Section 2 Lesson 0

Section 2 lesson 0 Windows

I Discuss new data structures, algorithms, or language features

S2L0_Windows_intro has all the files necessary to build a minimal Windows application. It has two rudimentary but functional menus, an empty graphics window, and allows the use of four accelerator keys. IT IS THE ONLY CODE IN THE ENTIRE COURSE WHICH I DID NOT WRITE. I found this example code on Transmission Zero's site a number of years ago and have used it to write applications using portable code. Please visit their site for the latest version of this code. I have included it in this folder but you need to visit their site for an explanation of everything they did. I am describing the example I found there originally, but you can use their latest example code if you like.

You need to be in a Windows environment, with MinGW installed, to compile this code. You can run the executable in that environment with no problems at all. If you are using a Linux or a Macintosh computer, you will have to install wine to run the application. Every Windows graphics app I've written runs properly under wine, and under Windows, without difficulty.

II Describe functions and modules - why

Win32App.exe is a rudimentary program. Only large enough to do a few things: display itself, take input from the mouse or keyboard, display an About dialog box, and respond to a few commands. But, it is a great framework to use for learning and creating new applications. You are ready to start developing a new app quickly by copying the entire folder, pasting it to a new location, and renaming it. This gives you a framework to create graphics when text output is just not enough. Plus, it has that cool factor. Lots of folks can whip out a text app, but how many can create a windowing app with cool colors, menus, and dialog boxes?

Each lesson in this section will expand on what I've covered here. I will explain each new command as it is added to the callback structure. WM_PAINT will do a lot but there are many more WM_ commands to make life fun. I really enjoyed learning about a WM_MOUSEDOWN_EVENT. Sounded like we were being invaded by space rodents!

Keep the file structure you get when you unzip the file. Otherwise the code will not compile properly. This structure is very useful, when things get complex, with dozens of files floating around. A good naming structure and folder categories can maintain your sanity.

III Compile and run the code

- Open a terminal window
- Navigate to … S2L0 using **cd** (change directory in Linux or Windows using MinGW)
- Type **make** in the MinGW environment which will link and compile the application called Win32App.exe
- Type **wine Win32App** (in Linux) or **Win32App** (in Windows) to run the application
- Read winmain.c, callbacks.c, callbacks.h, resource.h, resource.rc, and the Application.manifest in your editor to find how the code creates the application's output
- Open Application.ico with MS Paint or a similar program under Linux.

IV Describe code line by line - how

Windowing apps are a little different. They run as two parts: the event loop, which sends event messages, and the window procedure, which handles each event's message. In this application, the event loop is in winmain.c in the function WinMain(), and the event messages are processed by MainWndProc(). Messages are queued so you can send them rapidly, then they are processed in order. Later, we will examine threading, where the user interface (UI) has one thread of execution, and the working part of your app will be in another thread. As ever, those are more complex. This template app only recognizes enough messages to be useful, it is a skeleton. But, with so little content, you can see the structure better. More complex apps are just longer, not that much more complicated. Menus take diligence, they are not hard, just follow the templates.

winmain.c initializes the windowing environment then enters the event loop, awaiting orders from the user, the application, or the OS. Line 11 is where you change the title in the top border of the application window. Line 47 is where the graphics window is created. CreateWindowEx() is a complex function, I will only discuss a little of it here. If you want to change the size of the window, or where the upper left corner is located on start up, change them here.

Line 58 loads the accelerator keys, while the menus are created in lines 62 & 63. The accelerator keys are identified in resource.rc on lines 10, 12, 14, and 16 with the & character. Thus f, x, h, and a are accelerator keys. If you hit the alt key while inside Win32App.exe you will see the F on the file menu and the H on the help menu change to underlined. That tells you they are available as accelerator keys. Now type alt-F to open the File menu, showing Exit as the only choice, with x underlined. Type alt-f followed by alt-x to exit the app. Or type alt-h to see the help menu and alt-a to read the about window.

Accelerator keys are useful for navigating menus without using the mouse. Keeping your fingers on the keyboard makes data input much easier. Don't force the user to use the mouse unless you have to.

Lines 70 through 77 are the event loop, where OS events are captured in winmain.c. All of those events are passed to MainWndProc() for processing. These events are the menu choices, the About dialog procedures, the sizing area, and the quit box. The event loop captures keystrokes like the alt-f alt-x sequence. If the event is not handled by a defined callback function it is ignored.

