gcc Tutorial

What is gcc?

Gcc stands for gnu compiler collection. Gnu is a type of license for free, open source software. The majority of gnu software is for unix-based systems. A compiler is a computer program used to convert code into a file that the computer can execute. So gcc refers to a collection of Unix-based, free programs which convert source code into machine code. Our linux drives in the lab have a copy of gcc, along with several other compilers. On the Windows drives, we are using a program called mingw. Mingw stands for Minimalist Gnu for Windows. It is not gcc specifically, because it is not part of the gnu project. It is a full re-implementation of the gnu C and C++ compilers (the next section shows that gcc supports more languages) for the Windows operating system, and is used in the exact same way that gcc is used, keystroke by keystroke.

This tutorial assumes that the user understands how to work in a terminal window. In windows, the best way to open a terminal is to go to the start menu, select run, and type cmd in the resulting window. Due to the variation of Unix systems, there is no definite answer as to how to open a terminal window, but many environments will have an icon that looks like a computer screen that you can click to get a terminal window. In the lab, you will find such an icon in the lower left hand corner of the screen.

gcc Basics

- **Kinds of Code gcc Handles**

- Gcc handles C, C++, Objective C, Java and Fortran. You may see commands using 'g++' rather than gcc. If you run gcc on a .c file, gcc will run the C compiler automatically. If you run gcc on a .cc file, gcc will recognize the extension and run g++ automatically. If you for some reason have a different file extension, then g++ must be called so that the compiler knows what kind of code is in the file. This is true of all of gcc's compilers. For more information, go to the gcc project page. To be certain you are always using the correct command, use gcc for old-style C code, and use g++ when compiling C++ code. Also note that the java compiler is a rather young project, and may not handle code intended to be compiled using Sun's java compiler, javac. Gcc is most pupolar with c and c++ programmers.

- **Flags**

  - Flags are extra pieces of text that accompany a program called from the command line. For example, in Unix, 'ls' will display all the unhidden files in the current working directory. 'ls -a' will display all the files, including hidden files, in the current working directory. Flags can also have arguments accompanying them, such as 'tar -z -c directory.tgz -f directory', where -c takes a name of a file to create, and -f takes a directory of files to compress.

- **Getting Started**

  - It is good practice to use a standard extension for your filenames, and it also allows gcc to be more helpful. You should name your C files using the .c extension, such as file.c . C++ files should be named with a .cc extension, such as file.cc . This tutorial uses c++ as a language. You may use your own c++ file, or you can download source from the

links that follow. Make sure to right click on the link and choose 'save target as' to download the file. To compile the simple program helloworld.cc change to the directory the file is in, and type

> g++ helloworld.cc

where the '>' is the beginning of the command prompt. Assuming your code had no errors, you may detect a slight pause, then a second command prompt appears on the screen. What just happened? g++ just compiled your code and automatically created an executable file in the same directory. On unix systems, this default file is 'a.out', and on Windows systems, the default file is 'a.exe'. You can run your program by typing

> ./a.out

or

> a.exe

Assume for a minute that you may compile more than one program in your lifetime. You could compile the file to a.out or a.exe and then rename it to something more descriptive, but most computer scientists are obsessed with eliminating unnecessary steps, so gcc allows you to use the '-o' flag to specify the name you would like the executable to have:

> g++ helloworld.cpp -o helloworld.o

Now, instead of a.out, g++ creates an executable named helloworld.o. In Unix systems, any file can be labeled as an executable, and it is common to either use the .o extension, or

to have no extension at all. In Windows you should always use the .exe extension so that windows knows to run that file as an executable.
Another useful flag is the -Wall flag. This causes all warning messages, no matter how slight, to be printed to the screen. Warnings are different from errors, because when the compiler detects an error, no executable file will be created. A warning is more of a 'you did this, but it is not a very good idea' kind of message. Declaring a variable and never using it is an example of coding that will run without errors, but will produce a compiler warning. Warnings might provide a hint when you are trying to debug your program, so keep it in mind when compiling. You now know everything you need to know about compiling and running a simple c++ application using gcc. The next section will describe some more advanced operations.

- **Compiling More than One File into a Single Program**

  - g++ has the power to link compiled output from several files into a single executable. You can do this two different ways. You can either supply all of the source files in one compilation or you can compile them separately and then integrate them into a single executable at a later time. To supply all of the files at compile time, use this format:

    > g++ [options] file1 file2 ... fileN

    where the options can include '-o' and '-Wall'. It is important to remember that .h files should be in the same directory as the .cc files, but should not be included in the g++ command. Try this example. Download frog.cc, frog.h and main.cc into

the same directory. Then type

> g++ -Wall -o frog.exe frog.cc main.cc

Assuming there were no errors, you will find an executable named frog.exe in the same folder where you compiled. The second method, making multiple calls to gcc, has some distinct advanatages. Here is the process, using the same source files from above:

> g++ -c frog.cc
> g++ -c main.cc
> g++ frog.o main.o -o froggie.exe

This process is most useful for large projects. For our simple frog example, the extra steps don't make sense. However, imagine a large software project with 45 source-files, each one containg 2000+ lines of code. Compiling every single file into a single executable could take several minutes, or possibly hours. This is where makefiles are useful. Makefiles have the capability of checking each source file, determining if it has been changed, and calling g++ with the -c flag to compile only the files that require compilation. But even without a makefile, you could manually save yourself time by only compiling the files you know have changed. For a large project, if you are only working on a handful of files, this can reduce compilation time substantially.

- **For More Help**
  - For more help using g++, try the man page on any Unix system,

    > man g++

On the windows systems in the lab, there is no 'man' command, but the page can be found online here.

- *GCC tutorial written by Adam Anthony February 2005*