility with ASCII- and extended ASCII-encoding text being the main driving force behind the design.

They also included a fallback and auto-detection feature to be backwards compatible with 7-bit ASCII. This feature allows a UTF-8 processor, if erroneously received an extended ASCII text to process, it can "fall back" or replace the 8-bit bytes to the appropriate code point in the Unicode Latin-1 Supplement Block, which ranges from U+0080 to U+00FF. In short, real-world extended ASCII character sequences which look like valid UTF-8 multi-byte sequences are unlikely to ever occur, and as a result, the fallback errors will be false negatives and rare.

Self-synchronization is another feature in UTF-8, where the leading bytes and the continuation bytes, "prefix code," do not share values. The prefix code starts with `10` while a single byte starts with `0`, and longer leading bytes start with `11`. This helps with searching in UTF-8 processors, in that it will not accidentally find the sequence for one character starting in the middle of another character.

As with the same restrictions of Unicode being applied to match with UTF-16, UTF-8 is limited to U+000000 to U+10FFFF, but like UTF-1, it is capable of encoding up to U+7FFF FFFF. One aspect of matching UTF-8 to the UTF-16 standard restrictions is that the full "code points" range would require omitting 2,048 technically invalid "surrogate code points." This means, in practice, UTF-8 would have $17 \times 2^{16} - 2{,}048 = 1{,}114{,}064$ "valid code points." The "surrogate code points," ranging in the 0xD800 ~ 0xDFFF, are reserved exclusively for use with UTF-16. If a UTF-8 encoded value is in the range of the "high surrogate" 0xD800 ~ 0xDBFF or is in the "low surrogate" 0xDC00 ~ 0xDFFF, then it is considered to be an "unpaired surrogate," which is invalid in UTFs.

All of these features and its theoretical capacity to encode even more than the maximum Unicode character code point means UTF-8 is a popular choice when it comes to supporting non-ASCII characters. In fact, nowadays, it is the de-facto standard to implement as it is used by 92.5% worldwide [W3Techs, 2018]. It is now highly encouraged to use UTF-8 as the standard Unicode encoding, especially when C and C++ compilers can natively express Unicode characters in UTF-8 using the syntax, `u8"…"`, and the adoption of the encoding allows ease of internationalization and localization for software and web standards.

The resulting encoding for UTF-8 is similar to UTF-1, as seen in Table 5 from before, but it is more streamlined and more efficient than UTF-1 due to multiplication and division all uses a power of 2.

We will now go over comparisons of using UTF-8 and using other UTFs on the next page.

Table 8: Different octet types of UTF-8 (1993). The letter "x" indicates bits available for encoding bits of the Unicode code point.

Number Range (Hexadecimal)	UTF-8 Octet Sequence (Binary)
0000 0000 ~ 0000 007F	0xxxxxxx
0000 0080 ~ 0000 07FF	110xxxxx 10xxxxxx
0000 0800 ~ 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 ~ 001F FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
0020 0000 ~ 03FF FFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
0400 0000 ~ 7FFF FFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

The above table shows how UTF-8 is encoded based on the Unicode code points range on the left into the octet sequence on the right. The number range also applies to the UTF-16 limit of 0x10FFFF, thus the octet sequences can be applied universally.

When compared with single-byte encodings or encodings based on a “code page,” UTF-8 can encode any Unicode characters, therefore it avoids the need to figure and what “code page” or indicate what character set is in use. This allows outputs in multiple “scripts” at the same time. A “code page” is a type of character encoding with a set of printable characters and control characters with unique numbers. A “script” is a collection of letters and other written signs used to represent textual information in one or more writing systems. In other words, when using UTF-8 encoding, one can write both German scripts and Chinese scripts altogether without worrying both scripts will mix together.

When compared with multi-byte encodings, since UTF-8 can encode any Unicode characters, binary files and text files written or parsed in different language scripts can be displayed correctly without having to choose the correct code page or font. Given that the UTF-8 features self-synchronization, character boundaries are easy to find by using the above table to match the bit patterns in either forward or reversed directions. If any bit pattern is corrupted, a UTF-8 parser can always locate the next UTF-8 character and resume processing. This also means any byte-oriented string searching algorithms can be used with UTF-8 data, since the sequences of bytes for a Unicode character are unique and mutually exclusive to any other bytes for another Unicode character.

However, in both single-byte encodings and multi-byte encodings, UTF-8 will take more space than any other encodings, if the single-byte and multi-byte encodings are designed for a specific language script. For example, East Asian legacy encodings generally uses two bytes to encode characters, but it would take three bytes if encoded in UTF-8. This by itself has caused some controversy in some Asian countries, including India, Thailand, and Nepal, where their language scripts have a much larger Unicode code point such that it takes as much as two to three times the size of normal single-byte encoded scripts when encoded in UTF-8.

And finally, when comparing with UTF-16, there are some advantages and disadvantages with one another. One example is the UTF-16 encodings and UTF-8 encodings, both of which are represented by byte arrays in programs. UTF-16 encodings would rarely, if not would not need to, require doing extra work when converting to UTF-8. Since UTF-16 is represented by 16-bit word arrays, so converting to UTF-16 while maintaining compatibility with existing ASCII-based programs requires every API and data structure that takes a string to be duplicated, meaning one version accepting byte strings and another version accepting UTF-16. This is how Windows API is designed when handling UTF-16 and ASCII-based string texts.

The advantage over UTF-16 is text encoded in UTF-8 will be smaller than the same text encoded in UTF-16, if there are more code points below U+0080 than in the range U+0800 ~ U+FFFF. This is true for all European-based language scripts. If the encoded text is in Chinese, Japanese, or any Brahmic scripts, they will take up more space in UTF-8 if there are more of these Unicode characters than there are ASCII characters. This may not be true if the text is written in proses, poems, verses, or rhymes, however.

Technological advances in communications and networking are pushing towards various ways to send across bitstreams from one point to another. UTF-8 encoding is more suited for sending streams of bytes across the network. Reason why UTF-8 is so heavily pushed than other UTFs, is the fact UTF-8 is better than UTF-16. Since a UTF-16 string must use a pair of bytes for each code unit, the byte order of those two bytes would become an important issue to tackle when it comes to streaming data. If just one odd number of bytes is missing from UTF-16, then the entire rest of the UTF-16 string would be totally incomprehensible. On the other hand, the self-synchronizing UTF-8 do not have this issue, by dropping the corrupted character, and then recover back from corruption starting with the next character sequence that comes after.

UTF-8 encoding and decoding works like so [IETF, 2003]:

1. Determine the number of octets required from the Unicode code point and the first column of the table in Table 8. There is only one valid way to encode a given code point.
2. Prepare the high-order bits of the octets as per the second column of the table in Table 8.
3. Fill in the bits marked with the letter, "x," from the bits of the Unicode code point, expressed in binary. Start by putting the lowest-order bit of the Unicode code point in the lowest-order position of the last octet of the sequence, then put the next higher-order bit of the Unicode code point in the next higher-order position of that octet. Repeat until the x bits of the last octet are filled in, then move on to the next to last octet, and repeat again until every single "x" is filled in.

Here is a demonstration for the Unicode character, ᛟ, a Runic letter Othalan Ethel O.

Table 9: UTF-8 encoding step-by-step instructions.

Step-by-Step Instructions	What Happens Next
1. Determine Unicode code point.	U+16DF
2. Convert number to binary form.	0001 0110 1101 1111
3. Sort by byte order (high → low)	0001 0110 1101 1111
4. Get octet bits from lookup table.	1110xxxx 10xxxxxx 10xxxxxx
5. Fill in the "x" by byte order.	11100001 10011011 10011111

The end result of the UTF-8 encoded sequence for ᛟ is 11100001 10011011 10011111. We have to do step 3 so we can be sure the binary form is not written because it is parsed backwards and such, but it can be optional for others who are more experienced.

Let us do a bit of a twist of this. What if we want to convert a UTF-16 encoded character to a UTF-8 character? Well, it is not straightforward and intuitive like 16 being a multiple of 8. A passage in

RFC 3629 recommended the best way to encode a UTF-16 character to a UTF-8 character is to first decode the UTF-16 to get the Unicode code point, and then you start from there and encode it into UTF-8.

In short, using the same Runic character, ᛟ, we need to do Table 3 and then do Table 9.

To do the opposite and decode a UTF-8 character back into a Unicode character:

1. Initialize a binary number with all bits set to 0. Up to 21 bits may be needed.
2. Determine which bits encode the character number from the number of octets in the sequence, and do a quick lookup to get the number of bits marked with "x."
3. Distribute the bits from the sequence to the binary number, first by aligning the lower-order bits from the last octet of the sequence and proceed to the left until no "x" bits are left.
4. The binary number is now equal to the Unicode code point.

And here is a demonstration of the decoding steps for UTF-8 to decode the Runic character, ᛟ:

Table 10: UTF-8 decoding step-by-step instructions.