Normally, winmain.c is unchanged except for line 11, to title the window, line 47 to change the initial size of the graphics window, and lines 62 & 63 to change the menus. More menus can be added here, after the file menu and before the about menu, to maintain normal Microsoft Windows standards.

callbacks.c and **callbacks.h** is where the work gets done. MainWndProc() processes commands; in this case ID_HELP_ABOUT, ID_FILE_EXIT, WM_GETMAXINFO, WM_SYSCOMMAND, WM_CREATE, & WM_DESTROY are defined. ID_ designates menu items from the resource file and WM_ designates commands built into the Windows API.

The File and Help menus each have a single choice: exit the program, and display an About dialog box.

WM_GETMINMAXINFO is fun, it looks at the system information and limits the ability to size the window. Currently it is filled with 'magic' numbers; so called because they are numbers in the middle of the program which the programmer knows about but which are inaccessible to others. It is better to make them #define statements so you can modify them right at the top of the file. But, in the heat of the moment, they do the job. But, remember where they are for later when the errors start.

WM_SYSCOMMAND has only one option for now - show the About dialog box which is stored in the resource fork.

WM_CREATE is where the window actually gets made. Later you will also see WM_PAINT which used to put interesting stuff inside that window.

WM_DESTROY is where you exit the app, no matter how you got here it is the final path out.

The only other callback function in this template application is AboutDialogProc(). It displays the About dialog box stored in the resource fork, and accepts a few commands: you can either hit the X in the window's quit box, or the OK button. Either one goes to the same place - EndDialog() which ends the dialog procedure.

resource.rc and **resource.h** describe the menus and the dialog box. Both the menus and the dialog box are known as resources. In a fancy, full blown MS Visual Studio, IDE you have editors for this purpose. But, here we are old school :) I will need to dig to find out how to create and modify icons. For now, we won't worry about them. They do make things look spiffy though. {Note: I was able to open Application.ico with MSPaint. So you can edit icons there.}

Notice that macros in an .RC file are prefaced with an ID. Those ID numbers are defined in resource.h. The resource compiler wants the number, but we poor programmers use words, because we understand them better.

Lines 8 through 18 (of **resource.rc)** is where the menus get formed. The File menu contains Exit as its only choice. The Help menu too has only one choice: About. Menus can grow quite extensive, this is where it is all held together. Well, here and in resource.h.

Don't fiddle too much with the next section, lines 20 through 55. FileVersion, ProductVersion, on lines 25 & 26 are OK; and lines 41 through 48. All this information shows up when you right click on the .exe file and ask for Properties. Convenient place to store all this data for later.

Lines 57 through 67 is where the About dialog box is defined and filled. Notice the LTEXT lines (64 and 65) after IDC_STATIC 34, 7 is the x, y location of the first printed text. Adding 10 steps to the next line at 34, 17 and so forth.

Lines 70 through 73 define the accelerator key behavior for alt-A. If you try alt-x without first hitting alt-f nothing will happen. If you hit alt-a at any time you will display the dialog box.

resource.h has the #define statements to keep both compiler and programmer happy. This is your scorecard. These numbers don't really have any significance, but numbers in the 100s usually are the lowest level of menus while the larger numbers are the deeper choices. Make sure each one is unique or odd things happen which is hard to debug.

V Describe makefile line by line

The author is emulating the style of professional makefiles. Many macros are used, automatic variables are demonstrated, and individual folders are used for the objects, the include files, and the resources. The use of both console and graphics windows is shown in lines 12 and 13. The makefile is much more complex for this application. The resources need to be compiled and linked into the executable also. The makefile is sensitive to the size of characters. Unicode characters take more space, which requires modified print statements. Using international character sets is much easier with this method. Since windowing apps need to display characters one uses the TEXT macro. This calls an ASCII/Unicode sensitive print statement. If one sets CHARSET equal to ANSI the makefile creates an ANSI executable. Since it is not set the makefile creates the Unicode version of the app. There is also a provision for opening a console as well as a window; this is very convenient to error check variables using print statements.

Please skim this section the first time through. It is not necessary for you to fully understand each line of this makefile. Over the succeeding lessons, the makefile will be simplified, and each of the macros and automatic variables explained. Remember, this is how professional makefiles look. By the end of this series of lessons you will be able to understand and create complex makefiles. I like to keep things very plain and generic, so they are easy to read and edit. However, all the complexity provides the programmer with great power. A makefile can call other makefiles recursively to create suites of tools from code you have downloaded from the web. You can download a flavor of Linux and compile it on your computer.

