Linux C/C++ Programming

Dr. Changjiang Zhang

How to Install GCC/G++ in Ubuntu

General steps:

1. sudo apt update
2. sudo apt upgrade
3. sudo apt install build-essential

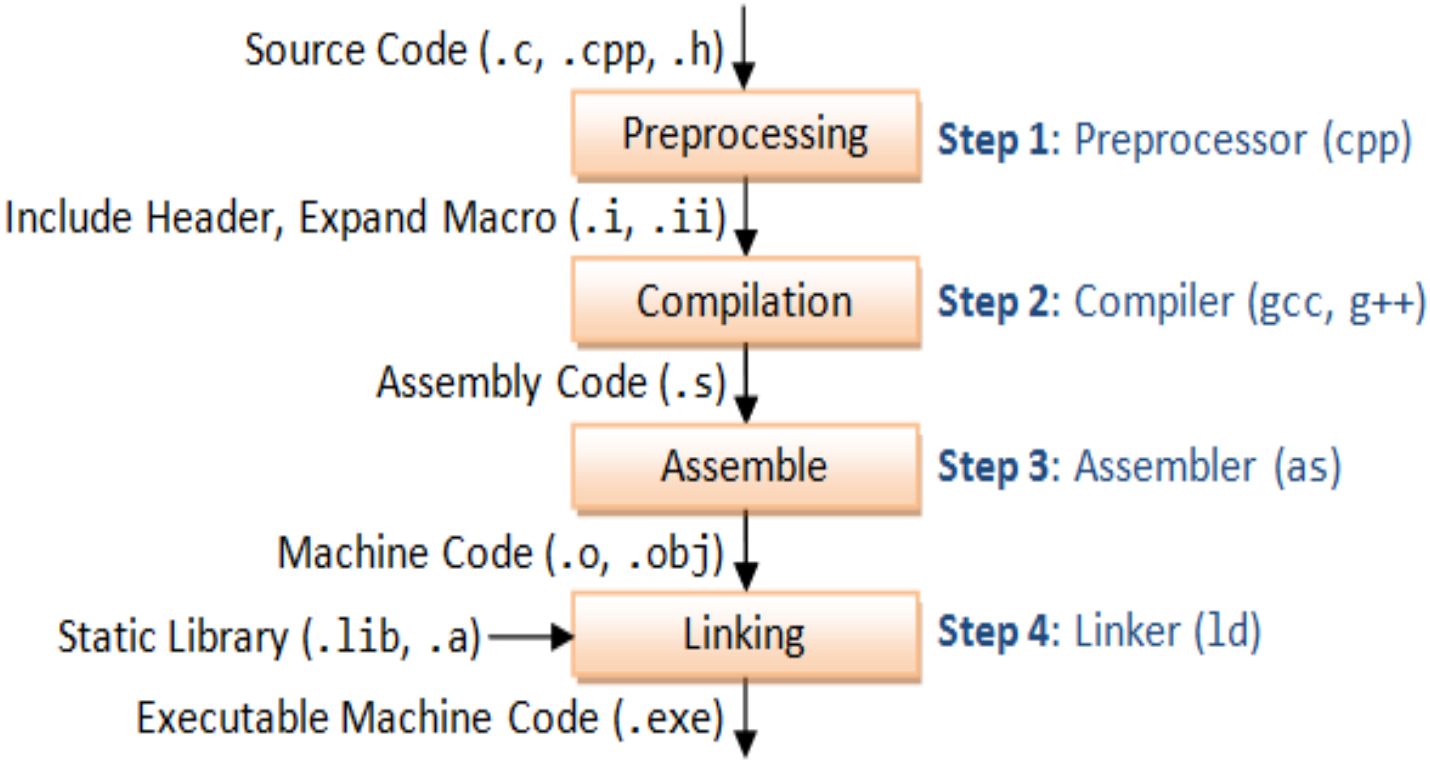
update is synchronization /etc/apt/sources.list (file) and /etc/apt/sources.list.d (directory) to get the latest package.

upgrade is to upgrade all installed software packages. update must be performed before upgrade.

