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

Main Driver

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

using namespace std;

int main(){

    string str = function1();
    cout << str << endl;

     return 0;
}
```

Function 1 - Append1

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

using namespace std;

string function1();

#endif
```

```
#include "function1.h"

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

2 - Append2

```
#ifndef __FUNCTION2_H__
#include <string>

using namespace std;

string function2();

#endif
```

```
#include "function2.h"

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. Exeptions 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

```
1  #   Simple Makefile with Macros
2  #   - Author: Marvin Smith
3  #
4  #  NOTE: When building makefiles, remember the
5  #        construction of rules...
6  #
7  #  Using macros we can simplify our efforts by removing
8  #  redundant code
9  #
10 # NOTE:  MACRO_NAME = whatever the macro is
11
12 CFLAGS = -g -Wall
13
14 all: mary
15
16 mary: main.o function1.o function2.o
17    g++ main.o function1.o function2.o -o mary $(CFLAGS)
18
19 # Build main driver source
20 main.o: main.cpp function1.o
21    g++ -c main.cpp $(CFLAGS)
22
23 # Build func1 source
24 function1.o: function1.cpp function2.o
25    g++ -c function1.cpp $(CFLAGS)
26
27 # Build func2 source
28 function2.o: function2.cpp
29    g++ -c function2.cpp $(CFLAGS)
30
31 # for cleanup
32 clean:
33    rm *.o mary
```

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

```
1  #   Solid Makefile with Auto Macros
2  #   - Author: Marvin Smith
3  #
4  #   Using Automatic Variables with Inferred Rules,
5  #   we can avoid entering every element of the makefile
6  #   and instead feed a list of sources.
7  #
8
9  # Some compiler options
10 CC       = g++       # select your compiler
11 CFLAGS   = -g -Wall  # select compile options
12
13 # List of sources
14 SOURCES = main.cpp function1.cpp function2.cpp
15 EXEC    = mary
16
17 # convert to a list of objects
18 OBJECTS = $(SOURCES:.cpp=.o)
19
20 all: $(EXEC)
21
22 # build executable
23 $(EXEC): $(OBJECTS)
24    $(CC) $(OBJECTS) -o $(EXEC) $(CFLAGS)
25
26 # Build source files
27 %.o: %.c
28    $(CC) -o $< -c $@
29
30 clean:
31    rm *.o $(EXEC)
32
```

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