Ve 280

Programming and Elementary Data Structures

Linux;

Developing and Compiling Programs on Linux

Outline

- Linux Basics

- Developing and Compiling Programs on Linux

I/O Redirection

- Many commands can accept input from a facility called **standard input**.
  - By default, standard input is our keyboard.
- We can redirect standard input from a file instead of keyboard by using '<'.
  - One application: testing
  - E.g., my_add < input.txt
    # my_add is a program taking two inputs from keyboard and output their sum on screen
- Question: what does the following command mean?
  - my_add < input.txt > output.txt

Other Commands

- Auto completion: type a few characters; then press 'Tab'
  - If there is a single match, Linux completes the remaining.
  - If there are multiple matches, hit the second time, Linux show the candidates.
- Compare two files: diff file1 file2
  - If files are the same, no output
  - If there are differences: lines after "<" are from the first file; lines after ">" are from the second file
  - In a summary line: 'c': change; 'a': add; 'd': delete
  - Useful option "-w": ignore white spaces (space, tab)

Other Commands

- Install a program: sudo apt-get install program
  - E.g., sudo apt-get install emacs
  - sudo command: execute command as a superuser
    - Need you to type your password
- Remove a program: sudo apt-get autoremove program
- Looking for help? man command E.g., man ls
  - Browse the manual using the same command as for ‘less’

Outline

- Linux Basics

- Developing and Compiling Programs on Linux

Developing 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
```

=

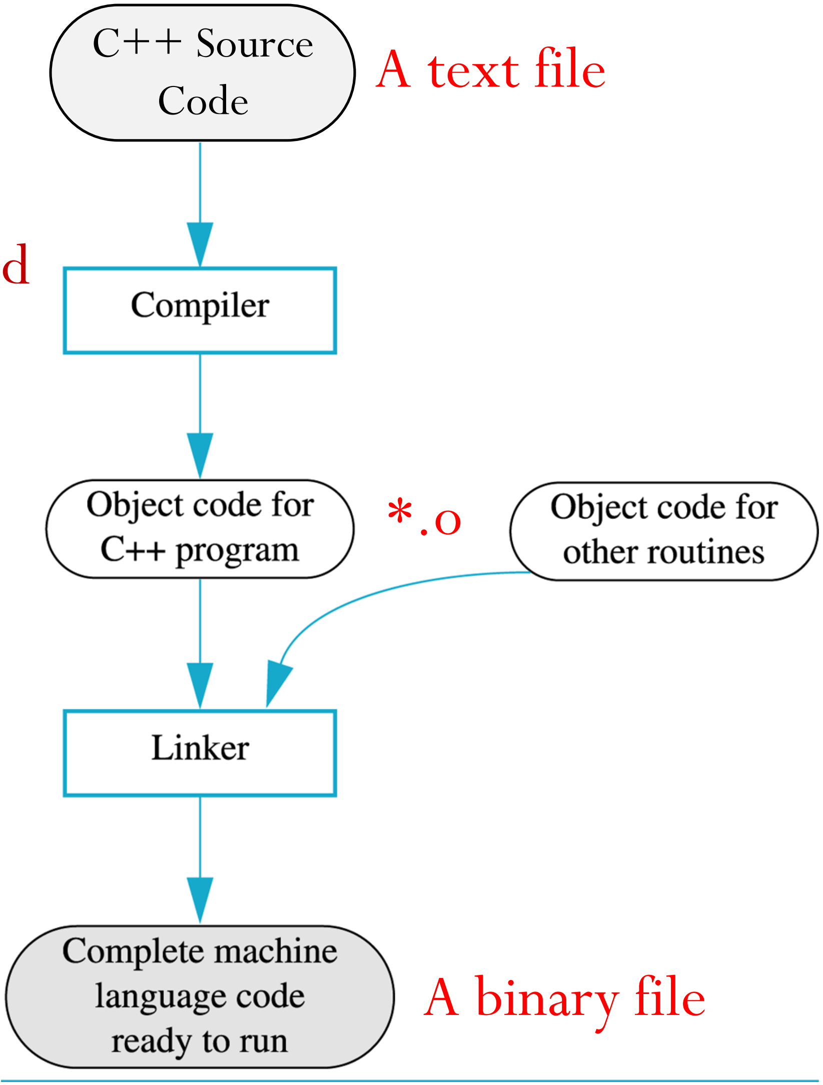
```
g++ -c source.cpp
g++ -o program source.o
```

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
```


C++ Source Code
A text file
Compiler
Object code for C++ program
*.o
Object code for other routines
Linker
Complete machine language code ready to run
A binary file

Developing Program on Linux

Multiple Source Files

- A large project is usually split into several source files in order to be manageable.
- Why?
  - To speed up compilation – changing a single line only requires recompiling a single small source file. Much faster!
  - To increase organization – make it easier for you to find functions, variables, etc.
  - To facilitate code reuse.
  - 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;
}
```

The #include is C++ **preprocessor**.

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
  - E.g., in `drawing.h`, we include `point.h` twice

Avoiding Multiple Inclusions

- We must ensure that including a header file more than once does not cause **multiple** definitions of the classes and functions defined in the header file.
  - Otherwise, 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.
  - 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 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 have
    g++ -Wall -o program src1.cpp src1.h src2.cpp src3.cpp
  - Why? ".h" files are already included

Another Way

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

    ```
    g++ -c run_add.cpp  # will produce run_add.o
    g++ -c add.cpp      # will produce add.o
    g++ -o run_add run_add.o add.o
    ```
  - Advantage?
  - Disadvantage?

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
```

- The file name is "Makefile"
- Type "make" on command-line

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

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
```

There is a target called “all”

- It is the **default** target
- Its dependency is program name
- It has no command

A Rule

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?

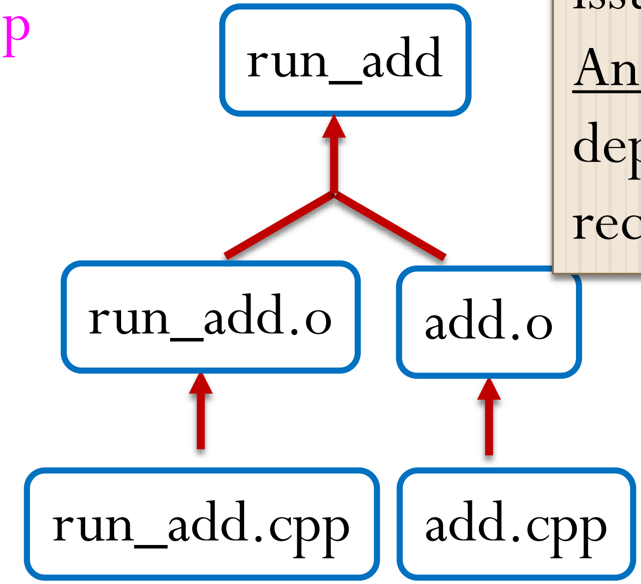
A Better Way: Makefile

```
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 Tree



When is a command issued?

Answer: When dependency is more recent than target

References

- Linux
  - http://linuxcommand.org/

- Developing Programs on Linux
  - C++ Primer, 4th Edition, Chapter 2.9

- Makefile
  - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/