Once finished, check your gcc/g++ version

```
$gcc  --version
$g++ --version
```

GCC Compilation Process



GCC(C/C++) Flags

gcc/g++

Synopsis:

gcc/g++ [**-c**|**-S**|**-E**] [**-std=***standard*] [**-g**] [**-pg**] [**-O***level*] [**-W***warn*...] [**-pedantic**] [**-I***dir*...] [**-L***dir*...] [**-D***macro*[=*defn*]...] [**-U***macro*] [**-f***option*...] [**-m***machine-option*...] [**-o** *outfile*] [@*file*] *infile*...

Commonly used flags：

-c -- Compile or assemble the source files, but do not link.

-E -- Stop after the preprocessing stage; do not run the compiler proper.

-o *file* -- specify executable *file*. If -o is not specified, the default is to put an executable file in a.out

-S -- Create assembly code (.s), no other files created.

-g -- Produce debugging information

-pg -- Generate extra code to write profile information suitable for the analysis program gprof.

-O -- optimization flag: -O0, -O1, -O2, -O3, -Os(Optimize for size)

-Wall -- Display all warnings during compilation

-w -- Suppress all warnings during compilation

-std=*standard* -- Specify the *standard* to which the code should conform

-I dir -- Add the directory dir to the list of directories to be searched for header files.

GCC – GNU project C/C++ compiler

`gcc for c code (.c, .h)`

command	output
gcc p.c	It will compile p.c, if no error, an executable code a.out will be created.
gcc –c p.c	It will only create an object file (p.o), no link, so no executable file.
gcc –S p.c	It only ouputs assembler code (p.s)
gcc –o p p.c	It will create an executable file(p). Don't output assembler or obj files.

GCC – GNU project C/C++ compiler

```
//C code: demo1.c
#include <stdio.h>

int main() {
        printf("This is a C program\n");
        return 0;
}
```

Compile with command:

```
gcc -o demo1 demo1.c
```

Demo1 will be created. You can run it:

```
./demo1
```

GCC – GNU project C/C++ compiler

g++ for C++ code (.cc, .cpp, .cxx, c++, .h, .h++, .hxx. .hpp, etc.)

command	output
g++ p.cpp	It will compile p.cpp, if no error, an executable code a.out will be created.
g++ –c p.cpp	It will only create an object file (p.o), no link, so no executable file.
g++ –S p.cpp	It only ouputs assembler code (p.s)
g++ –o p p.cpp	It will create an executable file(p). Don't output assembler or obj files.

GCC – GNU project C/C++ compiler

```
//C++ code: demo2.cpp
#include <iostream>
using namespace std;

int main() {
        cout<<"This is a C++ program"<<endl;
        return 0;
}
```

Compile with command:

```
g++ -o demo2 demo2.cpp
```

Demo2 will be created. You can run it:

```
./demo2
```

Project with a few C/C++ code files

Step1:

Using gcc/g++ -c to compile each source code file to create .o file

Step2:

Using gcc/g++ to link all obj files(.o) together to create an executable code

For example, a small project has 2 source files and a few .h files:

Step1: Type a few commands to create obj files

gcc **-c p1.c**

gcc –c p2.c

Step2: Link all obj files to create executable

gcc –o p p1.o p2.o

Question: Project with many C/C++ code files, what should we do?

Exercise

Create a sample c code as follows using vi:

vi hello.c

Click i

```
#include <stdio.h>
int main() {
        printf("Hello World!\n");
        return 0;
}
```

Click ESC

Type :wq

Exercise Cont.

Try a few flag using gcc

$gcc –c hello.c

$gcc –E hello.c

$gcc –o hello hello.c

$gcc –S hello.c

$gcc –g hello.c
Check the output file a.out size with ls -l

$gcc –pg hello.c
Check the output file a.out size with ls -l

$gcc –Os hello.c
Check the output file a.out size with ls -l

Critical Thinking

Imaging there is a project, which includes hundreds of C/C++ source code files.

How do you compile those source code files and create an executable code for this project?

In Windows OS: some kind of IDEs
In Mac OS: XCODE, or other IDEs
In Linux OS: ? makefile

Example Program

Here, we will use a program that deals with some geometric shapes to illustrate how to write a make file to compile programs in a project.

