ENGG1340 Computer Programming II

Module 4a: makefile

Estimated time to complete: 2 hours

Objectives

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

- Understand what shell scripts are
- Write Bash shell script with a sequence of shell commands
- Understand how to use the make tool

1. Separate compilation and the make tool

In this section, we will discuss how to issue commands to do separate compilation and how to use the **make** tool.

1.1 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.

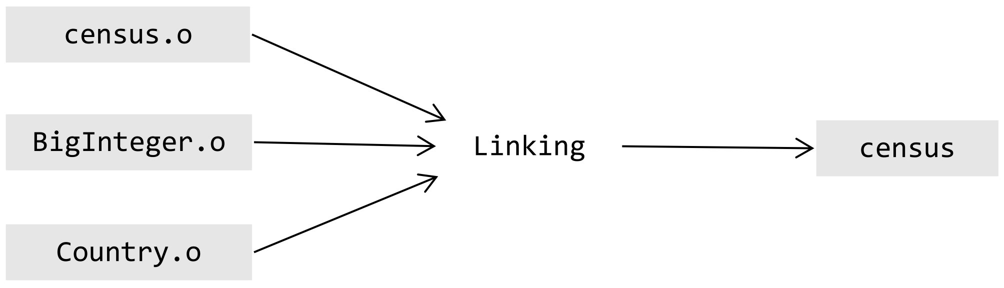
- It allows programmers to write and compile their source files separately. They may also test their object code independently. It largely eases the software development process.
- If we modify only a few source files, only these files need to be recompiled. Then we can link the object codes to obtain 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 separately by issuing the following command:

```
$ g++ -c census.cpp
$ g++ -c BigInteger.cpp
$ g++ -c Country.cpp
```

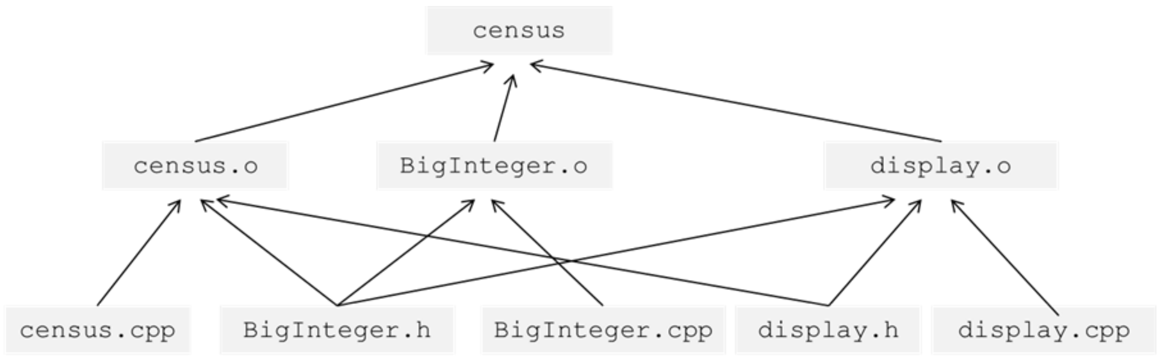
With the object codes ready, the final step is to "link" the object codes and generate the final executable (i.e. *census*).

```
$ g++ census.o BigInteger.o Country.o -o census
```



1.2 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. An arrow from file A to file B means that file B depends on file A.



If *census.cpp* is modified, we can rebuild the final executable by recompiling *census.o* and then linking with (the unmodified) *BigInteger.o* and *display.o*.

If *display.h* is modified, we need to recompile *census.o* and *display.o*, and then link with *BigInteger.o*.

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

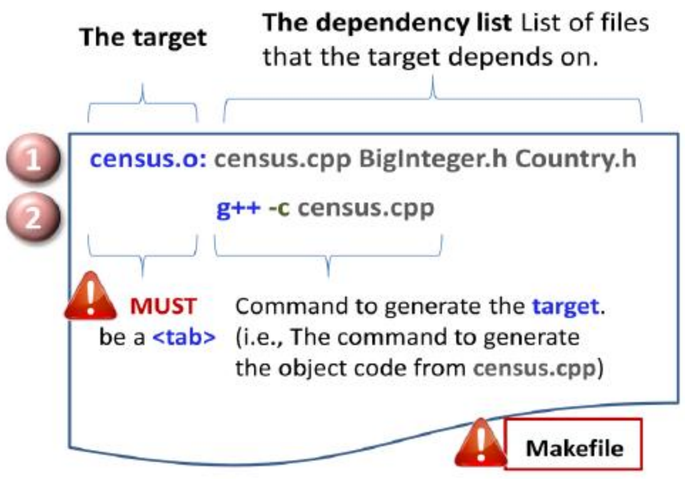
1.3 The make tool

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. Each rule contains two parts.

