

Compiling and Makefiles

2501ICT Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2009

Outline

- 1 Compiling C Programs
- 2 Makefiles
 - Using the `make` Utility
 - Makefiles for Objective-C Code
 - Makefiles for C++ Code

Compiling C Programs

- Integrated Development Environment (IDE)
 - Eclipse, XCode, Visual C++, Project Center, ...
 - Compiles programs at the press of a button (like BlueJ)
 - Often difficult to customise
 - Very rarely support multiple platforms and languages
- Command Line
 - Requires manual invocation
 - Requires knowledge of command line parameters
 - Can be tedious for large projects
 - Cross-platform and -language compilers (e.g. `gcc`)
- Makefiles
 - Combine the best of both worlds
 - Recompile a complex project with a simple `make` command

Getting a Command Line Interface

- Via Dwarf
 - using putty (Windows)
 - `ssh dwarf.cit.griffith.edu.au`
 - Via a local Terminal
 - Linux: e.g. through the Gnome program menu
 - Mac OS X: e.g. Applications / Utilities / Terminal.app
 - Windows: e.g. Start / Programs / Programming Tools / GNUstep / Shell
- ⇒ Enter commands to compile your program
- Hit *Return* (or *Enter*) after every command!

Compiling a C program using gcc

- Once on the command line change to the directory (folder) your program is in
 - `cd /my/example/directory`
- Compile the source code (e.g. `Hello.c`)
 - `gcc Hello.c`
 - Compiles `Hello.c` into an executable called `a.out` (or `a.exe` on Windows)
- `gcc -o Hello Hello.c`
 - Compiles `Hello.c` into an executable called `Hello`
 - On Windows always use `Hello.exe` instead of just `Hello`
- `gcc -Wall -o Hello Hello.c`
 - Prints all warnings about possible problems
 - Always use `-Wall` when compiling your programs!
- `./Hello`
 - Run the `Hello` command from the current directory

Makefiles

- Save compile time
 - only recompile what is necessary
- Help avoiding mistakes
 - prevent outdated modules from being linked together
- Language independent
 - work with any programming language
 - C, C++, Objective-C, Java, ...

How do Makefiles work?

Example (A simple Makefile)

```
Hello: Hello.c  
    gcc -Wall -o Hello Hello.c
```

- First Line: Dependency Tree
 - Target and Sources
 - Target: the module to be built (e.g. `Hello`)
 - Sources: pre-requisites (e.g. `Hello.c`)

Make Rules

Example (A simple Makefile)

```
Hello: Hello.c
    gcc -Wall -o Hello Hello.c
```

- Second Line: Make rule
 - command to execute
 - `gcc -Wall -o Hello Hello.c`
 - requires a **tab** character (not spaces) for indentation

Multiple Targets

Example (Makefile for compiling multiple Modules)

```
Program: module1.o module2.o
    gcc -o Program module1.o module2.o

module1.o: module1.c
    gcc -c -Wall -o module1.o module1.c

module2.o: module2.c module2.h
    gcc -c -Wall -o module2.o module2.c
```

- **Default Target: first target** (`Program`)
 - link two object files (`module1.o` and `module2.o`) into one program (`Program`)

Multiple Targets (2)

Example (Makefile for compiling multiple Modules)

```
Program: module1.o module2.o
    gcc -o Program module1.o module2.o

module1.o: module1.c
    gcc -c -Wall -o module1.o module1.c

module2.o: module2.c module2.h
    gcc -c -Wall -o module2.o module2.c
```

- **Second Target:** `module1.o`
 - rule to compile object file `module1.o` from `module1.c`
 - `gcc -c` compiles a single module (not a full executable)

Multiple Targets (3)

Example (Makefile for compiling multiple Modules)

```
Program: module1.o module2.o
    gcc -o Program module1.o module2.o

module1.o: module1.c
    gcc -c -Wall -o module1.o module1.c

module2.o: module2.c module2.h
    gcc -c -Wall -o module2.o module2.c
```

- **Third Target:** `module2.o`
 - **compile** `module2.o` from source `module2.c`
 - **also depends on** `module2.h` (header file)

Multiple Programs

Example (Makefile for compiling multiple Programs)

```
all: Program1 Program2

Program1: module1.o
    gcc -o Program module1.o

Program2: module2.o module3.o
    gcc -o Program module2.o module3.o

module1.o: module1.c
    gcc -c -Wall -o module1.o module1.c

module2.o: module2.c module2.h
    gcc -c -Wall -o module2.o module2.c

module3.o: module3.c module3.h
    gcc -c -Wall -o module3.o module3.c
```

- 'all' target:
 - compiles all programs (Program1 and Program2)
 - does not have any compiler commands itself!

