UNBC Laboratory Assignment
Computer Science 101—Winter 2007

Separate compilation & "make"

Purpose:

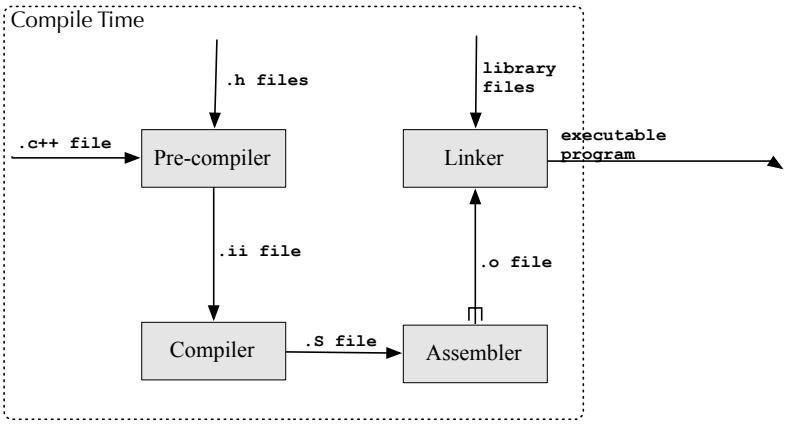
To learn how to use `g++` and make to efficiently maintain multi-file programs.

Due Date:

The questions at the end of this lab are due at the beginning of class Monday, 15 January.

How C/C++ compilers work:

Although we often think of `g++` as "the compiler" for C++ programs, it is in fact just a front-end that runs several different programs corresponding to the different phases of compiling a program. These phases are:



1. Pre-processing,
2. Compilation proper,
3. Assembly, and
4. Linking.

Pre-processing

This is the phase that replaces "`#include <file.h>`" lines by the contents of the appropriate file, replaces pre- processor macro calls by the expanded macro call, etc. Pre- processors are evil, and in C++ you usually only need to use the pre-processor to handle "`#include`" statements. The pre-processor takes files that begin with `.c` or `.cpp` and produces output files with extensions like `.i` or `.ii`. Normally these output files are suppressed and compilation happens at the same time.

After inclusion of `#include` files the resulting piece of text is called a *compilation units*. Many rules, for instance "do not define a static function twice", apply to compilation units.

Compilation and Assembly

These two phases together take pre-processed input and convert it into a format that is almost ready to be executed, more formally, into object files. On UNIX systems (including Mac OS X and Linux) object files typically have `.o` as their extension; on most other operating systems (including Vax VMS and Microsoft Windows) the extension is usually `.obj`.

Object files consist of three parts: machine language, a dictionary of symbols that this object file defines, and a dictionary of symbols that this object file requires from elsewhere. For instance, if you compile your `main` function into a `.o` file then the dictionary of definitions will contain the name `main`, and the dictionary of required symbols will most likely contain `std::cout`.

Usually when you run `g++` each `.c` or `.cpp` file is pre- processed, compiled, and assembled into a `.o` (object) file.

Linking

This is the final phase of producing a program. In this stage, several object files and libraries (which are essentially collections of object files) are combined together to create one executable file. This phase compares references and definitions and tries to match them up. It is this stage that produces error messages if you are missing functions, or have compiled them twice.

Running

When you run your program, it is copied from permanent storage into main memory and the operating system arranges for keyboard input to appear as `std::cin` and `std::cout` to be copied to your shell window, and so on. Sometimes further linking (so called "dynamic linking") occurs at this time. This has the disadvantage of slowing the start of your program, but the advantage of making your program files smaller (becuase they no longer require the contents of the dynamically linked files). It also has the advantage that multiple running programs that use the same libraries can sometimes share their copies in memory. Whether or not all of the linking occurs at compile-time can be controlled by options that you pass to the compiler.

File creation and deletion

Normally `g++` keeps only the files created by the last stage that it ran. For instance, if you run the pre- compiler and the compiler, but not the assembler or linker, then `g++` would keep the output from the compiler, but discard the output from the pre-compiler after it had been passed to the compiler.

Using the compiler efficiently:

There are two observations that allow us to use the compiler efficiently. One is that usually

between compilations only a few `.cpp` and `.h` files change. The second is that relatively speaking, linking is much faster than compiling and assembling. The general idea then, is to only run the pre-processor, compiler, and assembler on source code (`.cpp`) files that have changed since the last compilation. Instead of having the compiler produce a "a.out" file from several source files, you produce one object file for each source file, then link together all of the object files that you have created.

