Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 8

- gcc and g++
- Modular Programming
- Makefiles

How gcc and g++ roughly work

- `gcc` is a C compiler, `g++` is a C++ compiler
- They translate readable C/C++ program representation into machine code (sequence of numbers) that can be executed by CPU
- `g++ -o hello hello.c`
  - First runs the preprocessor on `hello.c`
  - Then checks whether the result is a valid C++ program,
  - if OK, it generates an assembly language representation of `hello.c` in a file – say `hello.s`
  - calls assembler (`/usr/bin/as`) with `hello.s` which produces object file `hello.o`, and
  - finally calls the linker (`/usr/bin/ld`) which generates executable file `hello` from all object (.o) files

Assembly Language

- Readable representation of machine code
- Issue `g++ -S hello.c` to create `hello.s` (`man g++` describes many more options)
- Use `less hello.s` to see what's in the file
- Assembly language is rarely used by application programmers anymore because compilers usually generate fast code
- Only needed if compiler generates slow/buggy code or it does not make use of the latest CPU instruction set extensions (e.g. SSE)

(online demonstration)

Modular Programming

Modularization makes large programming projects manageable

When implemented properly, parts can be compiled separately $\rightsquigarrow$ faster edit-compilation cycle

C/C++ way:

- Put function and type declarations in header (.h) files
- Put function definitions in modules (.c files) which can be compiled separately
- Modules that make use of functions and types need to `#include` the header files that contain their declaration

Separate Compilation of Modules

- For each module `file.c` call
  `g++ -c -o file.o file.c`

  This will create object (.o) file `file.o` which contains executable code plus house-keeping data such as function names. Object files are not executable!
- Finally, link all project object files with
  `g++ -o proj file_1.o ... file_n.o`

  This will combine all object files and start-up code into the executable file `proj`
- Libraries: archives of object files (`man ar`)

Example

foo.h:

```
#ifndef FOO_H
#define FOO_H
// comment ...
const int FOO_MAX = 100;
int foo(int x);

#endif
```

bar.h:

```
#ifndef BAR_H
#define BAR_H
// comment ...
int bar(int x);

#endif
```

foo.c:

```
#include "foo.h"
#include "bar.h"
int foo(int x) { ...bar(x)... }
```

bar.c:

```
#include "bar.h"
int bar(int x) { ... }
```

foobar.c

```
#include "foo.h"
#include "bar.h"

int main()
{
  foo(x) ... FOO_MAX ...
  bar(y) ...
}
```


Makefiles (1)

- Purpose: executing shell commands according to file dependencies and timestamps
- Handy for compilation
  - Only compile modules that depend on recent changes
  - Easy to change compiler options globally
  - Adjust to operating system environments using conditional statements
- Can also be used for other tasks including
  - Cleaning up directories
  - Create pdf-file from LaTeX source
  - Generating html-documentation (doxygen)

Makefiles (2)

- Rules = File dependencies and commands for updating files are stored in file commonly named `makefile` or `Makefile`
- Invocation: `make` or `make <target>`
  Executes commands for building first target in makefile or specific target

makefile Example

```
# executable foobar depends on foobar.o, foo.o, and bar.o
# generate it with g++ if one of those files is newer
# than foobar. foobar is "made" when make is called
foobar : foobar.o foo.o bar.o
<tab>   g++ -o foobar foobar.o foo.o bar.o

# foo.o depends on foo.c foo.h bar.h
# if one of those are newer than foo.o call g++ to update it
foo.o  :  foo.c foo.h bar.h
<tab>   g++ -c -o foo.o foo.c

bar.o  :  bar.c bar.h
<tab>   g++ -c -o bar.o bar.c

foobar.o  :  foobar.c foo.h bar.h
<tab>   g++ -c -o foobar.o foobar.c

"make" updates first target (foobar)
```


Variables

- Variables contain strings
- They can be used in command lines like so:
  - `CC := g++`
  - `CCOPTS := -Wall -O3`
  - `$(CC) $(CCOPTS)` later expands to `g++ -Wall -O3`
- Useful for changing compiler options globally

Recursively Expanded Variables

`=` sets the value of a variable that is expanded recursively

```
FOO = $(BAR)
BAR = $(MOO)
MOO = moo
```

Then `$(FOO)` is expanded to `moo`

Singly Expanded Variables

`:=` sets the value of a variable that is expanded once

```
X := foo
Y := $(X) bar
X := later
```

Then `$(Y)` is expanded to `foo bar`

Singly expanded variables contain no variable references (but their values at the time of definition)

Advantages: simpler behaviour, faster, can build lists!
E.g. `CCFLAGS := $(CCFLAGS) -O`

Pattern Rules

- Generalized file dependencies + command(s)
- Example:
  - `%.o : %.c`
    `$(CC) $(CCOPTS) -c -o $@ $<`

    means: file `%.o` depends on file `%.c` for all words `%` (`%` = wildcard)
  - command is executed whenever a `file.o` is needed and `file.c` is more recent than `file.o`
- Command line(s) must start with tab character!
- Special variables are replaced by actual values when rule is applied
  - `$@` : rule target
  - `$<` : first prerequisite
  - `$^` : all prerequisite

Complete makefile with Pattern Rule

```
CC := g++
WARN := -Wall -W -Wuninitialized

# debug settings, uncomment when debugging
# CCOPTS := $(WARN) -g

# optimization settings, uncomment when done with debugging
CCOPTS := $(WARN) -O3 -DNDEBUG

# how to compile .c files
%.o : %.c
        $(CC) $(CCOPTS) -c -o $@ $<

# link executable when .o files are newer
foobar : foobar.o foo.o bar.o
        $(CC) -o $@ $^

# remove object files and executable
clean:
        rm -rf *.o foobar

# file dependencies generated by "g++ -MM *.c"
foobar.o : foobar.c foo.h bar.h
foo.o : foo.c foo.h bar.h
bar.o : bar.c bar.h
```


GnuMake

- Part of the GNU ("Gnu is Not Unix") software collection
- Free software implementation of original make + many additional features
- Very powerful tool!
- Reading tutorials and documentation is highly recommended
  www.gnu.org/software/make/manual
- Interesting advanced reading dealing with managing large programming projects

  "Recursive make considered harmful" (google)