Step-by-Step Instructions	What Happens Next?
1. Get the UTF-8 character in binary form.	1110 0001 1001 1011 1001 1111
2. Initialize a binary number of same bit count.	0000 0000 0000 0000 0000 0000
3. Determine bit octet sequence from lookup table.	1110 xxxx 10xx xxxx 10xx xxxx
4. Filter out bits marked in "x" from binary form.	0001 0110 1101 1111
5. Convert binary form to hexadecimal form.	16DF
6. Obtain Unicode code point.	U+16DF

What is next?

You now have a comprehensive idea of what Unicode is all about, what formats are available, the history of Unicode, and how to encode and decode Unicode characters into UTFs. It is time for us to move on to the next section and the final part, where we shall discuss how to implement the encoding and decoding algorithms in a programmatic way using C programming language.

Give yourself a pat for reading this section, at the very least. You deserve an effort for this!

Applying Unicode (Part 3/3)

Welcome to this final section where we discuss about how to implement Unicode support into your applications in C programming language. By the end of this section, you will understand the meaning behind this phrase is outdated:

> Text processing in C++ is *orders of magnitude* less painful than it is in C.

What does this phrase mean prior to this? It has to do with new primitive data types introduced in both C11 and C++11, called `char16_t` and `char32_t`, whom both of these are used with Unicode strings and wide strings. These new primitive data types are paired with the Unicode library, `<uchar.h>`. Back before C11 and C++11 introduced these primitive data types, Unicode has not been set in stone in the language specification standards yet, and it was pretty difficult to handle text processing in general. Right now, the current state of text processing in C and C++ seems to be easier, but it is not easily to determine if it is a drastic quality-of-life change as a result.

Currently, there is no apparent answer as to which Unicode support is better, C11 or C++11. The only thing worth mentioning here is the "interoperability" of both C and C++ languages in your application code, where you might be mixing C code and C++ code together without even realizing this. It can be a headache if you are strictly adhering to either C or C++, so this section may be important to you, if you are having trouble with this.

This section is broken up into three subchapters. The first subchapter will break down the code in a line by line matter, or by a series of lines of codes, with added explanations and comments on why this is done. Following up is the second subchapter, which will let you see how the full code is implemented without any explanations nor comments in between the lines. That way, you can refer to the second group for a quick scan of how the code is written, and then come back to the first group and re-read passages in case you forget. The final subchapter teaches how to use the latest C11 standard to work with Unicode.

Making Common Functions

We start off with a hypothetical UTF-16 character in Little-Endian byte order, and we want to convert the character code point to UTF-8. Little-Endian is chosen as the primary choice, so it has an easier difficulty curve when it comes to reading the output.

The first thing we want to do is to convert all UTF-16 surrogate pairs to be in Little-Endian byte order. We need to create a function we can use to determine the byte order of the UTF-16 character number.

```c
#include <stdint.h>    //Recommended for C projects

static int IsLittleEndian() {
	uint16_t x = 0x0001;
	uint8_t* p = (uint8_t*) &x;
	return (int) *p != 0;
}
```

This function will help us determine the byte order at runtime. Refer to Part 1 of the 3-part series for more information about this function.

Another crucial step is to extract bits from any given number. "Bitwise extraction" is an operation where we only need to take out certain bits out from the number, using bitwise manipulation. For example, we have a given number 0xF, which in binary form is 1111. We want to extract only the 2nd and 3rd bits from that number.

To extract the 2nd and 3rd bits, we require the use of "bit masking," as shown below:

```c
unsigned int extract(unsigned int value) {
	return value & 0x6;
}

extract(0xF);
```

Can we make this flexible? Here is a quick rundown on how to create an algorithm out of this.

Since we only wanted the 2nd and 3rd bits, we can say we wanted a range of bits from the 2nd bit to the 3rd bit, and we wanted to include the 2nd and 3rd bits together. Therefore, by using only known numbers (2 and 3), the size of this range is 3 – 2 + 1 = 2.

Now, the known numbers have increased (2 and 3 are both indices, and size is 2), we can try thinking about breaking down the number, 0x6. 0x6 in binary form is 0110_b, with the subscript, $_b$, denoting the number is in binary form. We can see the 1's bits are in the 2nd and 3rd position. We can try to create a way to get from 2 and 3 into 0110 using bitwise operations.

Our main equation to getting the binary number of the 2nd and 3rd bits is this:

- $(1111_b \ \& \ 0110_b) >> 1 = 0011_b$

We first need to consider the many permutations of methods to get to 0110, which would look like one of these:

- $0110_b = 0111_b - 1$
- $0110_b = 011_b << 1$
- $0110_b = 1000_b - 2$

And so forth. Maybe the last method is a bit complex, because we do not have a second 2 in our list of known numbers. We will put that aside for now. Applying the substitution property to all of these remaining methods, we get these:

- $0110_b = (1000_b - 1) – 1$
- $0110_b = (100_b - 1) << 1$

Hmm, the first method looks to be a bit nasty, because we could not make 1000_b out of 2 and 3 and the size 2. We will put that aside for now. Let us try applying our known numbers to the remaining equation, so that when we are doing bit-shifting operations, we only use our known numbers. It will be like so:

- $0110_b = ((1 << (3 – 2 + 1)) – 1) << (3 – 2)$

Again, using substitute property, we will try solving the original problem by plugging in 1111_b into the method, and then work our way backwards to the above equation so the known numbers are used, like so:

- $(1111_b \& 0110_b) >> (3 – 2) = 0011_b$
- $(1111_b >> (3 – 2)) \& (0110_b >> (3 – 2)) = 0011_b$
- $(1111_b >> (3 – 2)) \& (0011_b) = 0011_b$
- $(1111_b >> (3 – 2)) \& (0100_b – 1) = 0011_b$
- $(1111_b >> (3 – 2)) \& ((1 << (3 – 2 + 1)) – 1) = 0011_b$

We now have two equations:

- $0110_b = ((1 << (3 – 2 + 1)) – 1) << (3 – 2)$
- $(1111_b >> (3 – 2)) \& ((1 << (3 – 2 + 1)) – 1) = 0011_b$

Let us change the first method, so we can plug both equation in together as one big equation:

- $0011_b = ((1 << (3 – 2 + 1)) – 1)$
- $(1111_b >> (3 – 2)) \& ((1 << (3 – 2 + 1)) – 1) = 0011_b$

Wow! We now have something that is arithmetically coherent out from all the mess we made. Can we create a C function with known variables (arguments or parameters) so the two equations are coherent? Assuming known numbers are “a,” “b,” “c,” etc., and we use new known numbers when we need do, we plug the known numbers in, and substitute the known numbers as C function arguments, like so:

- $0011_b = ((1 << (B – A + 1)) – 1)$
- $(C >> (B - A)) \& ((1 << (B – A + 1)) – 1) = 0011_b$, where A = 2nd bit, B = 3rd bit, and $C = 1111_b$.

We can convert the equations into C pseudo-code, like so:

- Mask = (1 << (B – A + 1)) – 1;
- (C >> (B – A)) & Mask = R, where A = 2nd bit, B = 3rd bit, $C = 1111_b$, and R is our return value.

With 0011_b being the final return value for the entire algorithm, we can start translating the pseudo-code into a pure C programming language, and finally get the following code:

```
unsigned int extract(unsigned int value, int begin, int end) {
	unsigned int mask = (1 << (end - begin + 1)) - 1;
	return (value >> (end - begin)) & mask;
}
```

When using the `extract()` function, we want to put in a Unicode code point, then specify grabbing a range of bits from the `begin` position to the `end` position (e.g., 2nd bit position to 3rd bit position).

We now have all the necessary common functions to aid us in encoding and decoding UTFs. Hopefully, the bitwise mathematics is helpful to you.

Converting UTF-16 to UTF-8

The concept of Unicode code points is analogous to a global currency unit on the trade market, such as the U.S. Dollar. With how the U.S. Dollar currency unit being used all over the world and is well known, it is easy to convert any lesser known currency to U.S. Dollar, and back. In this case, the lesser known currency is our concept of UTF-16, and the U.S. Dollar is our concept of Unicode code points. And we are here to convert UTF-16, a lesser known currency, into U.S. Dollars, a well-known global currency.

As mentioned in an earlier Appendix chapter, UTF-16 consists of a pair of high surrogates and low surrogates, altogether known as the surrogate pairs. What we need to do is convert the surrogate pair of any given UTF-16 character into the corresponding Unicode code point. We will use the tables, Table 2 and Table 3, to help us create the algorithms needed to encode code points into UTF-16, and then we do the opposite for decoding UTF-16 back into UTF-8 code points.

This section will only discuss about encoding. The next section will follow up with the decoding. Both sections will use different Unicode code points, so the readers can have more perspective on what is going on.

Taking the step-by-step instructions from Table 2 for UTF-16 code points above U+FFFF:

1. Give any Unicode character.
2. Subtract by 0x10000.
3. Divide by 0x400.
4. Add 0xD800 to quotient (high surrogate).
5. Add 0xDC00 to the remainder (low surrogate).

Let us create an algorithm based on those steps. In pseudo-codes, this is the result:

- Set Unicode code point as CP
- CP - 0x10000 = CP
- CP / 0x400 = Q and R, where Q is the quotient, and R is the remainder.
- Q + 0xD800 = Q
- R + 0xDC00 = R
- Q and R together is a surrogate pair.