Halting the compiler after a particular phase

From now on, instructions are specific to `g++`. Most other compilers will have similar flags and options, but you need to consult their documentation for the details. !

The default behavior of the compiler is to try to run as many phases of the compiler as makes sense. To have the compiler halt after a particular phase, you can use one of the following flags:

`-c`	Compile only. Halt after the assembly stage, and produce an object file, but do not link. This is a commonly used option.
`-S`	Stop after compilation proper; do not run the assembler. Produce a `.s` file.
`-E`	Only run the pre-processor. Produce a `.i` or `.ii` file

Other useful general options are:

`-v`	Print the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.
`-o`	Rename the output file.

Renaming the output

The default name for the file produced by the compiler depends on what phase it stops at. If the compiler stops before the linker phase the name is constructed from the base name of the input file followed by the appropriate ending from `.i`, `.ii`, `.S`, and `.o`. The default output from the linker is named `a.out`. The -o flag changes the default output name.

Warning Be very careful about how you use the `-o` flag! If you accidentally write "`g++ ... -o VeryImportant.c++`" you will be in for a nasty surprise. !

An example

Suppose that we have three source files: `blackjack.cpp`, `cards.cpp`, and `strategy.cpp` (and associated `.h` files.) Also, suppose that `blackjack.cpp` contains the main routine, and that we want to call the resulting program blackjack. One way to accomplish this is to type

```
g++ blackjack.cpp cards.cpp strategy.cpp
mv a.out blackjack
```

These can be combined on one line as

```
g++ -o blackjack blackjack.cpp cards.cpp strategy.cpp
```

However both of these methods eliminate the intermediate `.o` files. If we initially take a little bit longer by using

```
g++ -c blackjack.cpp
g++ -c cards.cpp
g++ -c strategy.cpp
g++ -o blackjack blackjack.o cards.o strategy.o
```

we can then make changes to one file and re-compile the program much faster. Suppose, for instance that we change `strategy.cpp`. To re-compile the program we only need to update the `strategy.o` file and re-link:

```
g++ -c strategy.cpp
g++ -o blackjack blackjack.o cards.o strategy.o
```

The advantage of doing things the second way is that compilation is much faster. The disadvantage is that re- compilation takes some thought.

For very large projects it is completely infeasible to recompile everything every time that a change is made and tested, and almost as infeasible to keep track of which files have been recompiled recently. Consequently, large projects require some kind of tool that automates the process of generating the code. Later in this lab, we look at one of the oldest such tools: `make` and Makefile's.

Laboratory Exercise 1

⇒ Write a program to solve the following problem (slightly modified from *Deitel & Deitel*)

> A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time.
>
> Write a program that will calculate and print the charges for each of the times stored in a file called "`times.txt`". Your output should be in a neat tabular format, and your program should compute the total time and charges. Your output should appear in the following format:
>
> ```
>    Car     Hours     Charges
>      1       1.5        2.00
>      2       4.0        2.50
>      3      24.0       10.00
>  TOTAL      29.5       14.50
> ```
>
> Assume that the times in "`times.txt`" are floating point numbers separated by spaces.

Although it may be overkill for this assignment, write your program as at least three separate functions: "`main`", "`void processFile(istream&, ostream&)`", and "`double calculateCharge(double)`", each with their own `.h` and `.cpp` file. In your script file, show that you know how to compile the files separately.

The `.h` file for `processFile` should look something like

processFile.h

```
//  File:                       processFile.h
//  Created by:                 David Casperson <casper@lib-444.fac.unbc.ca>
//  Created:                    Thu Jan  9 11:21:11 2003
//  Lab assignment:             1
//  Lab due date:               Friday, January 17 2003
//  Description:                interface for function to process
//                              parking garage entries.

#if !defined(processFile_INCLUDED)

void processFile(std::istream& data_in, std::ostream& report) ;

#define processFile_INCLUDED
#endif  // !defined(processFile_INCLUDED)
```

Laboratory Exercise 2

⇒ Create a new "`main`" function so that the user of the program can specify the name of the input file on the command line. Do not change any of your pre-existing files. Instead, create a second file that contains your new `main` function. Compile it separately, and link it to the appropriate previously existing `.o` files. In your script file make sure to show how you re-compiled your program.

