Workflow

Any new project begins with notes, tables of data, and a library of your chosen tools. You create a framework to build your application: a skeleton file for source code, a make file to build it, and your testing harness. The workday begins when you open your source code in an editor. Then you review your notes from the day before, to determine where to make changes. You edit the source files then use the make file to compile, and link them into an executable file. Next, you run the application to test your edits. You make a few notes and return to your editor to make the necessary changes. Recompile and continue.

If I am writing a command line tool I use one set of tools and windows. If I am writing a graphics application using the Win32 library I use another set. For the simplest case, I open a terminal window and type **nano appName.c** to write code. Then I type **g++ -o appName appName.c** to compile the code. Next I run the app with **./appName**. Simple and fast, but labor intensive. I did have to type the compile line, or use the up arrow key to get back to it. This method works for small chunks of code to test ideas quickly. Nano is limited but available. On a Windows machine I would use Notepad for my limited editor.

The next level would be to call up my editor to work in it, and in a terminal window. Open the terminal window and **cd** through the file system to where your project is located. Write your new app in the editor window, using its more flexible tool set. If you wish, you can use it to write your make file. Now you type **make** in the terminal window to compile and link the code. When your executable file is ready type **./appName** and it will run. Test the app for accuracy and fix any bugs you find. Go back to the editor window and start again.

If I am writing an application using graphics from the Win32 library, I add another tool: VirtualBox. VirtualBox lets me install various operating systems so I can test code in their environments. Much of the code written for this course, was tested in the Windows 7 virtual machine (VM) on VirtualBox. I use two or three windows while I am working on Win32 code. My editor window, a VM window for the Win7 environment, and possibly a terminal window. However, inside the Win7 VM I can use the command prompt which is quite similar to a terminal window under Linux. Depending on my make file settings I can send output to the VM's command prompt window as well as to the application's graphics window.

I can test and debug windowing applications inside the Win7 VM, but they will not run as fast as they could. I have installed wine on my Linux boxes so I can run Windows applications directly. Still not as fast as running them natively on a Windows box, but good enough for testing most things. Now you have the editor window open, and the Win7 VM window open with another window or two open inside of it. Or, you have your editor open, with a terminal window ready to run your graphics app using wine.

As you can see, your workflow requires more than one window, in almost all cases. Screen real estate is precious. I use multiple monitors on my main box to handle the requirement. On my laptop I use an operating system derived from Ubuntu. It is called Pop!_OS. This strange sounding OS lets me flip between workspaces. I have an editor open in one workspace while I have my Win7 VM running in another. I edit in the first space, then jump to the other space to compile, and run the app. My laptop lets me have many workspace open at once. Using them makes working on a laptop much more efficient. Instead of moving windows around to find things, I just page to the correct workspace and do the work. My workflow consists of edits in the editor, jumping to the VM, compiling, running, and testing in it's environment, then back to the editor to make any changes. On my desktop I use the mouse to jump to the correct window. On the laptop a few key strokes move me between workspaces to accomplish the same thing.

Edit the code, make the code, and run the code. This is the normal programming cycle. Make it a habit so your fingers learn the steps, while you are thinking about your code. If you don't have a makefile you will have to type **gcc -o name myfile.c** Again and again and again. Typing this only a few times shows you how a simple makefile can save time. Instead of typing **gcc -o name myfile.c** you simply type **make** which builds your executable. As your programs grow more complex, with more and longer files, typing **make** does all the work, saving even more time. The makefile is a template you can reuse. It has provisions for installation, cleaning, debugging, and profiling. Profiling is used to time each routine. It can show you where timing bottlenecks occur in your code and where to optimize.

You can use the command line more effectively after you have MinGW set up. Simply press the up arrow if you make an error while typing a command. That will restore the line you just typed, ready for editing. Use the back arrow (NOT the backspace key) to move where you need to make corrections. Once you have the error corrected hit enter, there is no need to be at the end of the line to do so. This makes life much easier, because you can arrow back to the very beginning of the command prompt session. Normally, I am working inside an editor with a command prompt window beside it.

Once I am done editing I hit **control-S** to save. This works in almost all editors. Then I type **make** in the command window. Once make completes successfully, I type the name of the app to run it. Then, I invariably want to make changes. So back to the editor and make the changes. Save the code, then in the command window - 'arrow up' to make, and hit enter. After it has compiled and linked you hit the down arrow to get to the name of the app to run it again.

This work flow is simple, quickly becoming second nature. Once you understand a makefile you'll be far ahead of the crowd. Using a graphic IDE is nice, but you miss the details. Details which can be important to your project. Optimization is part of the compiler's work. With the profiler, the debugger, along with the compile and link messages, you gain insight into how to use the compiler's optimization system better. You can optimize for speed, or for memory usage. Optimization is not trivial. You can have the computer work on larger problems, when you write your code more efficiently; or work on the same sized problems more rapidly. Or I could just split the program across the network and let a thousand cores run my code in parallel. Ah, dreams