But before we continue, we need to think about how we are going to store the Q and R together as a group. There are various ways to go about this, but in the end, the Q and R both need to be read and parsed together by a UTF-16 parser, so it can be translated into a valid Unicode character. In other words, both Q and R need to be combined together in the actual C code.

And when it comes to C code string manipulations, it gets really complex. Especially when considering that this book is teaching the readers how to write boilerplate C code, so the readers can create their own game engine foundations and software APIs. Jumping back to the topic…

To make the entire process easier to understand, we will consider an array that will hold UTF-16 characters to be the input, and as the output. An array of UTF-16 characters is called a UTF-16 string. Entirely, there is only one array buffer of storage, and we will be converting the UTF-16 to Unicode code points on the fly. Why are we doing this? If you recalled from our Unicode Primer chapter, UTF-8, UTF-16, and UTF-32 are all limited to 0x000000 ~ 0x10FFFF code points, and are all restricted to fit all UTF-16 valid code points. This means, the bytes that holds one UTF-16 character can fit either one UTF-8 character or one UTF-32 character. By using only one array buffer, we can maximize the efficiency of our memory usage, but at the cost of making our code a bit more complex.

After all, we are "sadistic" in our approach in this book, so why not go all out instead?

The first thing we need to in our actual C code, is to define a primitive data type that can hold one UTF-16 code unit. One UTF-16 code unit is 16 bits long, or 2 bytes. Normally, we use `unsigned short` as our primitive data type to hold. But we cannot assume that `unsigned short` is guaranteed to always be 16-bits long, no matter what, on different Windows environments. We have to use `char16_t`.

And no, we are not going to use `wchar_t` because it has been defined to be 32-bits long on Unix/POSIX operating systems (MacOS and Linux), and it is 16-bits long on Windows operating systems, making `wchar_t` a primitive data type that is defined by implementation. It is therefore not suitable for our purposes, where we would need a fixed size primitive data type to accurately handle and store UTF-32 code points, UTF-16 code units and UTF-8 code points.

But alas, we need to start somewhere. In C, what we need to do for the sake of learning how to convert UTFs, is by doing "unit testing," so we have the right data of UTF-16 strings for practicing. We can use the Windows SDK provided methods to achieve this, particularly the Windows C Header function, `WideCharToMultiByte()` and `MultiByteToWideChar()`. These functions will convert any given wide string to a multi-byte string, and vice versa. It can also provide us the actual length of the given input of a string or wide string data, by not giving any input data as the parameter `cbMultiByte` to these functions.

A useful resource that talks about `char`, `wchar_t`, `char16_t`, and `char32_t` (Microsoft, 2018) can be found on Microsoft Docs. It explains how to use these variable types and integrate them with C strings. And as such, we will be only covering `char`, `char16_t`, and `char32_t`, as these types are the more preferred, cross-platform compatibility-safe alternatives to the old `wchar_t`.

> The `char16_t` and `char32_t` types represent 16-bit and 32-bit wide characters, respectively. Unicode encoded as UTF-16 can be stored in the `char16_t` type, and Unicode encoded as UTF-32 can be stored in the `char32_t` type. Strings of these types and `wchar_t` are all referred to as wide strings, though the term often refers specifically to strings of `wchar_t` type.

Here is a demonstration in C and C++ codes for WideCharToMultiByte(). It demonstrates how to encode and decode out a UTF-16 string into a valid UTF-8 string, shown on the next page.

```c
//Standard headers required in C11. Program is running in Visual Studio 2017 v15.9.6.
#include <Windows.h>
#include <string.h>
#include <wchar.h>
#include <stdio.h>

//This code is running on 64-bit Windows. This means, we are going to use HeapAlloc,
//which is the default 64-bit Windows 10 OS method of allocating memory in C. There
//are other heap memory allocating functions, GlobalAlloc and LocalAlloc, but they
//are just function wrappers of HeapAlloc since 32-bit Windows XP. Therefore, it is
//faster to call on HeapAlloc function directly instead of the other ones.
//(Per MSDN Docs, in the "Comparing Memory Allocation Methods" article.)

//Pure C code is pretty brutal in terms of passing data in and out. So, to simplify
//the whole process, we use struct objects to hold our data.
typedef struct UTF8_String {
	LPSTR data;
	int length;
} UTF8_String;

typedef struct WideString {
	LPWSTR data;
	int length;
} WideString;

void utf8_encode(WideString* in, UTF8_String* out) {
	//Empty strings check.
	if (!in->data || in->length <= 0) {
		out->data = NULL;
		out->length = 0;
		return;
	}

	//Retrieves the number of bytes required, including the NUL terminating character,
	//for the creation of the UTF-8 string.
	out->length = WideCharToMultiByte(CP_UTF8, 0, in->data, in->length,
                                                                                NULL,
0, NULL, NULL);
	WideCharToMultiByte(CP_UTF8, 0, in->data, in->length,
                                                 out->data, out->length, NULL, NULL);
	return;
}

void utf8_decode(UTF8_String* in, WideString* out) {
	//Empty strings check
	if (!in->data || in->length <= 0) {
		out->data = NULL;
		out->length = 0;
		return;
	}

	//Retrieves the number of bytes required, including the NUL terminating character.
	out->length = MultiByteToWideChar(CP_UTF8, 0, in->data, in->length, NULL, 0);
	MultiByteToWideChar(CP_UTF8, 0, in->data, in->length, out->data, out->length);
	return;
}
```

xxx

```
int main() {
	WideString wstr;
	//This wide string consisting of Chinese characters will have differing lengths of
	//byte encoding. It will be 12 bytes long for CP950 (Big5 encoding), but it will
	//be 18 bytes long for UTF-8. Wide string in text form:  門阜陀阿阻附
	wstr.data = TEXT("\u9580\u961c\u9640\u963f\u963b\u9644");

	//Gets the length of the UTF-16 wide string, a total of 6 characters.
	wstr.length = wcslen(wstr.data);

	//Using wprintf(), not printf(). Using %ls for wide strings, not %s for strings.
	wprintf(TEXT("Original text is: %ls\n"), wstr.data);
	printf("Original length is: %d\n", wstr.length);

	//We need Windows OS to allocate some memory. The easiest way is to allocate from
	//a private heap called the default process heap.
	//All processes will have at least 1 default process heap.
	HANDLE heapHandle = GetProcessHeap();
	if (heapHandle == NULL)
		//We should check for errors if "heapHandle" is NULL (per Microsoft Docs).
		//For now, it will return -1.
		return -1;

	UTF8_String str;
	//Assuming each UTF-8 character takes up 4 bytes each, which is holding the
	//largest UTF-8 encoded Unicode code point character, 0x10FFFF. We could also
	//choose to preserve memory, at the cost of performance, by calling on
	//"utf8_encode" twice, with the first call to get the exact UTF-8 string
	//length, and the second call to pass in the data with exact heap size.
	str.data = (LPSTR) HeapAlloc(heapHandle, HEAP_ZERO_MEMORY, wstr.length * 4);
	utf8_encode(&wstr, &str);

	//This will print out garbage data, because it is now UTF-8 encoded.
	printf("Result is: %s\n", str.data);
	printf("Size of result is: %d\n", str.length);

	//Time to decode this garbage.
	WideString newWStr;
	//Using previously known length of the wide string.
	newWStr.data = (LPWSTR) HeapAlloc(heapHandle, HEAP_ZERO_MEMORY, wstr.length * 2);
	utf8_decode(&str, &newWStr);

	//Using wprintf(), not printf(). Using %ls for wide strings, not %s for strings.
	wprintf(TEXT("Result is: %ls\n"), newWStr.data);
	printf("Size of result is: %d\n", newWStr.length);

	//Must free up the heap whenever the heap is no longer required.
	HeapFree(heapHandle, 0, str.data);
	HeapFree(heapHandle, 0, newWStr.data);

	return 0;
}
```

Something to take note of, is that the output of the wide strings will not display correctly in the Command Prompt console window. This is due to the fact that certain code pages are only avaliable for managed applications[13], and not Visual Studio console window applications. The only way to display the characters is by outputting the string encoded in UTF-8, then setting the console window code page to CP65001, which is the identifier for UTF-8 on Windows.

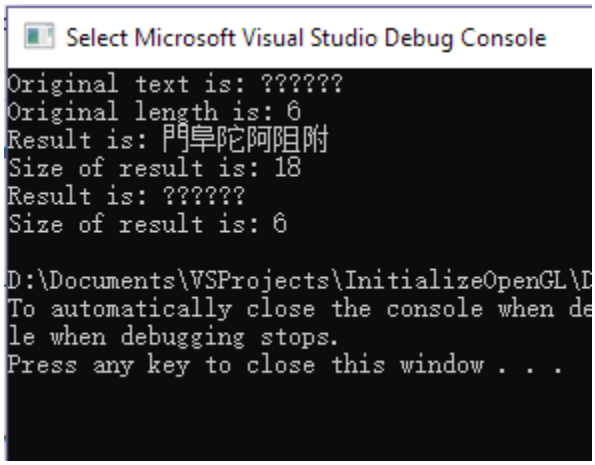

Figure 11: When changing the console font from Consolas (default) to NSimSun, it will then be able to display UTF-8 encoded characters. This can only happen after the code page has been set to 65001.

