Introduction to UNIX Makefiles and compiling programs with GCC

Marvin Smith

September 8, 2011
Why use makefiles?
Why use makefiles?

- easier than building each component individually
- you can ensure that all binaries are built with up-to-date code
General Composition of a Makefile

...Example of a Rule...

TARGETS: PREREQUISITES
RECIPE

...Sample...
a.out: main.cpp
   g++ main.cpp

What does this mean?

   TARGETS: what you want to build
   PREREQUISITES: files required to execute
   RECIPE: commands required to execute rule
Sample Application

Function 1 - Append1
```
// Function 1

#ifndef __FUNCTION1_H__
#include <string>
#include "function2.h"
#endif

using namespace std;

string function1()
{
    return (string("Mary had ") + function2());
}
```

Main Driver
```
// Main Driver

#include <iostream>
#include <string>
#include "function1.h"

using namespace std;

int main()
{
    string str = function1();
    cout << str << endl;
    return 0;
}
```

2 - Append2
```
// Function 2

#ifndef __FUNCTION2_H__
#include <string>
#endif

using namespace std;

string function2()
{
    return string("a little lamb");
}
```
Some Notes

- Prerequisites compare the timestamps of the listed files against the target. If they are newer than the target, the rule is executed automatically.

- Targets should always be the name of the file you wish to create. This is how the makefile determines dependencies and resolves them. Exceptions are for commands like `clean`, `check`, `all`, etc.

- When making the recipe, ensure that the recipe uses a TAB! Spaces are not the same and the makefile will fail...
Simple Makefile

```
# Simple Brute-Force Makefile
# - Author: Marvin Smith
#
# NOTE: When building makefiles, remember the
# construction of rules...
#
# <thing to build>: <what it needs to build it>
# <tab> <command>
#
# Build main executable
all: mary

mary: main.o function1.o function2.o
    g++ main.o function1.o function2.o -o mary -g -Wall

# Build main driver source
main.o: main.cpp function1.o
    g++ -c main.cpp -g -Wall

# Build func1 source
function1.o: function1.cpp function2.o
    g++ -c function1.cpp -g -Wall

# Build func2 source
function2.o: function2.cpp
    g++ -c function2.cpp -g -Wall

# for cleanup
clean:
    rm *.o mary
```

Notice that each object is built individually and the parameters for identical builds are required in each line.
Macros

macro_name = some_value

- example: CC = g++
- example: CFLAGS = -Wall -g
- example: SOURCES = main.cpp function1.h function2.h

NOTE: macro names should be in capital letters for convention
Simple Makefile with Macros

Notice now that we have replaced the compile options with a Macro variable. You can do this with files, items, flags, etc.
Automatic Macros

Automatic macros are predefined macros which make sorting through lists.
Advanced Makefile with Automatic Macros

Using Automatic Macros, we can now skip listing each item separately and focus on just placing our files in the source list. That way, our building will be less prone to errors.
Useful G++ flags

- `-g`: compile library with GDB flags, essential if you want to use debugger
- `-Wall`: compile with all warning flags. Better type checking and more warnings for things that are allowed but not recommended.
- `-O1`: compile code with optimizations that don't reduce compile speed
- `-O2`: compile with all optimizations which don't affect binary size, like unrolling loops.
- `-O3`: compile with all supported optimizations
- `-Os`: optimize code for binary size
- `-c`: compile object files. Requires the .cpp files and headers of other libraries (no other objects)
- `-o`: allows compiler to build outputs with name other than a.out.
CS 302 Project Requirements

▶ All code must be able to compile on GCC for Unix machines. You can use Windows for personal use and for your demo, but I need to be able to compile it on my own.

▶ All projects must include a makefile. I should be able to just type make and everything works. Organization of your project is up to you.

▶ If certain functions don’t work in your program, it **MUST** be annotated in your report. Your grade will severely suffer otherwise.

▶ All code must be turned in using either zip or tar compression. Just right click on any operating system and compress. Don’t email me your files as separate attachments.

▶ Don’t worry. If I have problems building and running your code, I will let you know and usually let you fix it.