Generic Rules

- Save lots of typing
 - avoid repeating the same compiler call over and over again
- Help with consistency
 - what if you want to change the compiler invocation?
- Simply list suffixes to convert from one file type to another
 - e.g. `.c.o` to compile a `.c` to a `.o` file

Generic Rule Example

Example (Makefile containing a generic rule)

```
.c.o:  
    gcc -c -Wall -o $*.o $*.c  
  
Program: module1.o module2.o  
    gcc -o Program module1.o module2.o  
  
module2.o: module2.c module2.h
```

- `.c.o:`
 - how to compile a `.c` into a `.o` file
 - `$*` gets replaced by the file name (without extension)

Generic Rule Example (2)

Example (Makefile containing a generic rule)

```
.c.o:  
    gcc -c -Wall -o $*.o $*.c  
  
Program: module1.o module2.o  
    gcc -o Program module1.o module2.o  
  
module2.o: module2.c module2.h
```

- No need for a `module1.o`: rule!
 - compiler already knows how to compile `.c` into `.o`
 - But: `module2.o` needs a rule (also depends on `.h`)

Generic Rules for Languages other than C

- The `make` utility by default only knows about C
 - “what if I want to compile a different language?”
- Suffixes can be specified
 - using the `.SUFFIXES :` command, e.g.:
 - `.SUFFIXES: .o .m`
 - “a `.o` file can also be compiled from a `.m` (Objective-C) file”

Make Variables

- Allow more flexible make files
 - “what if the compiler is not called `gcc`?”
- Variables allow assigning a value, e.g.:
 - `CC=gcc`
- Variables can be used using `$` (*variable*), e.g.:
 - `$(CC) -c -Wall -o $*.o $*.c`
 - will replace `$(CC)` with `gcc`

Mixed Makefile Example: C++

Example (Makefile for a mixed C/C++ program)

```
#  
# A mixed makefile example for C and C++  
#  
CC=gcc  
CPLUS=g++  
  
.SUFFIXES: .o .c  
.SUFFIXES: .o .cc  
  
.C.o:  
    $(CC) -c -Wall -o $*.o $*.c  
  
.CC.o:  
    $(CPLUS) -c -Wall -o $*.o $*.cc  
  
Program: cmodule.o cppmodule.o  
    $(CPLUS) -o Program cmodule.o cppmodule.o  
  
cppmodule.o: cppmodule.cc cppmodule.h
```

Mixed Makefile Example: Objective-C

Example (Makefile for a mixed C/Objective-C program)

```
#  
# A mixed makefile example for C and Objective-C on Mac OS X  
#  
CC=gcc  
  
.SUFFIXES: .o .c  
.SUFFIXES: .o .m  
  
.c.o:  
    $(CC) -c -Wall -o $*.o $*.c  
  
.m.o:  
    $(CC) -c -Wall -o $*.o $*.m -framework Foundation  
  
Program: cmodule.o objcmodule.o  
    $(CC) -o Program cmodule.o objcmodule.o -framework Foundation  
  
objcmodule.o: objcmodule.m objcmodule.h
```

Compiling Objective-C

- The GNU C compiler knows Objective-C
 - `gcc -c -Wall -o file.o file.m`
 - Linking is more complex, requires:
 - standard Objective-C runtime: `libobjc`
 - standard OpenStep API: `libFoundation` and `libAppKit`
 - Different API setups have different locations
 - flags for `gcc` vary, depending on where to find libraries
- ⇒ Standardised ways of accessing API
- `-framework` on Mac OS X
 - `GNUmakefile` framework for GNUstep (Linux, Windows, ...)

Mac OS X Makefile Example for Objective-C

Example (Mac OS X Makefile for an Objective-C program)

```
#
# A Mac OS X Makefile example for Objective-C and the Foundation framework
#
# -- this assumes a main() module ObjcMain.m and a class ObjcModule.m
# -- (the class comes with a corresponding ObjcModule.h)
#
CC=gcc

.SUFFIXES: .o .m

.m.o:
    $(CC) -c -std=c99 -Wall -o $*.o $*.m -framework Foundation

Program: ObjcMain.o ObjcModule.o
    $(CC) -o Program ObjcMain.o ObjcModule.o -framework Foundation

ObjcModule.o: ObjcModule.m ObjcModule.h
```