Lines 3 through 5 fill the HEADERS, OBJS, and INCLUDE_DIRS macros with all the files needed to make the application.

Line 7 sets the WARNS macro to all to generate all the warnings possible for cleaner code.

Line 8 introduces the resource compiler windres. This is a resource compiler for Windows which uses the resource.rc, Application.manifest, the Application.ico, and resource.h to create the resource object file.

Line 9 picks gcc as our compiler. It is adequate for this template app. You will want to change this to CC = g++ when using C++ code.

Lines 12 and 13 are very similar. Each links in the common controls (-lcomctl32), the graphics window, and the console window. But with line 13 we're adding static libraries to our build. The dynamically linked libraries, scanned when you use line 12, create a smaller application but one which requires us to be within the MinGW environment to run the executable file. Line 13 links these libraries into the code statically so you can share your application with folks running Windows or Linux (under wine) and have it run with no problems. This creates a larger executable file, but one which runs in most environments.

Lines 20 through 24 set up the compiler flags according to whether you are using Unicode characters or not. The macro CFLAGS sets the optimization level to high, the language standard to C99, and the minimum usable OS to Windows 2000. Line 23 differs, with the addition of Unicode features.

In a makefile, comments are indicated with a # symbol in the very first column. If you put the # anywhere else you get odd behavior. One common error is not using the proper indentation. Your target starts in the first column but the recipe to build it must be indented by one tab. Three spaces won't work and generates an error. Some editors change tabs into spaces while you are cutting and pasting. If you get an odd error, look at the line number in the error message. Backspace the rule to the 1st column and then hit the tab key. Often that will fix your problem.

Line 27 exists so you can type **make all** instead of just **make**. Seems rather superfluous here but will be useful later for multiple files.

Lines 29 and 30 link the object files to create an executable file. "$@" is a make automatic variable. It says to use the target of the rule (Win32App.exe) at this location. When you see "$@" substitute Win32App.exe ${OBJS} is the idiom used to represent the list of OBJS from line 4.

Lines 35 through 37 cleans up during construction. It is best to remove the previous version, and all the object files used to create it, before you rebuild the application. In this way, you get a clean build each time. Sometimes, if you recompile without removing all of the object files, the app won't link correctly. It is best to rebuild all the parts each time to be sure you get the result you intend. The first line (36) removes all of the object files within the object folder while line 37 removes the .exe file. Sometimes there is a glitch and make will throw odd errors. It is always a good idea to type **make clean** and use **make** again. Typing **make clean** forces make to run every step. Normally make will only compile, or link, the least amount necessary to get the app created. Sometimes this is not quite what you want done, so you type **make clean** to remove the .exe and object files. Then **make** will execute each rule again.

Lines 39 and 40 do a lot of work in a small amount of space, compiling each .c file into the .o file of the same name. Once again, the makefile uses internal variables < and @.   For every target in the obj folder, find its matching .c file, and compile it with the headers assigned to the macro HEADERS. In this case, you have callbacks.c and winmain.c matched with callbacks.h and winmain.h. If you had had even more files, these two lines would be that much more efficient. Line 40 uses the assigned compiler (gcc), the CFLAGS, and the include files to compile $< (the name of the next prerequisite) with a name of $@ (the name of the current target). Thus callbacks.c compiles to callbacks.o, and winmain.c compiles to winmain.o. ${INCLUDE_DIRS} tells make about any non-normal include directories necessary. Each of the object files gets created in line 40 "$@", one by one. $@ is an automatic variable in the make language. It grabs the next OBJS when it is time. ${LDFLAGS} is the idiom for typing out the long link flags from line 12.

Lines 43 & 44 compile the resources (icons, menus, etc.) and link them into the app .exe file. Once again using make's automatic variables $< & $@. obj/resource.o has these prerequisites: res/resource.rc, res/Application.manifest, res/Application.ico, and include/resource.h. These are compiled on line 44 by the windres resource compiler. Again we use $< and $@ to represent the prerequisite and the name of the target. This resource object allows your application access to the menus, accelerators, icons, etc. used by a Windows app.

VI Deeper

If you are using a Mac, or Linux OS, you must install wine to run any of the programs in either section 2 or section 4, graphics or advanced graphics. This is because I am using the Win32 application programmer’s interface (API) to create the code. Any of these apps must be compiled and linked under the MinGW environment. If you use wine and MinGW, you can write windowing applications with the win32 API, and run them on a Windows machine, on a Macintosh, or under Linux. I wrote much of this set of lessons on a Linux box using a VirtualBox virtual machine. I purchased a new Windows 7 installation disk to create the virtual machine (VM). Once it was installed, I downloaded MinGW and set it up inside of it. At that point I could write code with my Linux tools, call up the Win7 VM, and compile the code under the proper environment. I could run the application directly on the VM. Then I installed wine on my Linux machines so I can use my applications across my heterogeneous network.