1. The first part of a rule gives the name of a file to be generated, which called the target. It is followed by a colon and then a list of files that the target depends on. In our example, the target *census.o* depends on *census.cpp*, *BigInteger.h* and *display.h*. This dependency relationship is specified in the first line of the Makefile below.

2. 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.


The target
The dependency list List of files that the target depends on.
1
2
census.o: census.cpp BigInteger.h Country.h
g++ -c census.cpp
MUST be a <tab>
Command to generate the target. (i.e., The command to generate the object code from census.cpp)
Makefile

Here is a completed example of a *Makefile*.

```
census.o:census.cpp BigInteger.h Country.h
	g++ -c census.cpp

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

Country.o:BigInteger.h Country.h Country.cpp
	g++ -c Country.cpp

census:census.o BigInteger.o Country.o
	g++ census.o BigInteger.o Country.o -o census

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

The above *Makefile* contains 4 rules corresponding to each file generated.

With the *Makefile* ready, we can type make X to generate any target X. For example, to generate the target *census*, enter the following command in the shell.

```
$ make census
g++ -c census.cpp
g++ -c BigInteger.cpp
g++ -c Country.cpp
g++ census.o BigInteger.o Country.o -o census
```

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 *census.o*, the make tool will check whether *census.cpp*, *BigInteger.h* and *display.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.

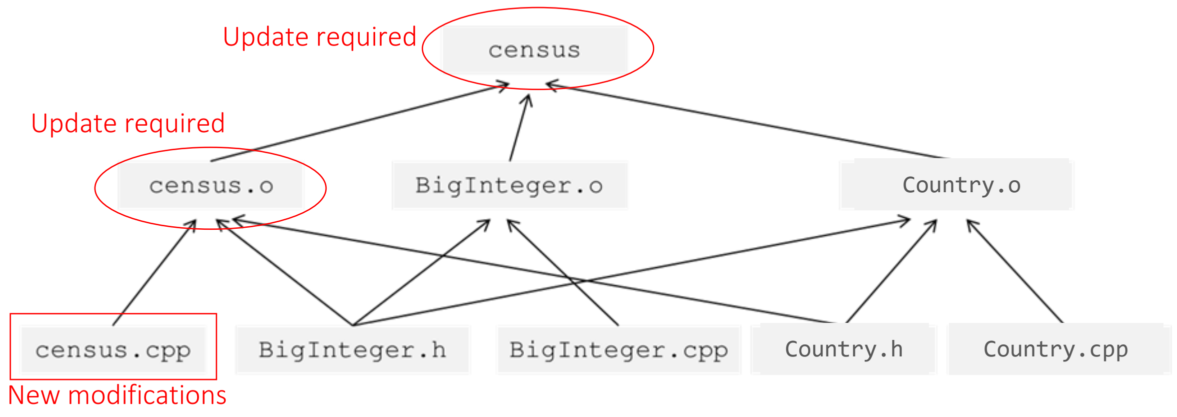
Now, suppose *census.cpp* is modified. Then we issue the command to generate the executable *census* again

```
$ make census
g++ -c census.cpp
g++ census.o BigInteger.o Country.o -o census
```

`make` will first look at the dependency list of *census*, which are *census.o*, *BigInteger.o* and *Country.o*, to ensure that the three files are up to date first.

- For *BigInteger.o* and *Country.o*, their last modification times are not older than those of their dependency list, so `make` realises that they are up to date. No regeneration of these two files is needed.
- For *census.o*, its last modification time is older than that of its dependency *census.cpp*. Thus, make realizes that it is NOT up to date and executes the command `g++ -c census.cpp` to generate *census.o*. The last modification time of *census.o* is also updated to the current time.

Finally `make` compares the last modification time of *census* with the last modification times of the files in its dependency list (i.e., *census.o BigInteger.o* and *Country.o*), it finds that census is older than *census.o*, therefore `make` issues the command `g++ census.o BigInteger.o Country.o -o census` to generate *census*.



1.4 Tricks with make

1.4.1 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 = census
OBJECTS = census.o BigInteger.o display.o

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

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

```
census:census.o BigInteger.o Country.o
	g++ census.o BigInteger.o Country.o -o census
```

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 Makefiles are functionally equivalent.

