Ve 280

Programming and Elementary Data Structures

Developing Programs on Linux;

Review of C++ Basics

Outline

- Developing programs on Linux

- Review of C++ basics

Compile a Program

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

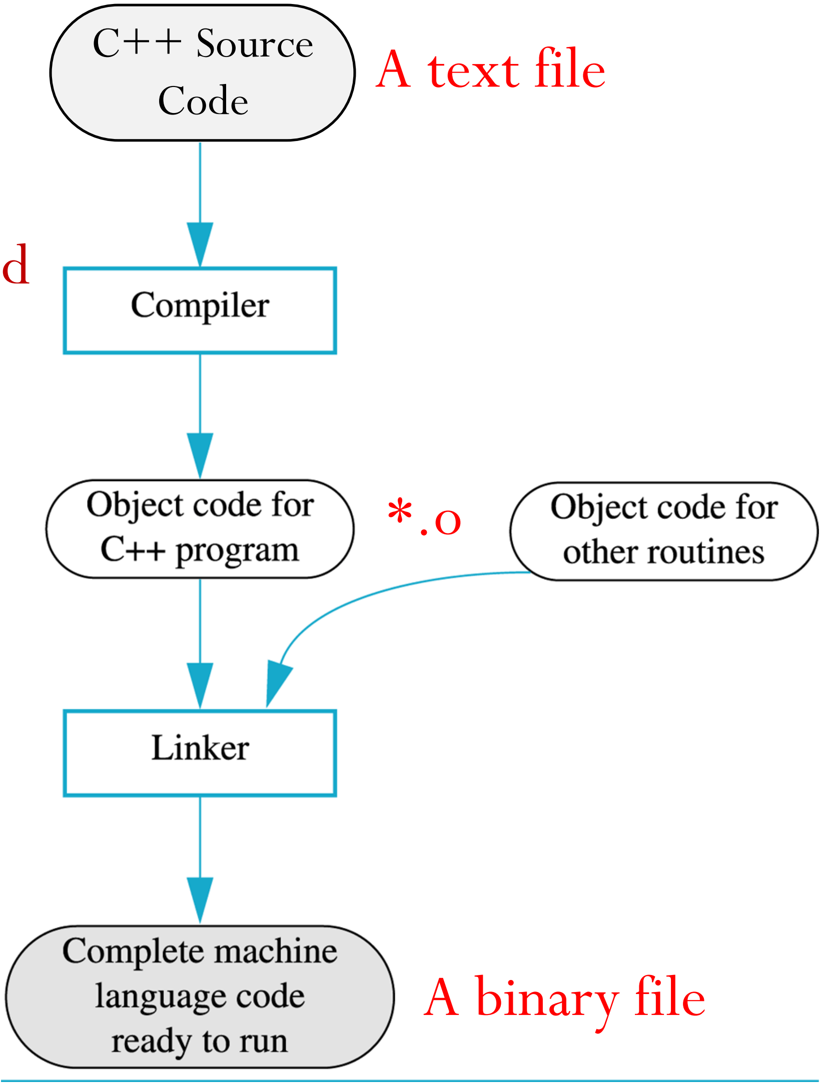
=

```
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
Compiler
Object code for C++ program
Object code for other routines
Linker
Complete machine language code ready to run

A text file

*.o

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;
}
```

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
  - E.g., 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 have

    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

Another Way

- Generate the object codes (.o files) **<u>first</u>**
- Example: g++ -Wall -o run_add run_add.cpp add.cpp
  - **<u>Equivalent</u>** 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
    ```
  - 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?

