01 - Lecture - Building a C program using gcc and make

History

The epoch
- around 1970
- UNIX and K&R C

1989
- ANSI C, ISO C89, C90

1999
- ISO C99
- gcc supports almost all C99

Hello, world!

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("%s\n", "Hello, world!");
    return 0;
}
```

compilation:

- compile, link, and execute:

```
gcc hello.c
./a.out
```

- compile:

```
gcc -c hello.c
```

  or

```
gcc -g -Wall -c hello.c
```

- link:

```
gcc hello.o
```

  or

```
gcc -g hello.o -o hello
```

- link multiple files and library:

```
gcc -g myfile1.o myfile2.o -lm -o myprogram
```

preprocessing:

- part of compilation
- process lines that begin with ’#’
- can be invoked separately with cpp or gcc -E

function definition

- return type
- argument list
- function body
- functions can only be at the top level (file scope)

main()

- the only function that a C program will execute
- other functions can be called from main()

Using multiple functions

example:

```
int add(int x, int y);

int main(int argc, char **argv)
{
    int sum;
    sum = add(1, 2);

    printf("%d\n", sum);
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

function declaration

- also called a prototype
- a function must have been seen before it’s called
- enables compiler to do type-checking

Using multiple files

example:

- myadd.h (called a header file):

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

  - myadd.c:

```
        #include "myadd.h"

        int add(int x, int y)
        {
            return x + y;
        }
```

  - main.c:

```
        #include "myadd.h"

        int main(int argc, char **argv)
        {
            ...
        }
```

preprocessor directives:

  - conditional compilation

```
        #ifdef __unix__
        printf("you are cool");
        #else
        printf("go away");
        #endif
```

  - file inclusion

```
        #include <stdio.h>
        #include "myadd.h"
```

  - macros

```
        #define  PI  3.14
```

      - just a textual substitution - so be careful!

```
        #define  square(x)  x * x     // wrong!
```

C vs. Java

C:

  - prog.c (source file) ---- [compiler] ----> prog.o (object file)

  - multiple object files ---- [linker] ----> executable file

  - objects and executables are CPU-specific and OS-specific

Java:

  - prog.java (source file) ---- javac ----> prog.class (byte code)

  - "java" (or java.exe) is the actual executable, which implements
    the Java Virtual Machine (JVM)

  - JVM runs a java program by translating machine-independent byte
    code into CPU/OS-specific machine instructions on the fly

Makefile

```
# This Makefile should be used as a template for future Makefiles.
# It’s heavily commented, so hopefully you can understand what each
# line does.

# We’ll use gcc for C compilation and g++ for C++ compilation

CC  = gcc
CXX = g++

# Let’s leave a place holder for additional include directories

INCLUDES =

# Compilation options:
# -g for debugging info and -Wall enables all warnings

CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

# Linking options:
# -g for debugging info

LDFLAGS = -g

# List the libraries you need to link with in LDLIBS
# For example, use "-lm" for the math library

LDLIBS =

# The 1st target gets built when you type "make".
# It’s usually your executable.  ("main" in this case.)
#
# Note that we did not specify the linking rule.
# Instead, we rely on one of make’s implicit rules:
#
#     $(CC) $(LDFLAGS) <all-dependent-.o-files> $(LDLIBS)
#
# Also note that make assumes that main depends on main.o,
# so we can omit it if we want to.

main: main.o myadd.o

# main.o depends not only on main.c, but also on myadd.h because
# main.c includes myadd.h.  main.o will get recompiled if either
# main.c or myadd.h get modified.
#
# make already knows main.o depends on main.c, so we can omit main.c
# in the dependency list if we want to.
```

```
#
# make uses the following implicit rule to compile a .c file into a .o
# file:
#
#     $(CC) -c $(CFLAGS) <the-.c-file>
#

main.o: main.c myadd.h

# And myadd.o depends on myadd.c and myadd.h.

myadd.o: myadd.c myadd.h

# Always provide the "clean" target that removes intermediate files.
# What you remove depend on your choice of coding tools
# (different editors generate different backup files for example).
#
# And the "clean" target is not a file name, so we tell make that
# it’s a "phony" target.

.PHONY: clean
clean:
        rm -f *.o a.out core main

# "all" target is useful if your Makefile builds multiple programs.
# Here we’ll have it first do "clean", and rebuild the main target.

.PHONY: all
all: clean main
```