```
census: census.o BigInteger.o display.o
	g++ $^ -o $@

census.o: census.cpp BigInteger.h display.h
	g++ -c $<
```

```
census: census.o BigInteger.o display.o
	g++ census.o BigInteger.o display.o -o census

census.o: census.cpp BigInteger.h display.h
	g++ -c census.cpp
```

1.4.2 Fake targets

We can also create fake targets that will never exist. When we make those fake 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.

```
# omitted some rules
clean:
	rm census census.o BigInteger.o Country.o
tar:
	tar -czvf census.tgz *.cpp *.h

.PHONY: clean tar
```

When we make the target clean, assuming there is not file named clean in the current directory, the command `rm census census.o BigInteger.o Country.o` will be executed. Essentially, it replaces the long command with the shorter one `make clean`.

This usage has a pitfall. If a file named clean exists in the current directory, make will finds that clean is up to date and does not perform the command. We can solve this problem by specifying that clean is a ".PHONY" target. It instructs make to execute the commands even if the target is up to date.

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

Here is a completed example of Makefile using tricks

```
census.o:census.cpp BigInteger.h Country.h
	g++ -c $<
#The above command is the same as g++ -c census.cpp

BigInteger.o:BigInteger.cpp BigInteger.h
	g++ -c $<
#The above command is the same as g++ -c BigInteger.cpp

Country.o:Country.cpp BigInteger.h Country.h
	g++ -c $<
#The above command is the same as g++ -c Country.cpp

census:census.o BigInteger.o Country.o
	g++ $^ -o $@
#The above command is the same as g++ census.o BigInteger.o Country.o -o census

clean:
	rm census census.o BigInteger.o Country.o
tar:
	tar -czvf census.tgz *.cpp *.h

.PHONY: clean tar
```

Sample run

```
$ make census
g++ -c census.cpp
g++ -c BigInteger.cpp
g++ -c Country.cpp
g++ census.o BigInteger.o Country.o -o census
```

```
$ ls
BigInteger.cpp BigInteger.o census.cpp Country.cpp Country.o
BigInteger.h census* census.o Country.h Makefile
```

```
$ make clean
rm census census.o BigInteger.o Country.o
```

```
$ ls
BigInteger.cpp BigInteger.h census.cpp
Country.cpp Country.h Makefile
```

2. Further reading

Some short tutorials if you want to learn more about 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
- Makefiles by example

  http://mrbook.org/tutorials/make/
- 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

3. References

- Chapter 16. Introduction to shell script in Linux & UNIX shell programming, David Tansley, Addison-Wesley.
- Chapter 11.1 The interactive bash shell in UNIX shells by example, 3rd / 4th edition, Ellie Quigley, Prentice Hall.
- Getting Started in Programming by John S. Riley. Chapter 4.1. Interpreted vs. Compiled Languages http://www.dsbscience.com/freepubs/start_programming/start_programming.html
- Wikipedia Shebang (Unix) http://en.wikipedia.org/wiki/Shebang_(Unix)
- Chapter 5. Shell input and output and Chapter 14 Environment and shell variables in *Linux & UNIX shell programming*, David Tansley, Addison-Wesley.
- Chapter 11. The interactive bash shell and Chapter 12 Programming with the bash shell in *UNIX shells by example*, 3rd / 4th edition, Ellie Quigley, Prentice Hall.
- Bash Reference Manual http://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents
- Article: What’s the Difference Between Single and Double Quotes in the Bash Shell?
- http://www.howtogeek.com/howto/29980/whats-the-difference-between-single-and-double-quotes-in-the-bash-shell/
- Chapter 18. Control flow structure in Linux & UNIX shell programming, David Tansley, Addison-Wesley.
- Chapter 12 Programming with the bash shell in UNIX shells by example, 3rd / 4th edition, Ellie Quigley, Prentice Hall.