Working with Compilers and Makefiles

Richard Rollins

github.com/rprollins/jacs-compilers

Overview

- What is a compiler?
- A simple example
- Preprocessing, Compiling, Assembling and Linking
- Headers, Modules and Libraries
- GNU Make

Working through the examples

To work through the examples alongside the slides you must first clone the repository and create a few necessary directories from the command line:

```
git clone git@github.com:rprollins/jacs-compilers.git
cd jacs-compilers
mkdir bin build lib modules
```

What is a Compiler?

- A program that parses code (front end) and produces an equivalent set of machine instructions (back end)
- Interpreted languages do this conversion at run time
- Can create executable binaries or libraries

Some Common Compilers

- GNU Compiler Collection (GCC: gcc, g++, gfortran)
- Intel Compilers (icc, icpc, ifort)
- LLVM / Clang (clang, clang++)
- Cray Compilers (cc, cpp, fc)

Examples 1 & 2 : `return 0;`

Compiling a single file

Consider simple examples in C and Fortran:

```c
// src/example1.c
int main() { return 0; }
```

```fortran
! src/example2.f95
program main
end
```

We can trivially compile these examples to executable binaries:

```
gcc src/example1.c
gfortran src/example2.f95
```

Preprocessing, Compiling, Assembling and Linking

"Compiling" a program is actually a four-stage process:

- First the preprocessor handles directives such as ***#include***, ***#define*** and ***#pragma*** in the source code.
- Next the compiler translates the source code into assembly
- Then the assembler translates the assembly into machine instructions and stores them in object (.o) files
- The linker combines objects to produce binaries and libraries

We can explicitly separate linking from the other three steps:

```
gcc -o build/example1.o -c src/example1.c
gcc -o bin/example1 build/example1.o
```

Example 3 : C/C++ Headers

C/C++ Header Files

In C/C++ function declarations and definitions are separable:

- Function declarations should be given in a header file:

```c
// include/example_func.h
int int_sum (int arg1, int arg2);
```

- We **#include** function declarations via headers at compilation

```c
// src/example3.c
#include <stdio.h>
#include "example_func.h"
int main() { printf("%d\n",int_sum(23,72)); return 0; }
```

- Function definitions are not needed until link time and can be given in separate source files:

```c
// src/example_func.c
int int_sum (int arg1, int arg2) { return arg1 + arg2; }
```

C/C++ Header Files

- The path to the header is given at compile time using `-I`:

```
gcc -I/path/to/jacs-compilers/include \
  -o build/example3.o -c src/example3.c
```

- The function definition is compiled separately:

```
gcc -o build/example_func.o -c src/example_func.c
```

- The two object files can then be linked into an executable:

```
gcc -o bin/example3 build/example3.o build/example_func.o
```

Example 4 : Fortran Modules

Fortran Modules

Fortran can define modules containing variables and functions:

```fortran
! src/example_mod.f95
module example_mod
  public :: int_sum
contains
  function int_sum(i,j) result(k)
    integer :: i,j,k
    k = i + j
  end function int_sum
end module example_mod
```

A module (.mod) file is created at compile time. You can specify the module output path with `-J` (gfortran) or `-module` (ifort):

```
gfortran -J/path/to/jacs-compilers/modules \
  -o build/example_mod.o -c src/example_mod.f95
```

Fortran Modules

- We **use** modules in other code to access their contents:

```
! src/example4.f95
program main
  use example_mod
  print *, int_sum(42,43)
end
```

- The path to the module is given at compile time using `-I`:

```
gfortran -I/path/to/jacs-compilers/modules \
  -o build/example4.o -c src/example4.f95
```

- Then link both objects:

```
gfortran -o bin/example4 \
  build/example4.o build/example_mod.o
```

C/C++/Fortran Libraries

Object files can also be compiled to libraries:

```
# Static Library
gcc -o build/example_func.o -c src/example_func.c
ar rcs lib/libexample.a build/example_func.o
```

```
# Shared Library
gcc -fPIC -shared -o lib/libexample.so src/example_func.c
```

To link against a library, we pass its name and path at link time using `-l` and `-L` repectively:

```
gcc -L/path/to/jacs-compilers/lib -lexample \
  -o bin/example3 build/example3.o
```

LD_LIBRARY_PATH vs rpath

- Binaries linked against shared libraries need to be able to find them again at runtime.
- The environment variable `LD_LIBRARY_PATH` prepends to both the link path and runtime path. This can be unsafe and it should not be set globally.
- Preferably, add any non-standard library paths to the runtime search path of an executable at link time:

```
# Linux
gcc -L/path/to/lib -Wl,-rpath=/path/to/lib -lfoo main.o
```

GNU Make and Makefiles

GNU Make and Makefiles

Make is a build automation tool used to define set of rules for how to construct ***targets*** from their ***dependencies*** in scripts called Makefiles:

```
# Makefile
target : dependencies
  commands
```

Which is invoked with:

```
make target
```

Automatic Variables And Pattern Rules

- `$@` : The filename representing the target.
- `$<` : The filename of the first prerequisite.
- `$^` : The filenames of all prerequisites (excluding duplicates).
- `$?` : The filenames of all prerequisites newer than the target.

Standard Variables

CC, ***CXX***, ***FC***, ***LD***

- C, C++ and Fortran compilers and the linker respectively

CPPFLAGS

- Preprocessor flags e.g. include paths `-I/path/to/include`

CFLAGS, ***CXXFLAGS***, ***FCFLAGS***

- Compiler flags for each language e.g. `-std=c++14`

LDFLAGS

- Linker flags e.g. library paths `-L/path/to/lib`

Example Makefile

The following Makefile could be used to compile example 1

```makefile
# Makefile
.PHONY : example1
example1 : bin/example1
bin/% : build/%.o
        $(CC) $(LDFLAGS) -o $@ $<
build/%.o : src/%.c
        $(CC) $(CPPFLAGS) $(CFLAGS) -o $@ -c $<
```

Which is invoked with:

```
CC=gcc make example1
```

and translates to the following commands:

```
gcc -o build/example1.o -c src/example1.c
gcc -o bin/example1 build/example1.o
```

Examples of other uses for Makefiles

- Latex

```
$(FILE).pdf : $(FILE).tex $(FILE).bbl $(FILE).aux
        $(TEX) $(FILE)
        $(TEX) $(FILE)
$(FILE).bbl : $(FILE).bib $(FILE).aux
        $(BIB) $(FILE)
$(FILE).aux : $(FILE).tex
        $(TEX) $(FILE)
```

- Parallel scripting (jturner@stackoverflow)

```
#!/usr/bin/make -f
hosts:=$(shell cat)
all : ${hosts}
${hosts} %:
        @echo "$@: `ssh $@ uptime`"
.PHONY: ${hosts} all
```

Makefile Generator Tools

- Autotools (Autoconf, Automake and Libtool)
- CMake
- scons

Installing Autotools Packages

```
./configure && make && make install
```

- By default, packages are installed to `/usr/local`.
- If you don't have root access you can change the installation prefix at configure time using `--prefix`.
- `--prefix=$HOME/.local` is recommended since it matches the convention for `pip install --user` (python modules)
- Standard variables can also be set on the command line e.g. `CC=icc ./configure`
- Always check `./configure --help` first for available options