ENGG1340 Computer Programming II

COMP2113 Programming Technologies

Module 4a: makefile

Objectives

Estimated time to complete: 2 hours

At the end of this chapter, you should be able to:

- Understand how multiple programs could depends on each other
- Understand why separate compilation is useful and how to use it
- Understand how to use the make tool

1. Separate compilation

1.1 Multiple source files

Suppose we have the following program that reads two integers and output the corresponding GCD. The program mainly consists of 4 parts. (Note that the use of function will be covered in Module 5)

Headers	`// gcd_single.cpp`
	`// This program finds the GCD of two numbers`
	`#include <iostream>`
	`using namespace std;`

Function declaration	`int gcd(int a, int b);`

Main program	`int main() {`
	`  int a, b, c;`
	`  cout << "Please input two positive numbers: ";`
	`  cin >> a >> b;`
	`  c = gcd(a, b);`
	`  cout << "GCD is " << c << endl;`
	`}`

Function definition	`// for simplicity, we assume both inputs to be positive`
	`int gcd(int a, int b) {`
	`  while(a != b) {`
	`  if(a > b){`
	`  a -= b;`
	`  } else {`
	`  b -= a;`
	`  }`
	`  }`
	`  return a;`
	`}`

Sample compilation and execution:

```
$ g++ -pedantic-errors -std=c++11 gcd_single.cpp -o gcd_single
```

```
$ ./gcd_single
Please input two positive numbers: 18 24
GCD is 6
```

It is possible to separate the definition of function "`gcd`" from the program by define the function in a separated C++ file as shown below. This could be useful for two reasons:

- Separate features from main program for a better code organization and improve readability.
- Allow other programs to reuse the function.

```
//  gcd.cpp
#include <iostream>
#include "gcd.h"
using namespace std;

// for simplicity, we assume both inputs to be positive
int gcd(int a, int b) {
  while(a != b) {
     if(a > b){
       a -= b;
     } else {
       b -= a;
     }
  }
  return a;
}
```

To allow others to use this function definition, we need to provide a header file that declare the existence of the function. You can see that we have included *gcd.h* in the code above. This header file is shown below.

```
// gcd.h
#ifndef GCD_H
#define GCD_H

int gcd(int a, int b);

#endif
```

It is a common practice that a header file includes a set of include guard (the three lines starting with `#ifndef`, `#define`, and `#endif`) that avoids the same file being include multiple times.

The main program can then include this header file to use the function.

```
// gcd_main.cpp
// This program finds the GCD of two numbers
#include <iostream>
#include "gcd.h"
using namespace std;

int main(){
  int a, b, c;
  cout << "Please input two positive numbers: ";
  cin >> a >> b;
  c = gcd(a, b);
  cout << "GCD is " << c << endl;
}
```

To compile the new main program, you need to provide both the function definition file and the main program file.

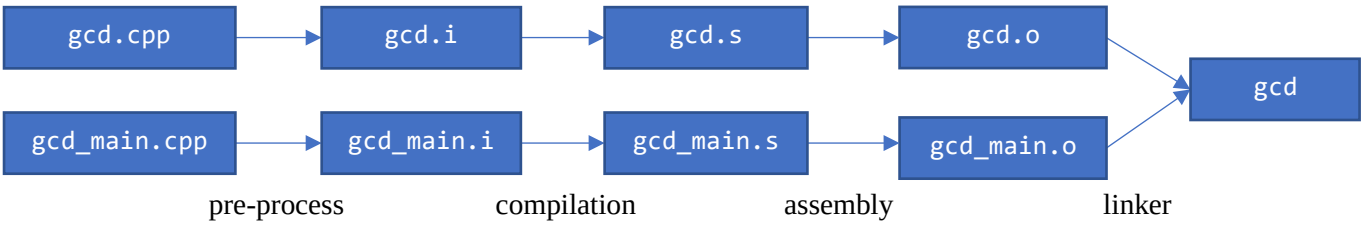
```
$ g++ -pedantic-errors -std=c++11 gcd_main.cpp gcd.cpp -o gcd
```

1.2 Compilation process

The compilation process consists of 4 stages:

1) Pre-processing – process pre-processing headers (lines starting with #) to generate an expanded source code.
2) Compilation – compile expanded source code to assembly code that depends on the machine it is executed.
3) Assembly – converts assembly code into machine code, in the form of an object code file.
4) Linker – links object codes with libraries to create an executable file.