You can change the code page of the console application window by calling on the Windows function, `SetConsoleOutputCP()`.

```
#include <Windows.h>

int main() {
	//CP_UTF8 is Code Page 65001, UTF-8.
	SetConsoleOutputCP(CP_UTF8);
	return 0;
}
```

If you wished to inspect the code just to be certain the code is working correctly, you can use the Visual Studio 2017 debugger and inspect the variables in the Autos, Locals, and Watch windows. See Figure 12 on the next page, to see the values stored in the variables from the main code on the previous pages.

[13] A managed application is an application where the system administrator exerts some level of control over the installation and maintenance of the product. Code pages for UTF-16 (CP1200 and CP1201) and UTF-32 (CP12000 and CP12001) are all only available for managed applications. (Microsoft, 2018)

As one would expect, the wide strings, `wstr` and `newWStr`, both store the UTF-16 string values. It is displayed correctly in Visual Studio 2017, mostly because the debugger and Windows in general understand the UTF-16 format, and can display the UTF-16 strings natively.

But the inspection of the variable `str`, which holds the UTF-8 string value, may look garbled because it is individually displaying each byte as type `char`.

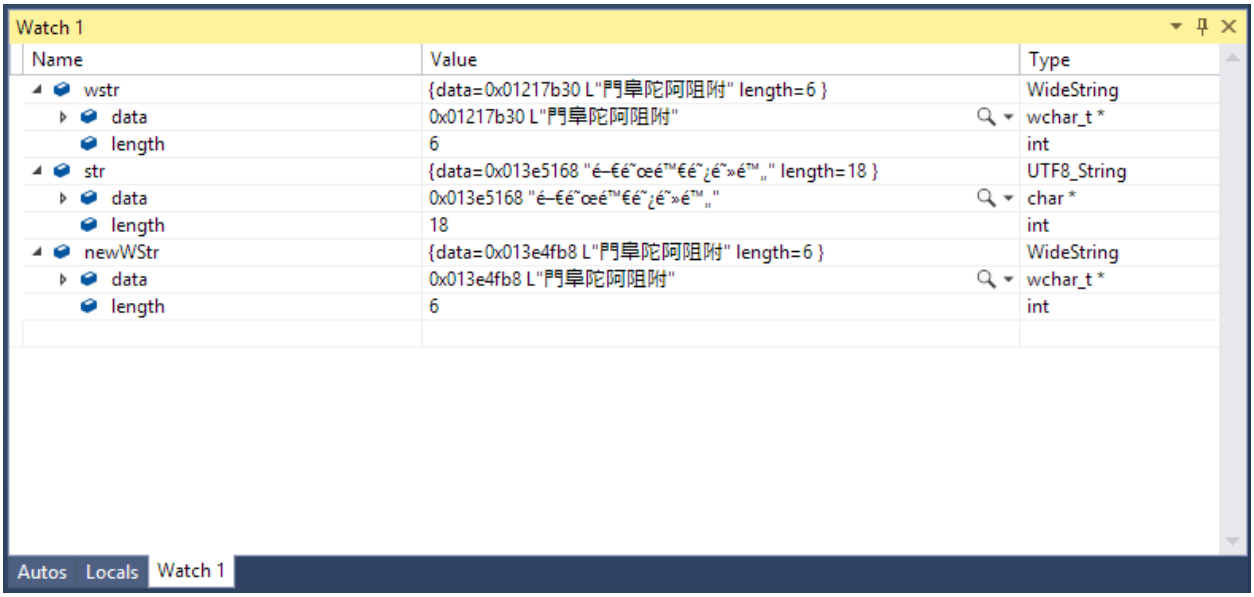

Figure 12: The Watch window. One of the many useful ways of inspecting variables while the code is paused at a breakpoint.

To fix this, we need to consult the use of some format specifiers[14]. One of the supported format specifier is `s8`, which allows the debugger to parse the watched data (or inspected data) as a UTF-8 string.

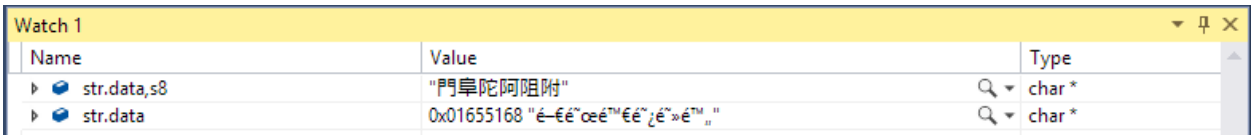

Figure 13: Using Format Specifiers in the Debugger.

What if we do not want to use the Windows header file, and instead, have to rely on standard headers instead? For such purposes, we can take a look at an alternative code snippet that will not only handles UTF-8 to UTF-16 conversion, but also handles UTF-16 to UTF-8 conversion. The code can be considered to be compatible with GNU C compiler (GCC), because it uses the standard headers and does not rely on the Windows header at all. This code is written on the next page.

[14] Format specifiers for C++ are used to assist the debugging session in Visual Studio. (Microsoft, 2018)

```c
//This is the C11 standard header file to process Unicode UTF-16 and UTF-32 formats.
#include <uchar.h>
#include <stdio.h>

#define __STD_UTF_16__

int main() {
	char16_t* ptr_string = (char16_t*) u"我是誰";

	//C++ disallows variable-length arrays.
	//GCC uses GNUC++, which has a C++ extension for variable length arrays.
	//It is not a truly standard feature in C++ pedantic mode at all.
	//https://stackoverflow.com/questions/40633344/variable-length-arrays-in-c14
	char buffer[64];
	char* bufferOut = buffer;

	//Must zero this object before attempting to use mbstate_t at all.
	mbstate_t multiByteState = {};

	//c16 = 16-bit Characters or char16_t typed characters
	//r = representation
	//tomb = to Multi-Byte Strings
	while (*ptr_string) {
		char16_t character = *ptr_string;
		size_t size = c16rtomb(bufferOut, character, &multiByteState);
		if (size == (size_t) -1)
			break;
		bufferOut += size;
		ptr_string++;
	}

	size_t bufferOutSize = bufferOut - buffer;
	printf("Size: %zu - ", bufferOutSize);
	for (int i = 0; i < bufferOutSize; i++) {
		printf("%#x ", +(unsigned char) buffer[i]);
	}

	//This statement is used to set a breakpoint. It does not do anything else.
	int debug = 0;
	return 0;
}
```

But if you want to compile the code above using the GCC compiler, then execute the output program, the program will display the wrong results. The `bufferOutSize` would be printed as `0`, and the `buffer` array would be empty, and thus displaying a blank empty line. The main cause is the function `c16rtomb()` is returning `-1`, and inspecting `errno` will give `84`.[15]

To fix this issue, call the function `setLocale(LC_ALL, "en_US.utf8")` near the top of the program, before calling `c16rtomb()`. `LC_ALL` stands for "the entire locale," and is usually referred to the

[15] "errno" is defined in <errno.h>, a C standard header file. If the header file is included, then "errno" variable is globally scoped. When the "errno" is 84, the human readable error message states, "Invalid or incomplete multibyte or wide character." This message can be obtained by #including <string.h> and then passing in the "errno" to the function, `strerror()`, or to its thread-safe counterpart, `strerror_r()`.

environment variable overriding all other localization settings on the system in general. The target locale is then passed in as the second parameter, which in this case, it's based on the American English, encoded in UTF-8.

On a Linux/POSIX system, you can find out more by entering the command, "locale," in the terminal. Here is the output, copied from the terminal running on Ubuntu 14.04.

```
Nintendo@NorthAmerica: ~
$ locale
LANG=en_US.UTF-8
LANGUAGE=
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
Nintendo@NorthAmerica: ~
$
```

And here is the modified code for compiling to GCC, before moving on to the next subchapter:

```
#include <stdio.h>
#include <uchar.h>
#include <locale.h>

int main() {
	//Here is where one would usually put this function at, near the
	//beginning of the program.
	setlocale(LC_ALL, "en_US.utf8");

	char16_t* ptr_string = (char16_t*) u"我わが是ぜ誰だれ";
	char buffer[64];
	char* bufferOut = buffer;
	mbstate_t multiByteState = {};
	while (*ptr_string) {
		char16_t character = *ptr_string;
		size_t size = c16rtomb(bufferOut, character, &multiByteState);
		if (size == (size_t) -1)
			break;
		bufferOut += size;
		ptr_string++;
	}
	size_t bufferOutSize = bufferOut - buffer;
	printf("Size: %zu - ", bufferOutSize);
	for (int i = 0; i < bufferOutSize; i++)
		printf("%#x ", +(unsigned char) buffer[i]);
	return 0;
}
```

The output of the program code for both instances should be like the following result:

```
Size: 9 - 0xe6 0x88 0x91 0xe6 0x98 0xaf 0xe8 0xaa 0xb0
```

The byte values can then be written to a binary file, a text file, or do whatever the user wants to do with them.

