C++ classes and Makefiles

Tutorial 11

Katja Mankinen katja.mankinen@hep.lu.se

10-11 October 2017

Aim

This week you have created many classes: `Student, Teacher` and `Person`. You should have two files for each class:

1. header file (`classname.h`): containing the class definition
2. source code file (`classname.cpp`): containing the member function implementations

What if your program has 10 header and 20 source code files? How about 1000 files in total?

Compiling your source code files can become quite boring and tedious, especially when you want to include several source files. To make your life easier, today you will learn how to write and use Makefiles.

0.1 What is a Makefile?

Makefiles have a special format, and together with the `make` utility you can automatically build and manage your projects. Makefile is a set of rules describing build products and their dependencies, and commands to build them. Then we tell `make` to build our program. It saves you from long and complex `g++` commands. Instead, it will be enough to type `make` and it will do all the hard work for you.

As you remember from earlier lectures, there are many steps happening "behind the scene" after you have written your source code and header files and start compiling. First, the preprocessor copies the contents of the included header files and replaces symbolic constants defined using `#define` with their values. Then the pre-processed source codes are compiled into object codes (`.obj, .o`), which are then linked together with other object and library object codes to produce the final executables.

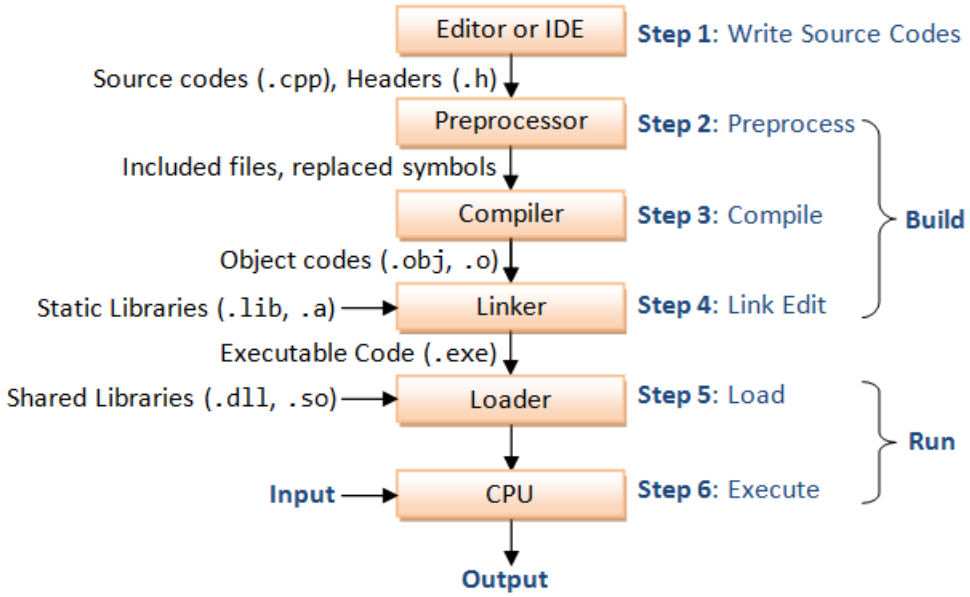

Figure 1: Figure taken from `https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp0_Introduction.html`

You may think why isn't `g++ *.cpp` enough or writing a simple shell script to compile all your files. However, it is not necessary to recompile them all when you make change to one of them. You only need to recompile a small subset of all the files. `Make` is a much more elegant solution to build our program!

1. Let's write a Makefile

In our current case, we have many files:

- Student.h, header file for the Student class
- Student.cpp, source code file for the Student class
- Teacher.h, header file for the Teacher class
- Teacher.cpp, source code file for the Teacher class
- Person.h, header file for the Person class
- Person.cpp, source code file for the Person class
- main.cpp, a main program

Today we will just write another file called Makefile.
Before moving on, let's define some words in a short glossary:

- `target`: the name of an executable or object file that is generated by `g++`
- `prerequisites` or `dependencies`: a list of files that are needed to create the target
- `command`: an action that make carries out; usually compilation or linking

In addition, you should know some compiler options:

```
g++ -Wall -g -o student student.cpp -I ./
```

- `-Wall`: prints **"all" W**arnings
- `-g`: enables extra debugging information
- `-o`: specifies the output filename
- `-I`: adds the directory to the list of directories to be searched for header files
- `-c`: compile or assemble the source code files, but do not link

You can find many many more options from g++ man page: `man g++`
First Makefile may look very scary, but when it's working, you just need to type in "make" and everything else is done automatically for you.
As a pseudo-code a Makefile looks something like

```
<the file> : <needs these files>
[TAB] <the file is created by this command>
```

which is essentially the same as

```
target : dependency1 dependency2 ...
[TAB] command1
[TAB] command2
```

so not too bad!
For example, very simple case with Hello world:

```
all: hello

hello: main.o hello.o
    g++ main.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm *o hello
```

Let's now get more familiar with Makefiles: how to write and use them in our Student/Teacher/Person case. **Go to Live@Lund, navigate to Lesson plans and read through a file "Makefile". Try to understand what is written, implement missing parts, and test it.**