When multiple files are involved, stage 1 to 3 will be applied to each of the files, then the last stage will be applied to all files to generate a single executable.



1.3 Separate compilation

Having understood the process of compilation, we notice that the steps from pre-processing to object code generation do not require all source files to be available. Some functions used can be declared but not defined. Hence, we can compile each source file separately up to obtain the object codes. Then, we link the object codes to form the final executable. Such a compilation scheme is called **separate compilation**.

The separate compilation has the following advantages.

- It allows programmers to write and compile their source files separately. They may also test their object code independently. It largely eases the software development process.
- If we modify only a few source files, only these files need to be recompiled. Then we can link the object codes to obtain the updated executable. It can save a significant amount of compilation time. E.g., compiling the whole Linux OS would take 8 hours; compiling only a few modified files and relinking would take less time.
- We can provide our class implementation in the form of an object code without the source file. Users can use our implementation by linking with their object code. Hence, we can hide the class implementation.

For example, we can generate the object code by using the option `-c` as shown below. (You should also add the options "`-pedantic-errors -std=c++11`" if you are working on the course materials). By default, a file with extension "`.o`" will be created.

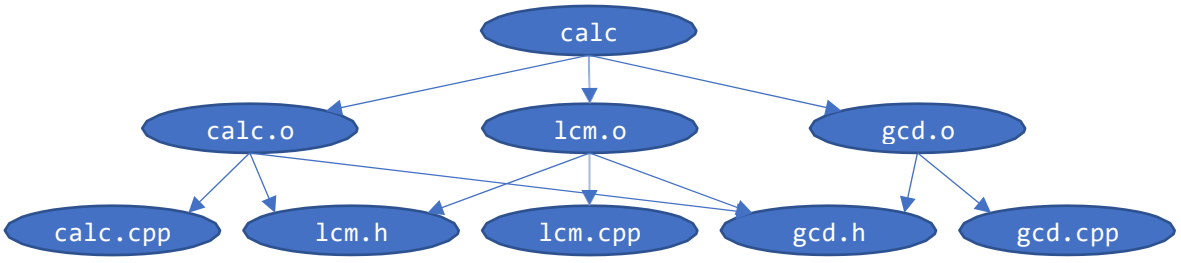
```
$ g++ -c gcd.cpp
$ g++ -c gcd_main.cpp
```

With the object codes ready, the final step is to "link" the object codes and generate the final executable (i.e. *gcd*).

```
$ g++ gcd.o gcd_main.o -o gcd
```

1.4 File dependency

Large projects can easily involve tens to hundreds of source files. They will be compiled into object codes and then linked into an executable. The following diagram shows a simple scenario.



For example, if *calc.cpp* is modified, we can rebuild the final executable by recompiling *calc.o* and then linking with (the unmodified) *gcd.o* and *lcm.o*.

In general, when the situation gets more complicated, we need a tool to handle the recompilation process.

2. Using the make tool

2.1 The make tool

The Linux **`make`** is a tool that smartly recompiles and links the files when some of the source files are changed. It tries to avoid unnecessary recompilation and regenerate a file if and only if it is needed. **`make`** needs information about the dependency of the files, which is specified in a file called **`Makefile`**. Note that the filename is important and needs to be called `Makefile` or `makefile` (No extension).