For your information, it is recommended ***NOT*** to touch the environment variable, LC_ALL, on your Linux/POSIX machine, and it should stay empty by default. For Windows, this environment variable is automatically set to the current system locale when the program is executed. For more information about the locale settings on the Windows side, see the Remarks section of the MSDN article, setlocale() and _wsetlocale().[16]

The environment variable, LC_ALL, exists only for developers to quickly change all the locale settings to a particular locale for debugging purposes, acting like a global override. The locale settings refer to all of the other LC_ environment variables listed out when entering the command, "locale," in the terminal. And those are the locale settings you can configure freely.

But if you have installed language packages or any locale packages and you have encountered some issues, either reconfigure the locale settings, or reconfigure in conjunction with the LC_ALL may be done so to troubleshoot some issues. Do this while staying cautious, and be sure to revert LC_ALL back to empty or unset it after all of this is done.

Converting UTF-8 to UTF-16

Before we actually get to the programming part of this section, we need to start some theory-crafting and some mathematical deductions to derive the algorithmic concept to convert UTF-8 encoding into UTF-16 encoding. Understanding all of this is not really that important, but it is a big deal to know how we get ourselves there.

Recall that a Unicode code point can be represented both as a UTF-16 character, and as a UTF-8 character. If written in a mathematical way, the equation would look somewhat like this:

$$U_C = U_{UTF-16} = U_{UTF-8}$$

Where U_C refers to any particularly given valid Unicode code point, U_{UTF-16} refers to Unicode character encoded in UTF-16, and U_{UTF-8} refers to Unicode character encoded in UTF-8. The goal here is to derive all of the steps necessary to convert between U_{UTF-16} and U_{UTF-8}.

We can derive U_{UTF-16} by splitting it up into two groups, the UCS-2 code points and the Surrogate Pairs. But first, we need to create ourselves a new mathematical symbol.

Using operands X and Y, we define the integer division operation to be shown like so:

$$\mathrm{X} \setminus \mathrm{Y} \equiv \lfloor X \div Y \rfloor$$

[16] MSDN documentation page, titled "setlocale and _wsetlocale." [Microsoft, 2016]. The default Windows system locale is "C."

In which ÷ is a normal mathematical division and ⌊ ⌋ denotes the floor function. We also define that the integer division has a higher order of precedence over addition. Therefore, we can say

$$\begin{cases} U_{UTF-16} = U_C, & \forall \begin{cases} U_C \in [0..D7FF_{hex}] \\ U_C \in [E000_{hex}..FFFF_{hex}] \end{cases} \\ U_{UTF-16} = U_C = H_{UTF-16}L_{UTF-16}, & \forall U_C \in [10000_{hex}..10FFFF_{hex}] \end{cases}$$

Of which, $H_{UTF-16}L_{UTF-16}$ is a surrogate pair, H_{UTF-16} is the high surrogate, and the L_{UTF-16} is the low surrogate. The surrogate pair can be explained in the following equations:

$$\begin{cases} H_{UTF-16} = (U_C - 10000_{hex}) \setminus 400_{hex} + D800_{hex} \\ L_{UTF-16} = (U_C - 10000_{hex}) \bmod 400_{hex} + DC00_{hex} \end{cases}$$

If you recalled that we defined U_C to be of any given valid Unicode code point. This actually means U_C resides in a range, and the range starts from 0 to 0x10FFFF. Since we are doing surrogate pairs, the actual valid range is $[10000_{hex}..10FFFF_{hex}]$. This also means, we can plug in a chosen number within $[10000_{hex}..10FFFF_{hex}]$, and simplify the equations so all of the numbers have a range instead. We now apply the range into the equations above, and solve the equations, like so:

$$\begin{cases} H_{UTF-16} \in ([10000_{hex}..10FFFF_{hex}] - 10000_{hex}) \setminus 400_{hex} + D800_{hex} \\ L_{UTF-16} \in ([10000_{hex}..10FFFF_{hex}] - 10000_{hex}) \bmod 400_{hex} + DC00_{hex} \end{cases}$$

$$= \begin{cases} H_{UTF-16} \in ([0..FFFFF_{hex}]) \setminus 400_{hex} + D800_{hex} \\ L_{UTF-16} \in ([0..FFFFF_{hex}]) \bmod 400_{hex} + DC00_{hex} \end{cases}$$

$$= \begin{cases} H_{UTF-16} \in [0..3FF_{hex}] + D800_{hex} \\ L_{UTF-16} \in [0..3FF_{hex}] + DC00_{hex} \end{cases}$$

$$= \begin{cases} H_{UTF-16} \in [D800..DBFF_{hex}] \\ L_{UTF-16} \in [DC00..DFFF_{hex}] \end{cases}$$

By applying the additions, subtractions, multiplications, integer divisions, and modulo operations to the surrogate pairs range, which is simply working with the minimum and the maximum numbers of the range while preserving the brackets, we can see the high and low surrogates are ranged exactly as stated in the previous chapter's section about UTF-16.

Putting all the equations together, we finally get:

$$\begin{cases} U_C, & \forall \begin{cases} U_C \in [0..D7FF_{hex}] \\ U_C \in [E000_{hex}..FFFF_{hex}] \end{cases} \\ U_C = H_{UTF-16}L_{UTF-16}, & \forall \begin{cases} U_C \in [10000_{hex}..10FFFF_{hex}] \\ H_{UTF-16} \in [D800..DBFF_{hex}] \\ L_{UTF-16} \in [DC00..DFFF_{hex}] \end{cases} \end{cases}$$

Ok, so what does this equation tells us? It proves to us there is a way to directly translate any given Unicode code points into UTF-16 encoded characters, and the key to do all of this is by converting UTF-8 encoded characters to Unicode. All of the basis and how the ranges are defined are taken from "A Unicode Primer (Part 2/3)", so if that part is skipped, all of this would be a quick refresher.

Now, let us work on the UTF-8 part. The equations for UTF-8 will tie back into the UTF-16 equations above, and all of this is doing is to help us craft out the programming algorithms much easier.

UTF-8 needs a bit of mathematical proof. It is going to get a bit messy, but it will be clear by the end of all this calculation.

We know that a given UTF-16 code point may have 1 code unit or 2 code units. Let U_{C1} be represented as 1 UTF-16 code unit, and U_{C2} be represented as 2 UTF-16 code units. For U_{C1}, there are a total of 63,488 code points, in the range, $\forall \begin{cases} U_C \in [0..D7FF_{hex}] \\ U_C \in [E000_{hex}..FFFF_{hex}] \end{cases}$, known as the range of the Basic Multilingual Plane (BMP). For U_{C2}, there are a total of 1,048,576 code points, in the range,

$$U_C = [H_{UTF-16}][L_{UTF-16}], \forall \begin{cases} H_{UTF-16} \in [D800..DBFF_{hex}] \\ L_{UTF-16} \in [DC00..DFFF_{hex}] \end{cases}$$

We can calculate the total valid code points to be 63,488 + 1,048,576 = 1,112,064. The total reserved code points is calculated to be $[H_{UTF-16}] + [L_{UTF-16}] = [D800..DFFF_{hex}] = 2{,}048$. We know that a full range is $[0..FFFF_{hex}][0..FFFF_{hex}] = [2^{16}][2^{16}] = 2^{16} \times 2^{16} = 2^{32}$ = 1,114,112 = 110000_{hex}.

Inferring from this, we can also say the sum of the total valid code points and the reserved code points is to be the full range, $[0..10FFFF_{hex}]$, with a length of 110000_{hex}, which is the range including the BMP and 16 other Supplementary Planes (SPs).

We know that, for any given UTF-8 code point, it will use up to 4 bytes of memory space. Thus, we need to categorize and filter out all the possible UTF-8 values.

Let U_C be the given UTF-8 code point. $U_{C1}$is a UTF-8 code point in the range $[0..7F_{hex}]$, $U_{C2}$is a UTF-8 code point in the range $[80..7FF_{hex}]$, $U_{C3}$is a UTF-8 code point in the range $[800..FFFF_{hex}]$, and $U_{C4}$is a UTF-8 code point in the range $[10000..1FFFFF_{hex}]$. Limiting the range of U_C to be within a valid UTF-16 range consisting of 1 BMP and 16 SPs, we have leftover "unallocated values" that are not considered to be part of the Unicode Transformation Format (UTF). Thus, we cannot say those "values" are code points, regardless if they are valid or invalid.

Tidying up, this is what we know thus far:

$$\forall\, U_C \begin{cases} U_{C1} \in [0..7F_{hex}] \\ U_{C2} \in [80..7FF_{hex}] \\ U_{C3} \in [800..FFFF_{hex}] \\ U_{C4} \in [10000..10FFFF_{hex}] \\ U_{CX} \in [110000..3FFFFFF_{hex}] \\ U_{CY} \in [4000000..7FFFFFFF_{hex}] \end{cases}$$

Since all of the values in the range U_{CX} and U_{CY} are not valid UTF code points, we can ignore them and only concentrate on the first 4 ranges. We finally can assign the ranges of UTF-8 code points to UTF-16 code points. The logic and algorithm to do so is given on the next page, and it is written with the assumption that we are iterating through each character in the string input and matching it up with the corresponding output.