I learned the language C, and how to create windowing programs, at the same time. I was using Manx Aztec C on my Macintosh, my second one if memory serves. I bought my first Mac within 100 days of them introducing it; all the designers’ signatures were embossed on the interior of the case. It was amazing but I soon ran into its limitations: severe lack of memory with no way of expanding it. However, this did not prevent me from learning C, and windowing code, on the Mac.

When I started using Windows machines I found many of the APIs were similar, if not the same. However, there was one jewel which got my attention. There was a single command to set any pixel to any color. Under the Mac APIs, I had to create really small rectangles to display a single pixel of a certain color. I also had to change brush colors and pen colors to get there. A real pain! XWindows is not much better. GLU/GLUT are designed for coloring faces of objects, not for setting pixels. But in the Windows API I found this:

```
SetPixel(hdcBack, x1, y1, RGB(r, g, b));
```

```
SetPixel(hdcBack, x1, y1, BLACK);
```

Both of these commands set a pixel in the background bitmap to a color. In the first we use a macro, RGB(), for the color choice, while in the latter we use BLACK, which was set in a #define statement. This command, and the ubiquity of Windows machines, caused me to choose the win32 API. When I can change each pixel's color, at any (x, y) point on my display, I can create comprehensive displays of data. As we go through these initial graphics lessons in section 2, you will see how I use SetPixel() to display fractals and hotplates, in my own palette of colors.

I also use MinGW on my Windows machine (and VM) so I can use the many handy Unix tools. I use two of them during code creation: gcc and make. I also use ls, rm, mkdir, rmdir, and cd on a regular basis. The Gnu compiler collection (gcc) is very handy if you are programming microcontrollers. Plus, it will compile code of C/C++, FORTRAN, and a few other languages. Using a makefile helps keep my projects under control. I know everything necessary to create my application. All the resources, all the libraries, and all the source files. Creating a makefile goes quickly once you understand the methods, target rules and recipes. Adding MinGW to your Windows box enhances it without being obtrusive; it has a small memory footprint which uses few processor cycles.

Some think an IDE, doing much of this work behind the curtains, saves time. I think exactly the opposite, because much of what I want to know is hidden. I had to spend the time learning the system and finding the exact spot to add a library or change a setting. With a makefile everything is right in front of you. If you interpret the errors your compiler sends you correctly you'll know which libraries are not available, or which files you have left out, so you can fix the makefile and run it again. Simply starting up MS Visual Studio takes quite a bit of time. I can have my text editor up and get to work within seconds. Hit save and then type **make**. Once the editor window is open and the terminal window is pointing to the right directory the edit, compile, run, edit, … , cycle goes very quickly.

I use manual tools rather than the automated ones for a few other reasons. Cost is a big concern. MS Visual Studio is not cheap, and the APIs keep changing. If I use the more generic win32 APIs instead, I can write software which doesn't become obsolete within six months. Once the makefile is correct, and the source code files are debugged, I can recompile my application years later without a problem. I don't want to get on the renew, update, renew, etc. ever again. I don't want to 'rent' my code either. By downloading NotePad+, Linux, VirtualBox, MinGW, and Eclipse I can create fine windowing code for free and stay up to date on all of them. Using the basic features of win32 assures me the application will run on a Windows box, as well as under wine on a Mac, or Linux box.

VII Assignments

1. Visit https://www.transmissionzero.co.uk/computing/win32-apps-with-mingw/ and read all of the notes. This page explains what the author intended to do and lists all the Windows features he demonstrated with his code. Follow any instructions for installing win32 libraries.
2. Visit https://www.winehq.org/ and follow all the instructions to download and install wine (if you are working on a Linux or Macintosh machine).
3. Go to http://www.winprog.org/tutorial/ and read about message handling, resources, menus, icons, dialog boxes, and standard controls.
4. Continue at http://www.winprog.org/tutorial/ and learn about bitmaps, device contexts, timers, and text.

VIII Links

http://www.transmissionzero.co.uk/computing/win32-apps-with-mingw/

https://github.com/TransmissionZero/MinGW-Win32-Application

http://www.winprog.org/tutorial/

http://www.winprog.org/tutorial/errors.html

https://www.winehq.org/