A **`Makefile`** stores a collection of rules. Here is an example of a rule:

```
gcd.o: gcd.cpp gcd.h
	g++ -c gcd.cpp
```

1. In the first line of the rule, it first gives the name of a file to be generated, which is called the target. In our example, we are specifying a target called "*gcd.o*". The target must be followed by a colon (`:`).
2. After the target (and the colon), we specify a list of files that the target depends on. In our example, the target *gcd.o* depends on *gcd.cpp* and *gcd.h*.
3. The second part of a rule gives the commands to generate the target from the files it depends on. Each command must start with a <tab>. **Note that we cannot use spaces to replace this <tab>**. There can be more than one commands used to generate the target. Those commands will be executed one after the other in order to generate the target. So in our example, when we want to generate target "*gcd.o*", we will execute `g++ -c gcd.cpp`.

Here is a completed example of a *Makefile* with 3 targets.

```
#This file must be named Makefile
#Comments start with #

gcd.o: gcd.cpp gcd.h
	g++ -c gcd.cpp

gcd_main.o: gcd_main.cpp gcd.h
	g++ -c gcd_main.cpp

gcd: gcd.o gcd_main.o
	g++ gcd.o gcd_main.o -o gcd
```

We can then use the make tool to generate a target. For example, `make gcd` will generate target gcd.

```
$ make gcd

g++ -c gcd.cpp
g++ -c gcd_main.cpp
g++ gcd.o gcd_main.o -o gcd
```

We can then use the **make** tool to generate a target. For example,

Given the above command, the make tool works as follows.

1. It first looks at the dependency list of the target. If any of the files in the dependency list is not up to date, make will make those files recursively first. A file is not up to date if it does not exist or if its last modification time is older than the last modification time of its dependency. For example, when we make *gcd_main.o*, the make tool will check whether *gcd_main.cpp and gcd.h* are up to date. If not, it makes the corresponding files.
2. After checking the dependency, make checks if the target exists or is up to date by comparing the modification time-stamp between the target and the sources. If no, make executes the commands specified in the rules to generate the target. Notice that if the target is up to date, no action is done. It is fine because the target exists and is newer than its dependency. Hence regenerating the target is just a waste of time.

Notice that the above procedure will regenerate the minimum number of files when a source file is changed.

Now, suppose *gcd_main.cpp* is modified. Then we issue the command to generate the executable *gcd* again

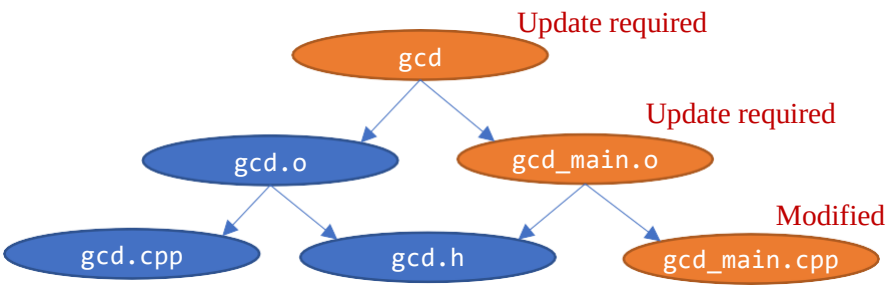
```
$ make gcd
g++ -c gcd_main.cpp
g++ gcd.o gcd_main.o -o gcd
```

`make` will first look at the dependency list of *gcd*, which are *gcd.o* and *gcd_main.o*, to ensure that these files are up to date first.

- For *gcd.o*, the last modification time are not older than those of their dependency list, so `make` realises that they are up to date. No file is regenerated.
- For *gcd_main.o*, its last modification time is older than that of its dependency *gcd_main.cpp*. Thus, make realizes that it is NOT up to date and executes the command `g++ -c gcd_main.cpp` to generate *gcd_main.o*. The last modification time of *gcd_main.o* is also updated to the current time.

Finally `make` compares the last modification time of *gcd* with the last modification times of the files in its dependency list. It finds that *gcd* is older than *gcd_main.o*.

Therefore `make` issues the command `g++ gcd.o gcd_main.o -o gcd` to generate *gcd*.



2.2 Tricks with make

2.2.1 Variables

We can define variables in a *Makefile*. It allows the users to avoid re-typing a lot of filenames. For example, the *Makefile* below defines two variables **`TARGET`** and **`OBJECTS`**.