This is the full code to convert UTF-8 strings into UTF-16 strings, with full commentary.

```
//This is the C11 standard header file to process Unicode UTF-16 and UTF-32 formats.
#include <uchar.h>

#define __STD_UTF_16__

//Pointer arrays must always include the array size, because pointers do not know about
the size of the supposed array size.

void utf8_to_utf16(unsigned char* const utf8_str,
                   int utf8_str_size,
                   char16_t* utf16_str_output,
                   int utf16_str_output_size) {
  //First, grab the first byte of the UTF-8 string
  unsigned char* utf8_currentCodeUnit = utf8_str;
  char16_t* utf16_currentCodeUnit = utf16_str_output;
  int utf8_str_iterator = 0;
  int utf16_str_iterator = 0;

  //In a while loop, we check if the UTF-16 iterator is less than the max output size. If
  //true, then we check if UTF-8 iterator is less than UTF-8 max string size. This
  //conditional checking based on order of precedence is intentionally done so it
  //prevents the while loop from continuing onwards if the iterators are outside of the
  //intended sizes.
  while (*utf8_currentCodeUnit &&
        (utf16_str_iterator < utf16_str_output_size ||
         utf8_str_iterator < utf8_str_size)) {
    //Figure out the current code unit to determine the range. It is split into 6 main
    //groups, each of which handles the data differently from one another.
    if (*utf8_currentCodeUnit < 0x80) {
      //0..127, the ASCII range.

      //We directly plug in the values to the UTF-16 code unit.
      *utf16_currentCodeUnit = (char16_t) (*utf8_currentCodeUnit);
      utf16_currentCodeUnit++;
      utf16_str_iterator++;

      //Increment the current code unit pointer to the next code unit
      utf8_currentCodeUnit++;

      //Increment the iterator to keep track of where we are in the UTF-8 string
      utf8_str_iterator++;
    }
    else if (*utf8_currentCodeUnit < 0xC0) {
      //0x80..0xBF, we ignore. These are reserved for UTF-8 encoding.
      utf8_currentCodeUnit++;
      utf8_str_iterator++;
    }
    else if (*utf8_currentCodeUnit < 0xE0) {
      //128..2047, the extended ASCII range, and into the Basic Multilingual Plane.

      //Work on the first code unit.
      char16_t highShort = (char16_t) ((*utf8_currentCodeUnit) & 0x1F);

      //Increment the current code unit pointer to the next code unit
      utf8_currentCodeUnit++;
```

```
  //Work on the second code unit.
  char16_t lowShort = (char16_t) ((*utf8_currentCodeUnit) & 0x3F);

  //Increment the current code unit pointer to the next code unit
  utf8_currentCodeUnit++;

  //Create the UTF-16 code unit, then increment the iterator
  int unicode = (highShort << 8) | lowShort;

  //Check to make sure the "unicode" is in the range [0..D7FF] and [E000..FFFF].
  if ((0 <= unicode && unicode <= 0xD7FF) ||
      (0xE000 <= unicode && unicode <= 0xFFFF)) {
    //Directly set the value to the UTF-16 code unit.
    *utf16_currentCodeUnit = (char16_t) unicode;
    utf16_currentCodeUnit++;
    utf16_str_iterator++;
  }

  //Increment the iterator to keep track of where we are in the UTF-8 string
  utf8_str_iterator += 2;
}
else if (*utf8_currentCodeUnit < 0xF0) {
  //2048..65535, the remaining Basic Multilingual Plane.

  //Work on the UTF-8 code units one by one.
  //If drawn out, it would be 1110aaaa 10bbbbcc 10ccdddd
  //Where a is 4th byte, b is 3rd byte, c is 2nd byte, and d is 1st byte.
  char16_t fourthChar = (char16_t) ((*utf8_currentCodeUnit) & 0xF);
  utf8_currentCodeUnit++;
  char16_t thirdChar = (char16_t) ((*utf8_currentCodeUnit) & 0x3C) >> 2;
  char16_t secondCharHigh = (char16_t) ((*utf8_currentCodeUnit) & 0x3);
  utf8_currentCodeUnit++;
  char16_t secondCharLow = (char16_t) ((*utf8_currentCodeUnit) & 0x30) >> 4;
  char16_t firstChar = (char16_t) ((*utf8_currentCodeUnit) & 0xF);
  utf8_currentCodeUnit++;

  //Create the resulting UTF-16 code unit, then increment the iterator.
  int unicode = (fourthChar << 12) | (thirdChar << 8) | (secondCharHigh << 6) |
                (secondCharLow << 4) | firstChar;

  //Check to make sure the "unicode" is in the range [0..D7FF] and [E000..FFFF].
  //According to math, UTF-8 encoded "unicode" should always fall within these two
  //ranges.
  if ((0 <= unicode && unicode <= 0xD7FF) ||
      (0xE000 <= unicode && unicode <= 0xFFFF)) {
    //Directly set the value to the UTF-16 code unit.
    *utf16_currentCodeUnit = (char16_t) unicode;
    utf16_currentCodeUnit++;
    utf16_str_iterator++;
  }

  //Increment the iterator to keep track of where we are in the UTF-8 string
  utf8_str_iterator += 3;
}
```

```
    else if (*utf8_currentCodeUnit < 0xF8) {
      //65536..10FFFF, the Unicode UTF range

      //Work on the UTF-8 code units one by one.
      //If drawn out, it would be 11110abb 10bbcccc 10ddddee 10eeffff
      //Where a is 6th byte, b is 5th byte, c is 4th byte, and so on.
      char16_t sixthChar = (char16_t) ((*utf8_currentCodeUnit) & 0x4) >> 2;
      char16_t fifthCharHigh = (char16_t) ((*utf8_currentCodeUnit) & 0x3);
      utf8_currentCodeUnit++;
      char16_t fifthCharLow = (char16_t) ((*utf8_currentCodeUnit) & 0x30) >> 4;
      char16_t fourthChar = (char16_t) ((*utf8_currentCodeUnit) & 0xF);
      utf8_currentCodeUnit++;
      char16_t thirdChar = (char16_t) ((*utf8_currentCodeUnit) & 0x3C) >> 2;
      char16_t secondCharHigh = (char16_t) ((*utf8_currentCodeUnit) & 0x3);
      utf8_currentCodeUnit++;
      char16_t secondCharLow = (char16_t) ((*utf8_currentCodeUnit) & 0x30) >> 4;
      char16_t firstChar = (char16_t) ((*utf8_currentCodeUnit) & 0xF);
      utf8_currentCodeUnit++;

      int unicode = (sixthChar << 4) | (fifthCharHigh << 2) | fifthCharLow |
                    (fourthChar << 12) | (thirdChar << 8) | (secondCharHigh << 6) |
                    (secondCharLow << 4) | firstChar;
      char16_t highSurrogate = (unicode - 0x10000) / 0x400 + 0xD800;
      char16_t lowSurrogate = (unicode - 0x10000) % 0x400 + 0xDC00;

      //Set the UTF-16 code units
      *utf16_currentCodeUnit = lowSurrogate;
      utf16_currentCodeUnit++;
      utf16_str_iterator++;

      //Check to see if we're still below the output string size before continuing,
      //otherwise, we cut off here.
      if (utf16_str_iterator < utf16_str_output_size) {
        *utf16_currentCodeUnit = highSurrogate;
        utf16_currentCodeUnit++;
        utf16_str_iterator++;
      }

      //Increment the iterator to keep track of where we are in the UTF-8 string
      utf8_str_iterator += 4;
    }
    else {
      //Invalid UTF-8 code unit, we ignore.
      utf8_currentCodeUnit++;
      utf8_str_iterator++;
    }
  }

  //We clean up the output string if the UTF-16 iterator is still less than the output
  //string size.
  while (utf16_str_iterator < utf16_str_output_size) {
    *utf16_currentCodeUnit = '\0';
    utf16_currentCodeUnit++;
    utf16_str_iterator++;
  }
}
```

Converting UTF-32 to UTF-8

UTF-32 is a Unicode Transformation Format (UTF) in 32 bits, that uses exactly 32 bits per Unicode code point. Thus, UTF-32 is a type of fixed-length encoding, which is unique in that all other UTFs are variable-length encodings. Each 32-bit value in a UTF-32 character represents one Unicode code point and it is exactly the same number of that code point's numerical value.[17]

UTF-32, like its cousin, UTF-16, has the property where the bytes of individual characters may differ based on the encoding byte order endianness. Meaning, UTF-32 can be encoded in Big Endian, or in Little Endian. To denote the difference, we would say UTF-32 encoded in Big Endian as UTF-32BE, and encoded in Little Endian as UTF-32LE.

