Ve 280

Programming and Elementary Data Structures

Developing and Compiling Programs on Linux

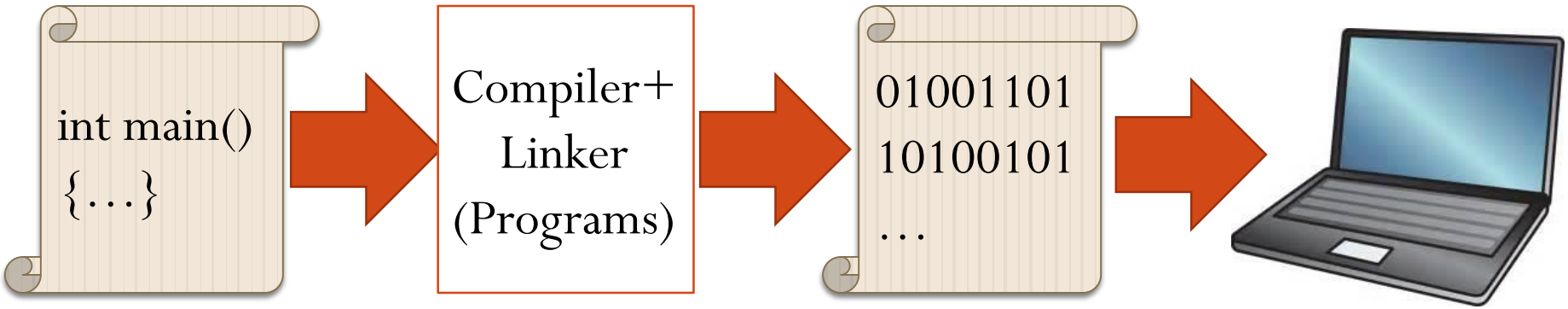
Learning Objectives:

Understand the compilation process

How to compile a single source file

How to compile multiple source files

Basic Working Mechanism of Computer



Developing a Program on Linux

Single Source File

- Write the source code, for example, using **gedit**
- Compile the program
  - Compiler: g++
  - Command: g++ -o program source.cpp
    - -o option tells what the name of the output file is.
- Run the program: ./program

- Useful options of g++
  - -g: Put debugging information in the executable file
  - -Wall: Turn on all warnings!