Note that we use `$(variable)` to retrieve the value of a `variable`.

```
TARGET = gcd
OBJECTS = gcd.o gcd_main.o

$(TARGET): $(OBJECTS)
	g++ $(OBJECTS) -o $(TARGET)
```

The rule in the above *Makefile* is equivalent to the following:

```
gcd: gcd.o gcd_main.o
     g++ gcd.o gcd_main.o -o gcd
```

Three special variables are defined automatically by **make**.

Special variable	Description
$@	The target
$^	The dependency list
$<	The left most item in the dependency list

See below for an example. These two files are functionally equivalent.

```
gcd_main.o: gcd_main.cpp gcd.h
	g++ -c $<

gcd: gcd.o gcd_main.o
	g++ $^ -o $@
```

```
gcd_main.o: gcd_main.cpp gcd.h
	g++ -c gcd_main.cpp

gcd: gcd.o gcd_main.o
     g++ gcd.o gcd_main.o -o gcd
```

2.2.2 Phony targets

We can also create phony target (fake target) that is not really used to generate the target. When we make these targets, the commands will be executed, and it serves as a short hand for those commands. For example, it is common to have targets like **clean** or **tar**, as follows.

```
# previous parts snipped
clean:
	rm -f gcd gcd.o gcd_main.o gcd.tgz

tar:
	tar -czvf gcd.tgz *.cpp *.h

.PHONY: clean tar
```

When we make the target `clean`, assuming there is no file named *clean* in the current directory, the command `rm -f gcd gcd.o gcd_main.o gcd.tgz` will be executed. Essentially, it replaces the long command with the shorter one `make clean`.

Similarly, when we execute the command `make tar`, the command `tar -czvf gcd.tgz *.cpp *.h` will be executed. That tar command essentially forms a compressed collection of all .cpp and .h files into the file *gcd.tgz*.

This usage has a pitfall. If a file named *clean* really exists in the current directory, make will find that *clean* is up to date and will not execute the command. We can solve this problem by specifying that target `clean` is a "`.PHONY`" target. Targets listed in "`.PHONY`" target will still be generated even if it is up to date.

Here is a complete example of *Makefile*

```
FLAGS = -pedantic-errors -std=c++11

gcd.o: gcd.cpp gcd.h
	g++ $(FLAGS) -c $<

gcd_main.o: gcd_main.cpp gcd.h
	g++ $(FLAGS) -c $<

gcd: gcd.o gcd_main.o
	g++ $(FLAGS) $^ -o $@

clean:
	rm -f gcd gcd.o gcd_main.o gcd.tgz

tar:
	tar -czvf gcd.tgz *.cpp *.h

.PHONY: clean tar
```

Sample run:

```
$ ls
Makefile  gcd.cpp  gcd.h  gcd_main.cpp
```

```
$ make gcd
g++ -pedantic-errors -std=c++11 -c gcd.cpp
g++ -pedantic-errors -std=c++11 -c gcd_main.cpp
g++ -pedantic-errors -std=c++11 gcd.o gcd_main.o -o gcd
```

```
$ ls
Makefile  gcd       gcd.cpp  gcd.h  gcd.o  gcd_main.cpp  gcd_main.o
```

```
$ make tar
tar -czvf gcd.tgz *.cpp *.h
gcd.cpp
gcd_main.cpp
gcd.h
```

```
$ ls
Makefile  gcd       gcd.cpp  gcd.h  gcd.o  gcd.tgz  gcd_main.cpp  gcd_main.o
```

```
$ make clean
rm -f gcd gcd.o gcd_main.o gcd.tgz
```

```
$ ls
Makefile  gcd.cpp  gcd.h  gcd_main.cpp
```

3. Further reading

Some short tutorials if you want to learn more about Makefile

- Introduction to Makefiles by Johannes Franken
  http://www.jfranken.de/homepages/johannes/vortraege/make_inhalt.en.html
- A simple Makefile tutorial
  http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Tutorial on writing Makefiles
  http://makepp.sourceforge.net/1.19/makepp_tutorial.html