The program is made up of 5 source files:

1. main.cpp, a main program.
2. Point.h, header file for the Point class.
3. Point.cpp, implementation file for the Point class.
4. Rectangle.h, header file for the Rectangle class.
5. Rectangle.cpp, implementation file for the Rectangle class.

Separate Compilation

Let's review what we need to do to compile this program *separately* (i.e., to intermediate *object files*) and then link it into an executable named main.

To just compile source code, use the -c flag with the compiler...

- **g++ -c main.cpp**
- **g++ -c Point.cpp**
- **g++ -c Rectangle.cpp**

This will generate three object files:

- main.o (for main.cpp),
- Point.o (for Point.cpp), and
- Rectangle.o (for Rectangle.cpp)

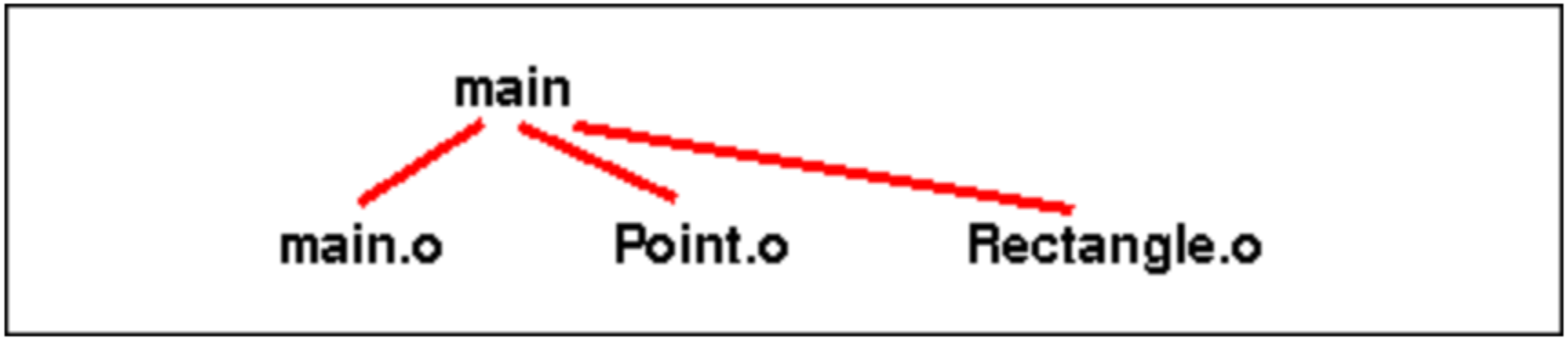
Then, to link the object files (.o) into an executable, we use the compiler again (although this time it will just pass the .o files on to the *linking* stage):

- **g++ -o main main.o Point.o Rectangle.o**

Dependencies

We would like to know what files need to be regenerated when we change certain parts of our program. We can determine this by creating a *dependency chart*.
Let's start from our goal (i.e., to have an executable named main)...

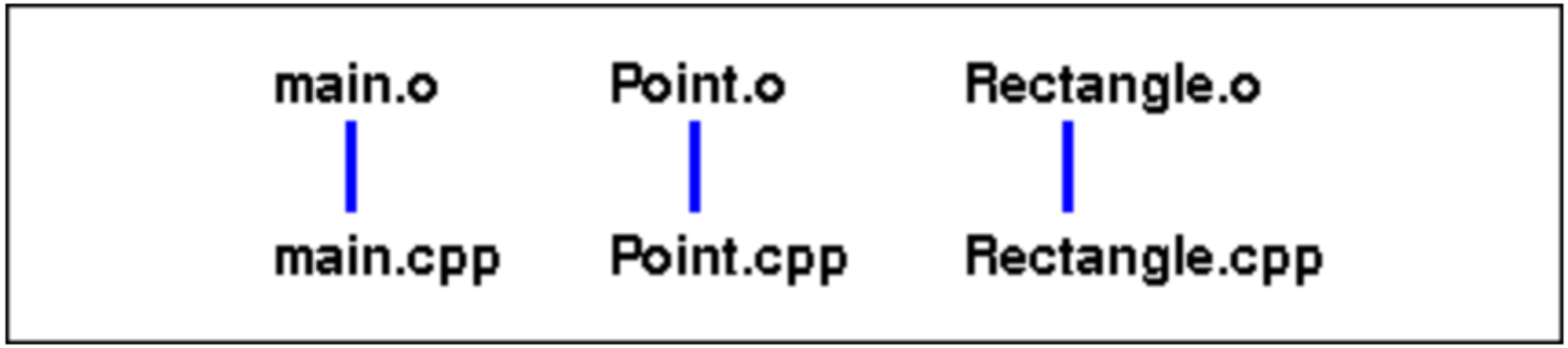
Linking Dependencies
The executable **main** is generated from 3 object files, **main.o**, **Point.o** and **Rectangle.o**. Thus, **main** *depends* on those 3 files.