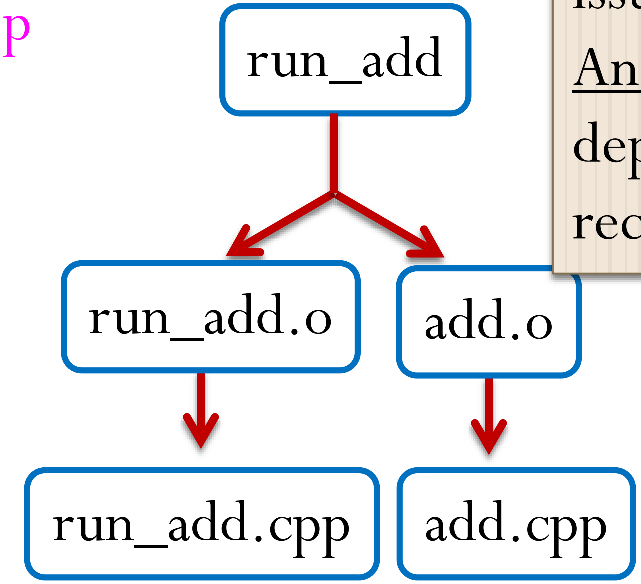
A Better Way: Makefile

```
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 Graph



run_add

run_add.o

add.o

run_add.cpp

add.cpp

When is a command issued?

<u>Answer</u>: When dependency is more recent than target

Outline

- Developing programs on Linux

- Review of C++ basics

Very Basic Concepts

- Variables
- Built-in data types, e.g., `int`, `double`, etc.
- Input and output, e.g., `cin`, `cout`.
- Operators
  - Arithmetic: +, -, *, etc.
  - Comparison: <, >, ==, etc.
  - x++ versus ++x
- Flow of controls
  - Branch: if/else, switch/case
  - Loop: while, for, etc.

An Example

```
#include <iostream>
using namespace std;
int main() {
  // Calculating the area of a square
  int length, area;
  cin >> length;
  if(length > 0) {
    area = length * length;
    cout << “area is ” << area << endl;
  }
  else
    cout << “negative length!” << endl;
  return 0;
}
```

lvalue and rvalue

- Two kinds of expressions in C++
  - **lvalue**: An expression which may appear as either the left-hand or right-hand side of an assignment
  - **rvalue**: An expression which may appear on the right- but not left-hand side of an assignment

- Which of the followings are lvalues? Which are rvalues?
  - `a    // a is an int variable`
  - `10`
  - `a+1 // a is an int variable`
  - `a+b // a and b are two int variables`
  - `a[2*3] // a is an array`

Function Declarations vs. Definitions

- Function declaration (or function prototype)
  - Shows how the function is called.
  - Must appear in the code before the function can be called.
  - Syntax:

```
Return_Type Function_Name(Parameter_List);
//Comment describing what function does
```

```
int add(int a, int b); //Comment
```

- Function definition
  - Describes how the function does its task.
  - Can appear before or after the function is called.
  - Syntax:

```
Return_Type  Function_Name(Parameter_List)
{
  //function code
}
```

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

Function Declaration

- Tells:
  - return type
  - how many arguments are needed
  - types of the arguments
  - name of the function
  - formal parameter names

Type Signature

Formal Parameter Names

- Example:

