6.824 - Spring 2006

Getting started with 6.824 labs

Introduction

The 6.824 labs require you to program C/C++ on FreeBSD machines. This document should help you on your way with logging into the machines, reading manual pages, compiling C/C++ programs, and understanding `Makefiles` that we will give you.

Logging in with ssh

After you get your login and password, you can log into class machines using ssh, which is installed on Athena. Other Unix versions are available on http://www.openssh.com/. Windows versions come with Cygwin. If you just want ssh for Windows without the fancy stuff, you can also just install PuTTY. You can download NiftyTelnet SSH for MacOS at http://www.lysator.liu.se/~jonasw/freeware/niftyssh/, or just use command line ssh from the Terminal.app.

```
% ssh student@pain.lcs.mit.edu
```

will log you in as `student` on machine `pain.lcs.mit.edu`. (Don't type the ``%"!) Needless to say, you need to replace `student` with the username you receive from us when you register for lab. If there's a problem with pain.lcs.mit.edu, you can check out the list of class machines.

If you are the adventurous type, you can set up your ssh so that you can log in without having to type a password each time. (The ssh man page may also be useful here.)

Finding and reading manual pages

If you need to find information on Unix commands, system calls, configuration files, etc., then `man` is your friend. For example,

```
% man socket
```

gives you the man page for the `socket` system call. (`man man` gives you the manual page for `man` itself.) Note that it may be necessary to specify which section of the manual you wish to read. For example, `man read` may yield the man page for the wrong `read`---the shell `read` rather than the `read` system call. If this is the case, try `man 2 read`. In this case, `2` specifies the manual section; there are 8 sections. You can find relevant man pages (and what section they are in) using `man -k`. For example,

```
% man -k read
```

shows a one-line description for each man page that has the word ``read'' in the description.

Compiling and linking simple programs

A compiler takes one or more source files and produces object files that contain machine code. Object files contain machine code and usually have a `.o` extension. C source files usually have a `.c` extension and g++ files usually have a `.C` or `.cc` extension. Object files may have undefined symbols (names of functions or global variables); linking object files into an executable resolves these undefined symbols. Note that one of the object files must have a `main()` function to create an executable.

Object files can be linked together to form an executable, provided that one (and only one) of the object files defines a `main()` function.

An executable is similar to an object file in that it contains machine code, but an executable contains some additional bits that allow the operating system to start the program and run it.

Compiling a source file into the equivalent object file is done using `g++ -c`. For example:

```
% g++ -c hello.C -o hello.o
```

The `-c` flag indicates pure compilation without linking. Even if `hello.C` defines `main()`, the resulting `hello.o` is not executable. To compile a source file into an executable, omit the `-c` flag:

```
% g++ hello.C -o hello
```

This creates an executable `hello`. Note that `g++` will bark at you if you try to create an executable without a definition of `main()`.

Note that all these examples assume you are compiling g++ files. Use `gcc` instead of `g++` if you are compiling plain old C files.

Multiple object files can be linked together into one executable. Suppose we have the following files:

common.h

```
#ifndef __COMMON_H
#define __COMMON_H
void hello();
void bye();
#endif // __COMMON_H
```

hello.C

```
#include <stdio.h>
#include "common.h"
void hello() {
  printf("hello!\n");
}
```

bye.C

```
#include <stdio.h>
#include "common.h"
void bye() {
  printf("bye!\n");
}
```

main.C

```
#include "common.h"
int main() {
  hello();
  bye();
}
```

To turn these files into a working executable, do:

```
% g++ main.C hello.C bye.C -o hellobye
```

This compiles `main.C`, `bye.C`, and `hello.C` and links them together into an executable called `hellobye`. The header file `common.h` declares---but does not define---`hello()` and `bye()` so that the compiler can compile `main.C`, without knowing the details of their implementation.

Linking is only possible when no two object files try to define C or g++ functions or global variables with the same name. For example, try declaring a variable `int foo` in both `hello.C` and `bye.C`. Compiling each file separately is no problem, but linking fails with a `multiple definition of `foo'` error. The same is true for functions. Depending on what you want, you can avoid this problem in one of a number of ways:

- If you want `hello.C` and `bye.C` to **share** variable `foo`, you can declare one of them as `extern int foo`.
- If you want `hello.C` and `bye.C` to each have their own independent variable named `foo`, then declare one or both of them as `static int foo`. Keyword `static` limits the scope of the declaration to just the current file.
- The best way to solve this problem is to avoid global variables altogether; use function arguments, local variables, or g++ class members instead.

Makefiles

While doing the 6.824 labs you probably don't need to create your own Makefiles from scratch, because we will give them to you. However, you may need to modify them. This section gives a quick overview of what a Makefile is and how it is used.

A Makefile is used in conjuction with `gmake` and describes how different source and header files in a project relate and how to compile and link them. A Makefile usually bears the meaningful name `Makefile` and its contents are probably roughly similar to this example:

Makefile

```
hellobye : libhellobye.a main.o
        g++ main.o -L. -lhellobye -o hellobye

libhellobye.a : hello.o bye.o
        ar cru libhellobye.a hello.o bye.o
# this command puts hello.o and bye.o in a library libhellobye.a

hello.o : hello.C common.h
        g++ -c hello.C -o hello.o

bye.o : bye.C common.h
        g++ -c bye.C -o bye.o

main.o : main.C common.h
        g++ -c main.C -o main.o

clean :
        rm -f *.a *.o hellobye
```

This Makefile has 6 sections. Each section describes a **dependency** on the first line and some **action** on the second. A dependency consists of a target file (before the colon) and prerequisite files (after the colon). Lines starting with # are comments.

When you run `gmake`, it will execute a target's action if any of the prerequisite files have been modified since the target was last modified. `gmake` executes the action in much the same way as if you had typed it to the shell. `gmake` only tries to make sure that the very first target in the `Makefile` is up to date. However, if the first target depends on other targets, `gmake` may end up executing multiple actions.

The first section in the Makefile above states that the file `hellobye` depends on `libhellobye.a` and `main.o`. If either `libhellobye.a` or `main.o` is newer than `hellobye`, then `hellobye` is rebuilt. Building the whole project is now as simple as typing

```
% gmake
```

If you write your Makefile correctly, gmake will only compile and link those parts of a project that have changed since the last `gmake`. If your project is big and the full compile/link process takes a long time, then gmake is your friend. It saves time.

Notice that the last section in this Makefile is not a real dependency, but allows the user to conveniently call

```
% gmake clean
```

to remove all object files, the library, and the executable.

Makefiles can get pretty complicated, but this gentle introduction may very well suffice for your 6.824 needs. If you're interested, read the full gmake manual (`info make`). It's fabulous.

Version Control

You may find it useful to keep track of different versions of your work as you progress. A version control system allows you to checkpoint your work and may help you track down regressions in your code. The change log tracked by the VCS can serve to remind you what your intention was in making a particular change days or weeks afterwards.

The lab machines have several popular version control systems available including CVS, Subversion, and darcs. If you are familiar with one of these systems, you are welcome to use it. If you are interested in learning one, you ought to be able to get started relatively quickly using darcs. There is copious documentation for each of these tools available on the web.