Technical Note

To read command line arguments, you need to change the signature of main from `int main(void);` to `int main(int argc, char* argv[]);` With this signature `argv` will contain an array of C-style strings, where `argv[0]` will be the name of your program, `argv[1]` will the second "word", on the command-line, and so on. "`argc`" is the number of words on the command-line, and `argv[argc]` is guaranteed to be a null pointer.

Ask your lab instructor if you are confused as to how to get a file name from the command line.

What Makefiles are:

Each phase of the compilation process depends on the previous one. Similarly, making (compiling) certain files depends on other files having been made. Doing things the second way above, the file `blackjack` depends on the files `blackjack.o` `cards.o` and `strategy.o`; and the file `cards.o`

Figure 1: Useful flags for `Makefile`s.

`-f`	`-f filename`. Use `filename` instead of `makefile` or `Makefile`.
`-k`	Continue after errors. Normally the make process stops as soon as an error is detected. With this option make will continue if it is reasonable to do so.
`-n`	This flag causes make to print out the list of commands it would execute to update the target, but doesn't actually execute any commands. Useful for debugging makefiles.
`-t`	Touch. Rather than execute the commands specified in a makefile to update targets, make just changes their time-stamps.

depends on the file `cards.cpp`. We can tell whether the blackjack file needs updating by comparing the time at which it was created against the creation times of the files `blackjack.o` `cards.o` and `strategy.o`. If any of the latter were created more recently than the blackjack file, then presumably the blackjack file needs to be updated. A makefile is a file used by the make program. It contains a list of dependencies and rules. For our example, it would contain the two lines

```
blackjack : blackjack.o cards.o strategy.o
     g++ -o blackjack blackjack.o cards.o strategy.o
```

as well as others. The first line specifies which line depends on which other. The second line specifies the commands to run in order to do the update should the dependent program need updating. Command lines like the second line shown above must begin with a TAB (Ctl-i, or ASCII 0x9) character. Some text editors will automatically substitute a string of spaces for a tab character. If your text editor does this you need to figure out how to over- ride this behavior when you are creating a makefile. Makefiles can become extremely complicated; and are often difficult to create correctly. The simplest strategy is often to have one standard makefile which you then modify slightly to work for a particular program or set of programs.

Makefiles are almost always named either `makefile` or `Makefile`; although most "`make`" programs let you choose a makefile with a different name.

Makefiles are complicated to create; but extremely easy to use. You simply type "`make target`", where `target` is the name of the file that you want the computer to create, and it automatically checks all of the appropriate dependencies and runs the commands necessary to ensure that the target is up to date.

Most make programs accept a bewildering variety of command- line flags, and the meaning of these flags often depends to some extent on the operating system being used. Some of the more common and useful ones are shown in Figure 1.

Laboratory Exercise 3

⇒ Create a makefile for the program in Exercise 2, and use it to compile your program. It should look something like the makefile shown in Figure 2. Be sure to include the answers to the following questions with your assignment.

Figure 2: Makefile

```
1  #This␣line␣is␣a␣comment.
2  #The␣following␣two␣lines␣are␣abbreviations.
3  GPP␣=␣/opt/sfw/gcc-3/bin/g++
4  CPPFLAGS␣=␣-g2
5
6  process␣␣␣:␣main.o␣processFile.o␣calculateCharge.o
7      →$(GPP)␣$(CPPFLAGS)␣-o␣process␣main.o␣processFile.o␣calculateCharge.o
8
9  %.o:␣%.cpp
10     →$(GPP)␣$(CPPFLAGS)␣-c␣$<
11
12 clean␣:
13     →-rm␣*.o
```

Note that in the following listing, spaces are shown as "␣", and tabs are shown as "→". Tab characters must be typed as shown.
Also note that if you compile with `/opt/sfw/gcc-3/bin/g++`, you need to ensure that your `$LD_LIBRARY_PATH` contains `/opt/sfw//opt/sfw/gcc-3/lib`!.

⇒ Modify `calculateCharge.cpp` by adding a comment to the beginning of the file. Re-execute make. What commands are executed?

⇒ What do you think `make clean` will do? Try it out. What commands are executed?

⇒ After `make clean`, what do you think

```
make GPP=/csd3/local/GCC-2.95.2/bin/g++ process
```

will do? Try it out. What commands are executed?