GNUstep Makefiles

- GNUstep Makefiles have all the rules already pre-defined
 - `GNUmakefile`
 - the name of the main makefile (rather than `Makefile`)
 - `common.make`
 - common rules to be included in all `GNUmakefiles`
 - `tool.make`
 - pre-defined rules for command line utilities
 - set `TOOL_NAME` to be the command name
 - `program_OBJC_FILES`
 - the Objective-C files needed to compile *program*
 - `ADDITIONAL_CPPFLAGS`
 - set to `-Wall -Wno-import`

GNUmakefile Example for Objective-C

Example (GNUmakefile)

```
#  
# A simple GNUmakefile example for an Objective-C command line utility  
#  
include $(GNUSTEP_MAKEFILES)/common.make  
  
# Build a simple Objective-C program, called Example  
TOOL_NAME = Example  
  
# The Objective-C Implementation files to compile  
Example_OBJC_FILES = Main.m Some_Class.m Other_Class.m  
  
# Class Header (Interface) files  
Example_HEADER_FILES = Some_Class.h Other_Class.h  
  
# Define the compiler flags  
ADDITIONAL_CPPFLAGS = -Wall -Wno-import  
  
# Include the rules for making Objective-C command line tools  
include $(GNUSTEP_MAKEFILES)/tool.make
```

GNUmakefile Example without Comments

Example (GNUmakefile after removing the Comments)

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Example

Example_OBJC_FILES = Main.m Some_Class.m Other_Class.m
Example_HEADER_FILES = Some_Class.h Other_Class.h

ADDITIONAL_CPPFLAGS = -Wall -Wno-import

include $(GNUSTEP_MAKEFILES)/tool.make
```

AutoGSDoc in GNUmakefiles

- `autogsdoc` extracts comments starting with `/**`
 - Can be automated in a GNUmakefile
 - `document.make`
 - pre-defined rules for `autogsdoc`
 - `DOCUMENT_NAME`
 - variable containing the name of the documentation
 - `Document_AGSDOC_FILES`
 - lists the source files to scan for documentation
- Only works for C and Objective-C (not C++)

GNUmakefile with Documentation

Example (GNUmakefile plus autogsdoc)

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = Example
Example_OBJC_FILES = Main.m Some_Class.m Other_Class.m
Example_HEADER_FILES = Some_Class.h Other_Class.h

DOCUMENT_NAME = Documentation
Documentation_AGSDOC_FILES = Some_Class.h Other_Class.m

ADDITIONAL_CPPFLAGS = -Wall -Wno-import

include $(GNUSTEP_MAKEFILES)/tool.make
include $(GNUSTEP_MAKEFILES)/documentation.make
```

Compiling C++

- The GNU C compiler frontend `g++` knows C++
 - `g++ -c -Wall -o file.o file.cc`
- Linking also works with `g++`
 - standard C++ runtime `libstdc++` is automatically linked
- Different add-on API setups have different locations
 - STL, `libqt`, etc.

Example Makefile for C++

Example (C++ example Makefile)

```
#
# An example Makefile for C++
#
# -- this Makefile is for a project containing a CppMain with main() and
# -- a CppModule.cc and CppModule.h class header and implementation file
#
CPLUS=g++

.SUFFIXES: .o .cc

.cc.o:
    $(CPLUS) -c -Wall -o $*.o $*.cc

Program: CppMain.o CppModule.o
    $(CPLUS) -o Program CppMain.o CppModule.o

CppModule.o: CppModule.cc CppModule.h
```

Doxygen in Makefiles

- `doxygen` extracts comments starting with `/**`
- Can be automated in a `Makefile`
 - add a `doc` target
- Needs a configuration file (`Doxyfile`)
 - manually run `doxygen -g`
 - `cvs add Doxyfile`
- The default `Doxyfile` is not very useful!
 - edit `Doxyfile`
 - fill in `PROJECT_NAME`
 - set `JAVADOC_AUTOBRIEF` to `YES`
 - set `EXTRACT_ALL` to `YES`

Example Makefile with Doxygen

Example (C++ example Makefile)

```
#  
# An example Makefile for C++ with a 'doc' target  
#  
#  
CPLUS=g++  
  
.SUFFIXES: .o .cc  
  
.cc.o:  
    $(CPLUS) -c -Wall -o $*.o $*.cc  
  
all: Program  
  
Program: CppMain.o CppModule.o  
    $(CPLUS) -o Program CppMain.o CppModule.o  
  
CppModule.o: CppModule.cc CppModule.h  
  
doc: CppMain.cc CppModule.cc CppModule.h  
    doxygen Doxyfile
```