Compiling Dependencies

Next, the object (.o) files depend on the .cpp files. Namely:

- **main.o** depends on **main.cpp**,
- **Point.o** depends on **Point.cpp**, and
- **Rectangle.o** depends on **Rectangle.cpp**.

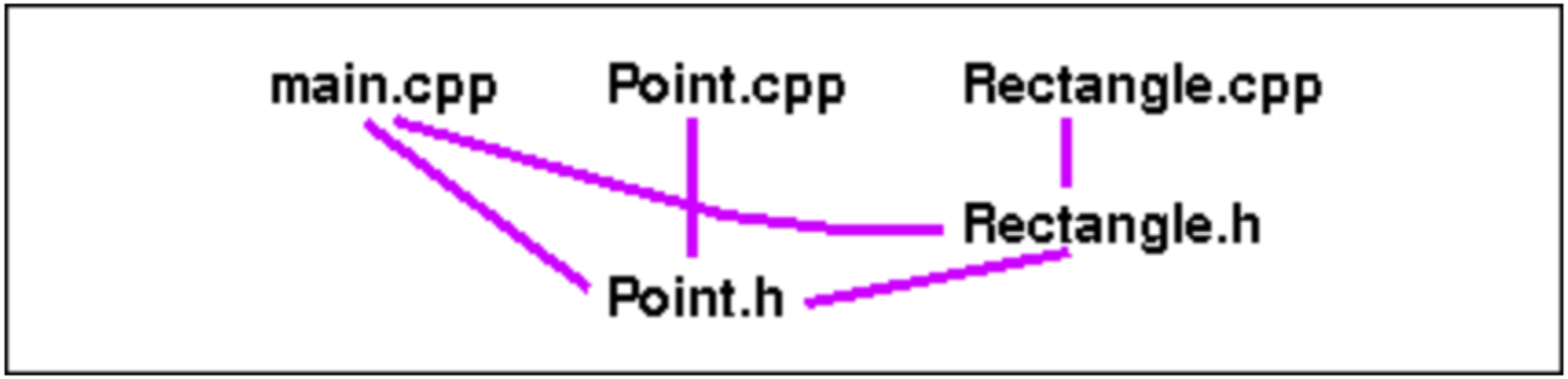


You generate these object files by *compiling* the corresponding .cpp files

Include Dependencies

Finally, source code files (.cpp and .h) depend on header files (.h) that they include:

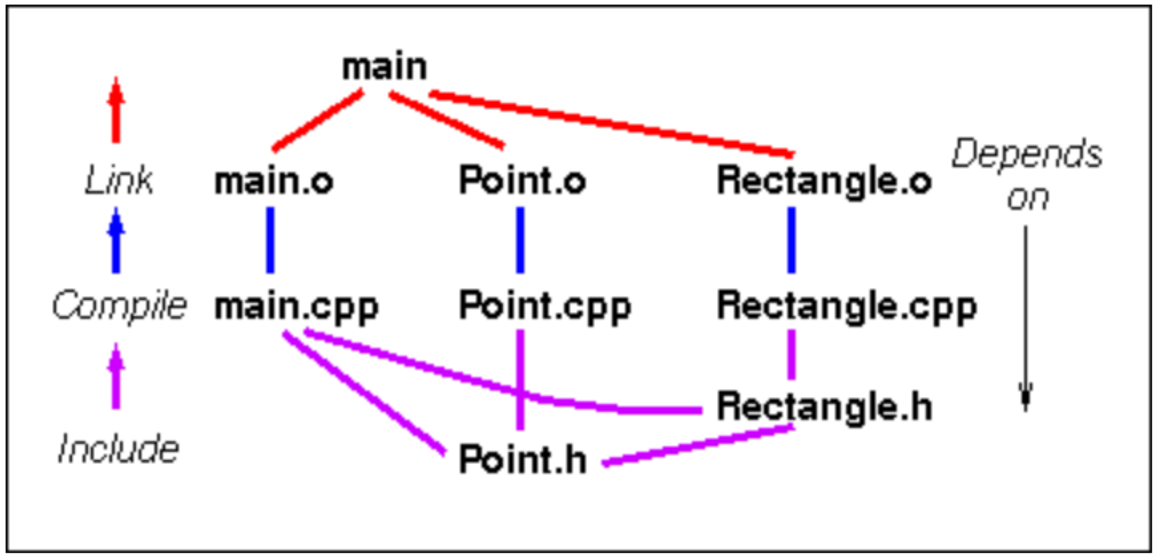
- 3 source code files include **Point.h** and
- 2 files include **Rectangle.h**.



- Notice that there may be additional *indirect* include dependencies (e.g., Rectangle.cpp depends on Point.h since Rectangle.cpp includes Rectangle.h, which includes Point.h).

- **Note:** Include dependencies are a little different since the .h files aren't used to *generate* other files, unlike .cpp and .o files.

Complete Dependency Chart



Dependencies go downward.
How to "get" or generate files goes upward. Now, we can answer questions about what needs to be regenerated if certain files change.

For example, suppose we change the file main.cpp...*What has to be regenerated?*
Now, suppose we changed Point.h?

Note: When either a .cpp file changes or a header file included (directly or indirectly) by a .cpp file changes, we have to regenerate the corresponding .o file.

Complete Dependency Chart

- Since it is tedious to recompile pieces of a program when something changes, people often use the ***make*** utility instead.
- *Make* needs a ***makefile*** that encodes both the dependencies between files and the commands needed to generate files.
- When you run the *make* utility, it examines the *modification times* of files and determines what needs to be regenerated. Files that are *older* than the files they depend on must be regenerated.
- Regenerating one file may cause others to become *old*, so that several files end up being regenerated.

Parts of Make File

- For this example, we will examine the parts of this Makefile, which generates the executable main.

- Simple make files have at least 2 parts:
  - A set of *variables*, which specify things like the C++ compiler/linker to use, flags for the compiler, etc.
  - A set of *targets*, i.e., files that have to be generated.

- Makefile may have comments that (hopefully) add to readability. Any line starting with a pound sign (#) is a comment. Note that our makefile has some comments.

Make File

```
# Makefile for Writing Make Files Example
# ****************************************************
# Variables to control Makefile operation
CXX = g++
CXXFLAGS = -Wall -g
# ****************************************************
# Targets needed to bring the executable up to date
main: main.o Point.o Rectangle.o
	$(CXX) $(CXXFLAGS) -o main main.o Point.o Rectangle.o
# The main.o target can be written more simply
main.o: main.cpp Point.h Rectangle.h
	$(CXX) $(CXXFLAGS) -c main.cpp
Point.o: Point.cpp Point.h
	$(CXX) $(CXXFLAGS) -c Point.cpp
Rectangle.o: Rectangle.cpp Rectangle.h Point.h
	$(CXX) $(CXXFLAGS) -c Rectangle.cpp
clean:
	rm *.o main
```

Using Make File

Try using the make file we've given you.
Run *make* with the command:

make

Remember that this will cause *make* to generate the *first target* in the make file, which is **main**.

Make File Rewrite

```
# Makefile for Writing Make Files Example
# ************************************************
# Variables to control Makefile operation
CXX = g++
CXXFLAGS = -Wall -g
# ************************************************
# Targets needed to bring the executable up to date
main:       *.o
            $(CXX) $(CXXFLAGS) -o main *.o
# The *.o target can be written more simply for any cpp file to be compiled to obj file
*.o:        *.cpp *.h
            $(CXX) $(CXXFLAGS) -c *.cpp
clean:
            rm *.o main
```

It can support any number of source codes