Compile a Program

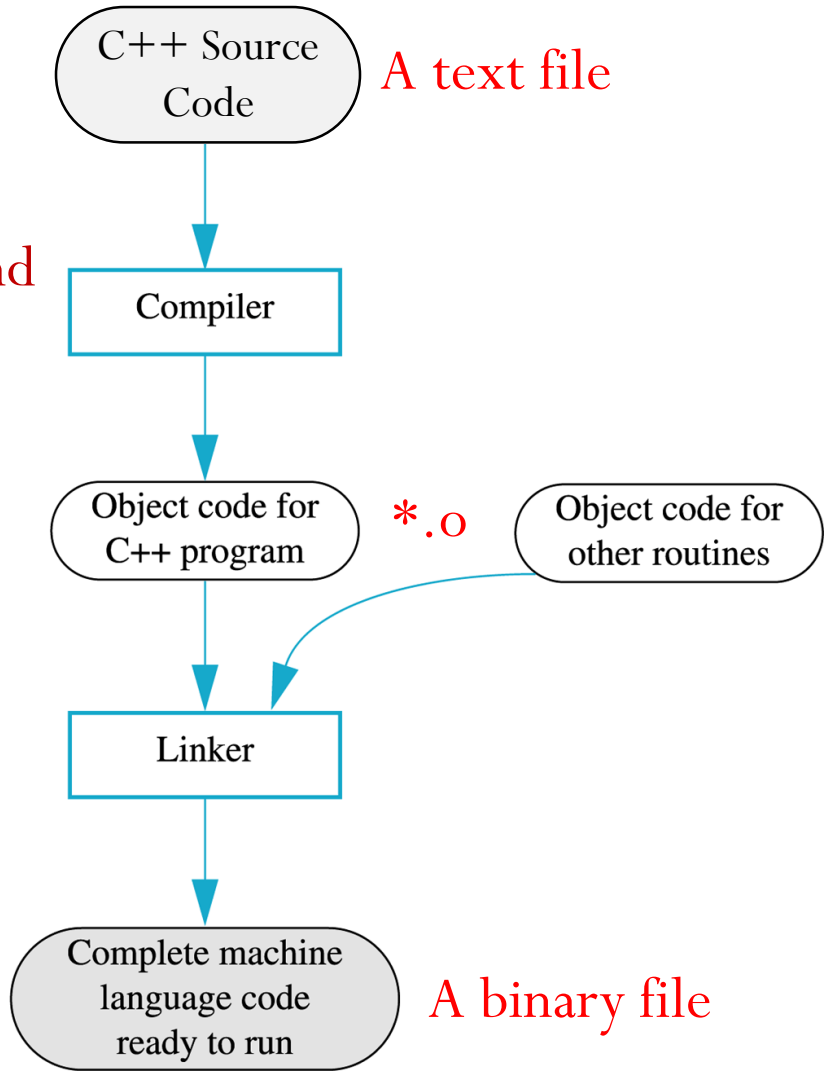
g++ -o program source.cpp

= g++ -c source.cpp
g++ -o program source.o

Link command

Object code: portion of machine code that has NOT yet been linked into a complete program

- Just machine code for one particular library or module
- Can be generated by command
  g++ -c source.cpp



A large project is usually split into several source files in order to be manageable. Why?

Select all the correct answers.

- **A.** To speed up compilation – changing a single line only requires recompiling a single small source file. Much faster!
- **B.** To increase organization – make it easier for you to find functions, variables, etc.
- **C.** To facilitate code reuse.
- **D.** To split coding responsibilities among programmers.

Developing Program on Linux

Multiple Source Files

- Multiple source files include two types of files
  - header files – "**.h**" files: normally contain class definitions and function declarations.
  - C++ source files – "**.cpp**" files: normally contain function definitions and member functions of classes.
- Example

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

```
// add.cpp
int add(int a, int b)
{
   return a+b;
}
```

Developing Program on Linux

Multiple Source Files

- If a function in another file calls function `add()`, we should put `#include "add.h"` in that file.

- Example

```
// run_add.cpp
#include "add.h"
int main()
{
   add(2,3);
   return 0;
}
```

In C++, the **preprocessor** replaces each #include by the contents of the specified file.

Headers Often Need Other Headers

`line.h`

```
#include "point.h"
...
```

`drawing.h`

```
#include "point.h"
#include "line.h"
...
```

- Consequence: A header file may be included more than once in a single source file
  - Which header file is included for more than once in this example?
  - Answer: in `drawing.h`, we include `point.h` twice

Problem of Multiple Inclusions

- The including of a header file more than once may cause **multiple** definitions of the classes and functions defined in the header file.
  - Compiler complains!

- Solution: **header guard**.
  - It avoids **reprocessing** the contents of a header file if the header has already been seen.

Header Guard

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

Header guard to prevent multiple definitions!

- `#ifndef VAR`: a conditional directive --- tests whether the **preprocessor variable** VAR has **not** been defined.
  - If not defined, #ifndef **succeeds** and all lines up to #endif are processed.
    - Specially, #define defines VAR.
  - If defined, #ifndef **fails** and all lines between #ifndef and #endif are **ignored**.

Header Guard

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

- What happens if the header is included **first** time?
  - #ifndef succeeds. ADD_H is defined and the content is included
- What happens if the header is included **second** time?
  - Since ADD_H has been defined the first time we include the header, #ifndef fails. The lines between #ifndef and #endif are ignored
  - Good! No multiple declarations of the function `add`
- With header guard, we guarantee that the definition in the header is just seen **once**!