```
double total_cost(int number, double price);
// Compute total cost including 5% sales tax on
// number items at cost of price each
```

Function Definition

- Provides the same information as the declaration
- Describes how the function does its task

- Example:

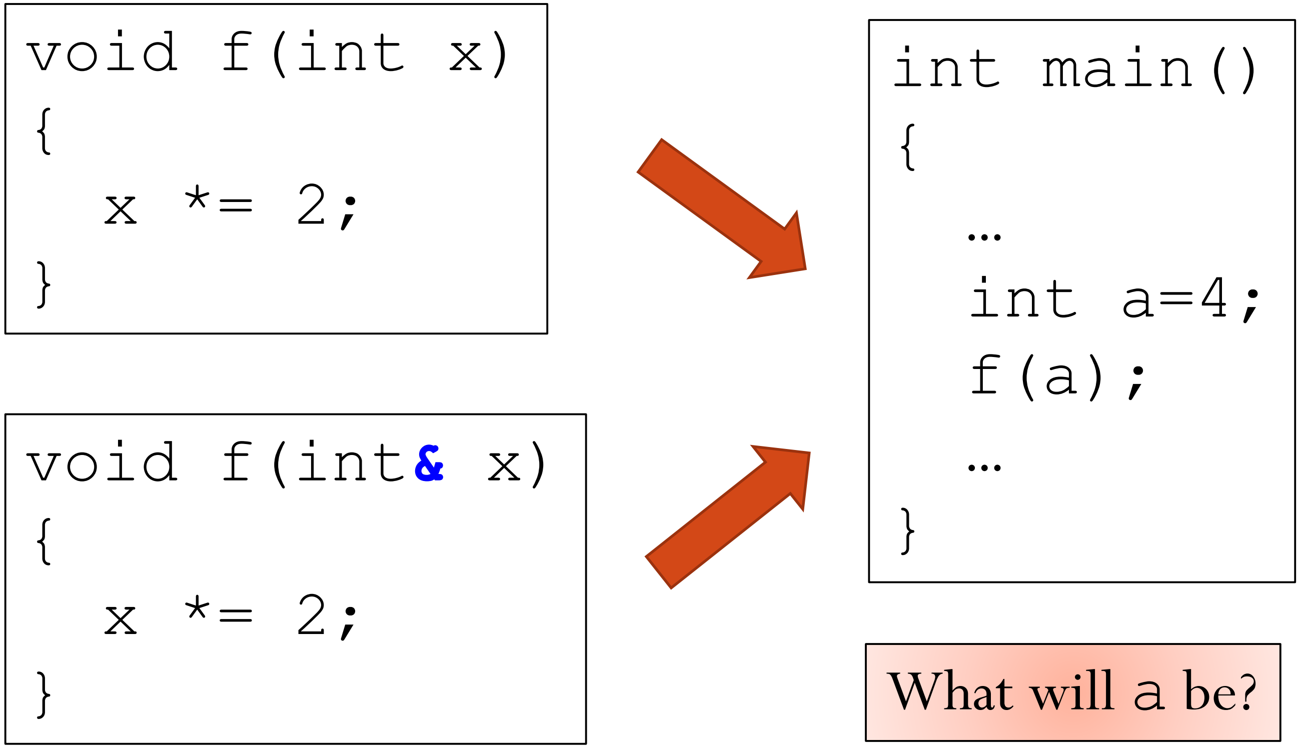
function header

```
double total_cost(int number, double price)
{
    double TAX_RATE = 0.05; //5% tax
    double subtotal;
    subtotal = price * number;
    return (subtotal + subtotal * TAX_RATE);
}
```

function body

Function Call Mechanisms

- Two mechanisms:
  - Call-by-Value
  - Call-by-Reference



```
void f(int x)
{
   x *= 2;
}
```

```
void f(int& x)
{
   x *= 2;
}
```

```
int main()
{
   …
   int a=4;
   f(a);
   …
}
```

What will a be?

Array

- An array is a fixed-sized, indexed data type that stores a collection of items, all of the same type.
- Declaration: `int b[4];`
- Accessing array elements using index: `b[i]`
- C++ arrays can be passed as arguments to a function.

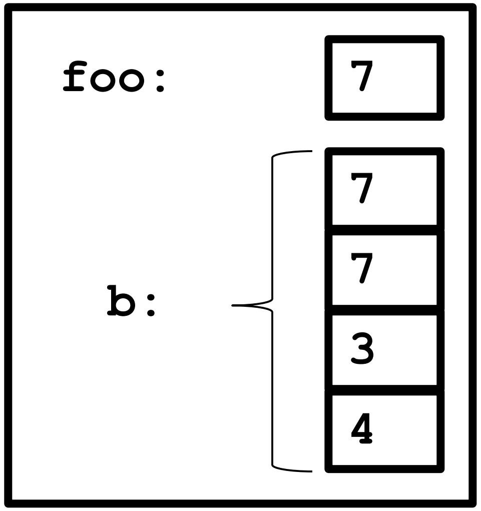
```
int sum(int a[], unsigned int size);
  // Returns the sum of the first
  // size elements of array a[]
```

Array is passed by **reference**.

Array as Function Argument

- Using the values below, what would the contents of `b` be after calling `add_one(b, 4)`?

```
void add_one(int a[], unsigned int size) {
  unsigned int i;
  for (i=0; i<size; i++) {
    a[i]++;
  }
}
```



Reference

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