Now that we understand what UTF-32 is, we can try to determine the correlation of a UTF-32 character's code point with UTF-32 character's numeric value, and use that information to encode the character to UTF-8. It is as simple as converting a Unicode code point to the equivalent in UTF-8. The following code snippet shows how to convert a UTF-32 encoded string into a UTF-8 encoded string.

```
//Required headers on Windows. Program is running in Visual Studio 2017 v15.9.6.
#include <Windows.h>

//C doesn't have bool, so we use the latest possible ISO C standard, C11,
//supported in Visual Studio 2017.
#include <stdbool.h>

//In C and C++, pointers must always trail in the back of the struct.
typedef struct UTF32_String {
   bool isLittleEndian;
   unsigned int length;
   unsigned int dataLength;
   unsigned int* data;
} UTF32_String;

typedef struct UTF8_String {
   unsigned int length;
   unsigned int dataLength;
   unsigned char* data;
} UTF8_String;

void utf32_to_utf8_process(UTF32_String* in, UTF8_String* out) {
   //Assuming all inputs are in Big Endian, because it is easier to handle the abstract
   //math involved.
   for (unsigned int i = 0; i < in->length; i++) {
      unsigned int* iterator = in->data + i;

      //Filter the code point based on number range
      if (0 <= *iterator && *iterator <= 0x7F) {
         *(out->data + out->length) = (unsigned char) *iterator;
         out->length++;
```

[17] Taken from an excerpt mentioned from the SIL International website, (Summer Institute of Linguistics, 2001).

```
        }
        else if (*iterator <= 0x7FF) {
            unsigned char firstByte = (unsigned char) ((*iterator & 0x7C0) >> 6);
            unsigned char secondByte = (unsigned char) (*iterator & 0x3F);
            *(out->data + out->length) = firstByte | 0xC0;
            *(out->data + (out->length + 1)) = secondByte | 0x80;
            out->length += 2;
        }
        else if (*iterator <= 0xFFFF) {
            unsigned char firstByte = (unsigned char) ((*iterator & 0xF000) >> 12);
            unsigned char secondByte = (unsigned char) ((*iterator & 0xFC0) >> 6);
            unsigned char thirdByte = (unsigned char) (*iterator & 0x3F);
            *(out->data + out->length) = firstByte | 0xE0;
            *(out->data + (out->length + 1)) = secondByte | 0x80;
            *(out->data + (out->length + 2)) = thirdByte | 0x80;
            out->length += 3;
        }
        else if (*iterator <= 0x1FFFFF) {
            unsigned char firstByte = (unsigned char) ((*iterator & 0x1C0000) >> 18);
            unsigned char secondByte = (unsigned char) ((*iterator & 0x3F000) >> 12);
            unsigned char thirdByte = (unsigned char) ((*iterator & 0xFC0) >> 6);
            unsigned char fourthByte = (unsigned char) (*iterator & 0x3F);
            *(out->data + out->length) = firstByte | 0xF0;
            *(out->data + (out->length + 1)) = secondByte | 0x80;
            *(out->data + (out->length + 2)) = thirdByte | 0x80;
            *(out->data + (out->length + 3)) = fourthByte | 0x80;
            out->length += 4;
        }
    }
}

void utf32_to_utf8(UTF32_String* in, UTF8_String* out) {
    if (!in || in->length <= 0) {
        if (!out)
            return;
        out->data = NULL;
        out->length = 0;
        out->dataLength = 0;
        return;
    }

    if (in->isLittleEndian) {
        for (unsigned int i = 0; i < in->length; i++) {
            unsigned int* iterator = in->data + i;
            unsigned short high = *iterator & 0xFFFF;
            unsigned short shortTemp = ((high & 0xFF) << 8) |
                                       ((high & 0xFF00) >> 8);
            high = shortTemp;

            unsigned short low = (*iterator >> 16) & 0xFFFF;
            shortTemp = ((low & 0xFF) << 8) | ((low & 0xFF00) >> 8);
            low = shortTemp;

            unsigned int highInt = high << 16;
            unsigned int lowInt = low;

            *iterator = highInt | lowInt;
        }
```

```
      utf32_to_utf8_process(in, out);
   }
   else
      utf32_to_utf8_process(in, out);
}

int main() {
   HANDLE heapHandle = GetProcessHeap();
   if (heapHandle == NULL) {
      //Error getting the current process's private heap.
      return -1;
   }

   //Readable text: "我是誰?"
   //Data source is in Little Endian format.
   unsigned int utf32_data_le[] = {
      0x11620000, 0x2f660000, 0xb08a0000, 0x3f000000
   };

   UTF32_String str;
   //UTF-32 string length.
   str.length = sizeof(utf32_data_le) / sizeof(utf32_data_le[0]);
   //UTF-32 string data memory allocation size, including the NUL terminating character
   //size.
   str.dataLength = (str.length * sizeof(unsigned int)) + 1;
   //UTF-32 string memory allocation, with NUL terminating character.
   str.data = (unsigned int*) HeapAlloc(heapHandle, HEAP_ZERO_MEMORY, str.dataLength);
   for (unsigned int i = 0; i < str.length; i++)
      *(str.data + i) = utf32_data_le[i];
   //Data source is in Little Endian format.
   str.isLittleEndian = true;

   UTF8_String result;
   //UTF-8 string data memory size is determined by the number of code points
   //(UTF-32) times the number of largest UTF-8 character bytes size (4 bytes), plus
   //the NUL terminating character.
   result.dataLength = (str.length * 4) + 1;
   //UTF-8 string memory allocation, with NUL terminating character.
   result.data = (unsigned char*) HeapAlloc(heapHandle, HEAP_ZERO_MEMORY,
                                            result.dataLength);
   //UTF-8 string length is zero, because the string is empty.
   result.length = 0;

   utf32_to_utf8(&str, &result);

   //Debug statement only.
   //Put a breakpoint on this line, and then inspect "result.data", with format
   //specifier, "s8", in the Watch window in Visual Studio.
   int debug = 0;

   //RAII
   HeapFree(heapHandle, 0, str.data);
   HeapFree(heapHandle, 0, result.data);
   return 0;
}
```

As for the Debug Statement mentioned in the code snippet above, this is what happens after you set the breakpoint down.

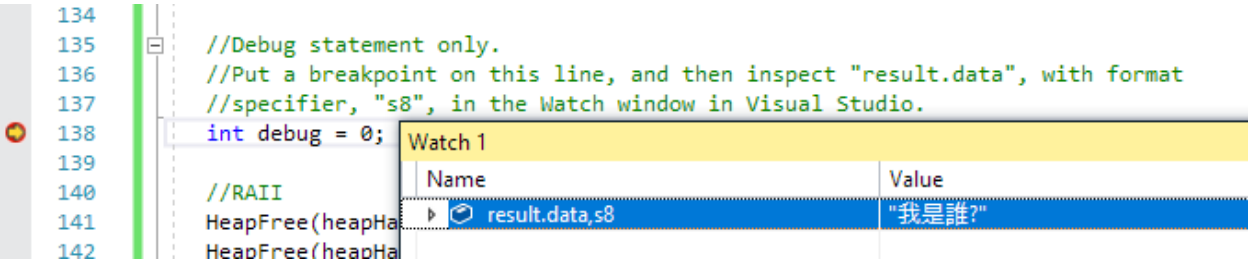

Figure 14: The Debug Statement, in action.

After the conversion has completed, we can then inspect the result in the Watch window, along with the format specifier, `s8`, to show that the conversion worked correctly. This is, of course, assuming that there are no edge case scenarios ruining the code.

Converting UTF-8 to UTF-32

Now that we have established the conversion from UTF-32 strings to UTF-8 strings, it is time for us to revert the strings back. The same steps are applied, but we can trace each step individually in reverse using the code and concepts given above. In other words, we are to convert from an integer of type `char`, to an integer of type `unsigned int`, and all of this is by doing "integer upcasting."

There are some gotchas we need to be aware of when doing "integer upcasting" and "integer downcasting." The behaviors exhibited when doing these casting conversions are as follows[18]:

- Downcasting from any large integer, signed or unsigned, to a small integer will truncate the most significant bits down enough to fit into the maximum bit size of the small integer.
- Upcasting from a small signed integer to a large signed integer, will sign-extend the source type. In other words, it will pack in bytes with bits equal to the signed bytes to preserve the signed value. Positive and negative integers are preserved.
- Upcasting from a small unsigned integer to a large unsigned integer will sign-extend the source type. Same as above, it will preserve the source integer values, but will extend all the bits.
- Upcasting from a small signed integer to a large unsigned integer will sign-extend the source type, then casts from signed to unsigned. This method of upcasting does not guarantee the source value is preserved.

[18] Most of the descriptions are copied from the answer from the user, abligh, on Stack Overflow (abligh, 2015). Stack Overflow is a great resource for finding answers to commonly asked programming questions, so there are no exceptions not to include detailed answers from Stack Overflow into this book.

- Upcasting from a small unsigned integer to a large signed integer will pad the significant bits with zeroes, thus always preserving the source value. It is guaranteed there will be one bit dedicated as the signed bit of the destination value, and the destination value will always be positive.

Because we are upcasting a `char` to an `unsigned int`, heeding the gotchas mentioned above should help.