Compiling Multiple Source Files

- To compile multiple source files, use command
  - g++ -Wall -o program src1.cpp src2.cpp src3.cpp

Program name | All .cpp files

  - E.g., g++ -Wall -o run_add run_add.cpp add.cpp

- Note: you don't put ".h" in the compiling command
  - I.e., you don't need
    g++ -Wall -o program src1.cpp src1.h src2.cpp src3.cpp
  - Why? ".h" files are already included.
    E.g., run_add.cpp includes add.h

One More Thing

- For our example on defining function, there is no need to "#include add.h" in "add.cpp"

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

```
// add.cpp
int add(int a, int b)
{
   return a+b;
}
```

- However, for defining class, you need to include the .h file in the corresponding .cpp file.

Another Way

- Generate the object codes (.o files) **first**
- Example: g++ -Wall -o run_add run_add.cpp add.cpp
  - **Equivalent** way:

```
g++ -Wall -c run_add.cpp  # will produce run_add.o
g++ -Wall -c add.cpp      # will produce add.o
g++ -Wall -o run_add run_add.o add.o
```

What are the advantages/disadvantages of compiling the cpp files separately?

Select all the correct answers.

- **A.** Advantage: Only changed files need to be recompile.
- **B.** Advantage: It facilitates code reuse.
- **C.** Disadvantage: It requires a lot of typing!
- **D.** Disadvantage: It requires us to remember which files have been changed.

A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
	g++ -o run_add run_add.o add.o

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

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

clean:
	rm -f  run_add   *.o
```

A Rule

Target: Dependency
<Tab> Command

Don't forget the Tab!

Dependency: A list of files that the target depends on

A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
        g++ -o run_add run_add.o ad

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

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

clean:
        rm -f  run_add   *.o
```

- The file name is "Makefile"
- Type "make" on command-line for the first target ("all" in this case)
- Type "make <target>" for a specific <target>

Target: Dependency
<Tab> Command

Usually, there is a target called "clean"

- A **dummy target**. Type "make clean"
- It has no dependency!
- Question: what does "clean" do?

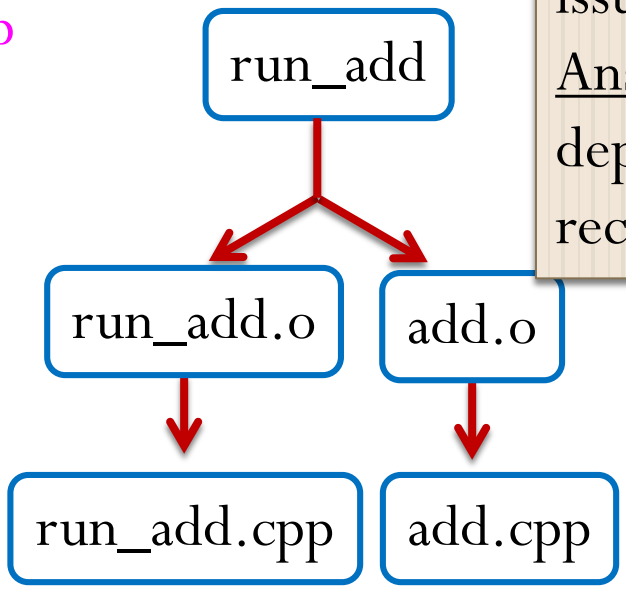
A Better Way: Makefile

```
all: run_add

run_add: run_add.o add.o
        g++ -o run_add run_add.o add.o

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

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

clean:
        rm -f  run_add   *.o
```

A Rule

Target: Dependency
<Tab> Command

Dependency Graph



When is a command issued?

Answer: When dependency is more recent than target

References

- Makefile
  - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Developing Programs on Linux
  - C++ Primer, 4th Edition, Chapter 2.9