For each of the characters in the UTF-8 string, we have to decode each character's byte codes to get the Unicode code points. Once we get the code points, we can easily store the code points, as is, in a data structure to be parsed as strings later on. The unique aspect of UTF-32 encoded characters is that all of the character's code point is the Unicode representation of the character itself. Knowing this, we can begin implementing algorithms to work out the encoding and decoding procedures.

Note that in Visual Studio, particularly obvious in Visual Studio 2017, when specified with a format specifier, `s32` or `s32b`, the UTF-32 data either in an array or in a string must be encoded in Big Endian. Even though the Windows operating systems are historically operating in Little Endian, when it comes to displaying data, it will be displayed in Big Endian, prioritizing human readability over consistency with how it operates under the hood. The format specifier, `s32`, displays the UTF-32BE string with quotation marks, while the format specifier, `s32b`, displays the same string but without the quotation marks.

Without further ado, behold, the full code:

```
//Required headers on Windows. Program is running in Visual Studio 2017 v15.9.6.
#include <Windows.h>

//C doesn't have bool, so we use the latest possible ISO C standard, C11,
//supported in Visual Studio 2017.
#include <stdbool.h>

//In C and C++, pointers must always trail in the back of the struct.
typedef struct UTF32_String {
   bool isLittleEndian;
   unsigned int length;
   unsigned int dataLength;
   unsigned int* data;
} UTF32_String;

typedef struct UTF8_String {
   unsigned int length;
   unsigned int dataLength;
   unsigned char* data;
} UTF8_String;

const unsigned int BOM_LittleEndian = 0xFFFE0000;
const unsigned int BOM_BigEndian = 0x0000FEFF;
```

```
void utf8_to_utf32(UTF8_String* in, UTF32_String* out) {
   unsigned int tempByteCodeHolder = 0;
   unsigned int partialBytesCount = 0;

   //BOM is intentionally not counted towards the UTF-32 string length.
   if (out->isLittleEndian)
      *out->data = BOM_LittleEndian;
   else
      *out->data = BOM_BigEndian;

   for (unsigned int i = 0; i < in->dataLength; i++) {
      unsigned char* iterator = in->data + i;
      if (*iterator < 0x80) {
         //UTF-8 byte code representing a valid ASCII character.
         //Just push it in.
         tempByteCodeHolder = *iterator;
         partialBytesCount = 0;
      }
      else if (*iterator < 0xC0) {
         //UTF-8 byte code representing a valid portion of the character. It
         //needs to be merged into the other valid portions of the character's
         //bytecodes.
         unsigned int bits = *iterator & 0x3F;
         tempByteCodeHolder = (tempByteCodeHolder << 6) | bits;
         partialBytesCount--;
      }
      else if (*iterator < 0xE0) {
         //We have to save 5 bits stored in this byte code to the temporary
         //holder.
         tempByteCodeHolder = *iterator & 0x1F;

         //There is only 1 partial byte count.
         //(header byte + 1 number of partial bytecodes)
         partialBytesCount = 1;
      }
      else if (*iterator < 0xF0) {
         //We have to save 4 bits stored in this byte code to the temporary
         //holder.
         tempByteCodeHolder = *iterator & 0x0F;

         //There are only 2 partial bytes count.
         //(header byte + 2 number of partial bytecodes)
         partialBytesCount = 2;
      }
      else if (*iterator < 0xF1) {
         //We have to save 3 bits stored in this byte code to the temporary
         //holder.
         tempByteCodeHolder *= *iterator & 0x07;

         //There are only 3 partial bytes count.
         //(header byte + 3 number of partial bytecodes)
         partialBytesCount = 3;
      }
      else {
         //Invalid UTF-8 byte code. Skipping. We cannot just let the code
         //hang here.
         continue;
      }
```

```
      //When the "partialBytesCount" reaches zero, it marks the end of the
      //previous character's byte code, thus we can push the previous value
      //to the output string and clear the bits.
      //This code also ensures "tempByteCodeHolder" holds data. If it does not,
      //then we do not do anything with it. (ASCII character check)
      if (partialBytesCount == 0 && tempByteCodeHolder) {
         //Converting the UTF-32 value to the system's byte order.
         if (out->isLittleEndian) {
            unsigned char firstByte = (tempByteCodeHolder & 0xFF000000) >> 24;
            unsigned char secondByte = (tempByteCodeHolder & 0xFF0000) >> 16;
            unsigned char thirdByte = (tempByteCodeHolder & 0xFF00) >> 8;
            unsigned char fourthByte = (tempByteCodeHolder & 0xFF);

            //Combine all bytes in reverse order.
            tempByteCodeHolder =
               (fourthByte << 24) | (thirdByte << 16) |
               (secondByte << 8) | firstByte;
         }
         //We skip the first byte, which is always the BOM byte.
         *(out->data + out->length + 1) = tempByteCodeHolder;
         out->length++;

         //Clears the temporary data holder once we are done with it.
         tempByteCodeHolder = 0;
      }
   }
}



int main() {
   HANDLE heapHandle = GetProcessHeap();
   if (heapHandle == NULL) {
      //Error getting the current process's private heap.
      return -1;
   }

   //Readable text: "我是誰?"
   //For learning purposes, these are the values of the readable text.
   unsigned char utf8_data[] = {
      0xe6, 0x88, 0x91, 0xe6, 0x98, 0xaf, 0xe8, 0xaa, 0xb0, 0x3f, 0
   };
   unsigned int utf32_data_le[] = {
      0xfffe0000, 0x11620000, 0x2f660000, 0xb08a0000, 0x3f000000, 0
   };
   unsigned int utf32_data_be[] = {
      0x0000feff, 0x00006211, 0x0000662f, 0x00008ab0, 0x0000003f, 0
   };

   //Initializing input and output string data.
   UTF8_String str;
   str.length = 4;
   str.dataLength = sizeof(utf8_data) / sizeof(utf8_data[0]);
   str.data = (unsigned char*) HeapAlloc(heapHandle,
                                         HEAP_ZERO_MEMORY,
                                         str.dataLength + 1);
```

```
    //Populating the input string data.
    for (unsigned int i = 0; i < str.dataLength; i++) {
       *(str.data + i) = utf8_data[i];
    }

    UTF32_String result;
    result.length = 0;
    result.dataLength = str.length;
    //To hold the data, the BOM, and the NUL terminator character.
    result.data = (unsigned int*) HeapAlloc(heapHandle,
                                            HEAP_ZERO_MEMORY,
                                            result.dataLength + 2);
    //Visual Studio debugger parse UTF-32 in Big Endian.
    result.isLittleEndian = false;

    //Finally, we invoke the function.
    utf8_to_utf32(&str, &result);

    //Debug statement only.
    //Put a breakpoint on this line, and then inspect "result.data", with format
    //specifier, "s8", in the Watch window in Visual Studio.
    int debug = 0;

    //RAII
    HeapFree(heapHandle, 0, str.data);
    HeapFree(heapHandle, 0, result.data);
    return 0;
}
```

By using the provide “Debug Statement,” we can inspect the data in the Watch window, as shown in Figure 15.

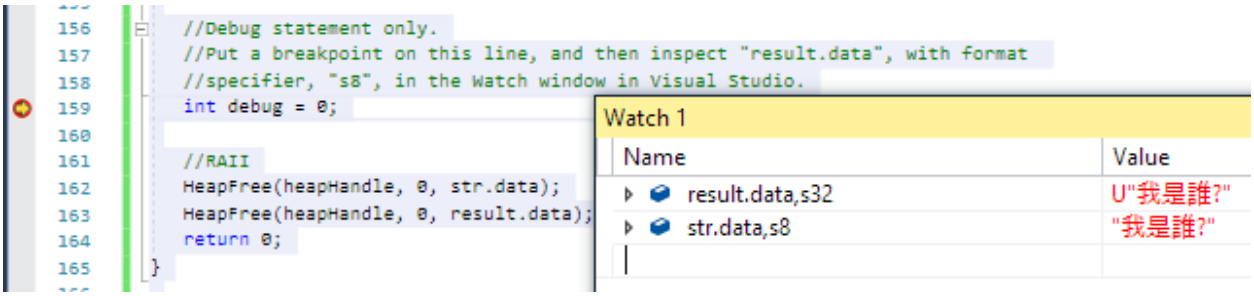

Figure 15: The Debug Statement, in action again.

As always, be sure to remember the special quirk in any Visual Studio debugger when parsing UTF-32 encoded strings. If you happen to see the debugger not being able to parse out the string, or it is telling you there are invalid numbers, then it may be because the UTF-32 string is encoded in Little Endian, or the decoded characters have incorrect values.

Converting UTF-32 to UTF-16

Converting UTF-16 to UTF-32

Source code for working with UTF-8/UTF-16/UTF-16:

https://android.googlesource.com/platform/frameworks/native/+/android-4.2.2_r1/include/utils/Unicode.h

https://android.googlesource.com/platform/frameworks/native/+/android-4.2.2_r1/libs/utils/Unicode.cpp

END OF BOOK (Temp)