Chapter 8: Technicalities: Functions, etc. (continues)

Plan for today

- We will talk about:
  - Declarations and definitions
  - More about scope
  - Passing arguments by value
  - Passing argumetns by const reference
  - Passing arguments by reference

Declarations and definitions

- Recall from our previous lecture on functions that
  - A *declaration* is a statement that introduces a name into a scope
    - specifying a type for what is named (e.g. a variable or a function)
    - optionally, specifying an initializer (e.g. initializer value, or a function body)
  - A declaration that also fully specifies the entity declared is called a *definition*
  - Every *definition* (by definition) is also a *declaration*.

Declarations and definitions

- For a *variable*, a *declaration* supplies the type, but only *definition* supplies the object (memory)
- For a *function*, a *declaration* provides the type (arguments + return type), but only the *definition* provides the function body (executable statements)
  - Note that function bodies are store in memory as a part of the program, so it is fair to say that function and variable definitions consume memory, whereas declarations do not.

Declarations and definitions: examples

- A declaration that (also) fully specifies the entity declared is called a definition
  - Examples

```
int a = 7;
int b;                               // an (uninitialized) int
vector<double> v;                 // an empty vector of doubles
double sqrt(double) { … };     // a function with a body
struct Point { int x; int y; };
```

  - Examples of declarations that are not definitions:

```
double sqrt(double);   // function body missing
struct Point;  // class members specified elsewhere
extern int a;  // extern means "not definition"
         // "extern" is archaic; we will hardly use it
```

Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- Often we want the definition “elsewhere”
  - Later in a file
  - In another file
    - preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - To libraries
    - Libraries are key: we can’t write all ourselves, and wouldn’t want to
- In larger programs
  - Place all declarations in header files to ease sharing

Kinds of declarations

- The most interesting are
  - Variables
    - `int x;`
    - `vector<int> vi2 {1,2,3,4};`
  - Constants
    - `void f(const X&);`
    - `constexpr int i{2};`
  - Functions
    - `double sqrt(double d)  { /* … */ }`
- Namespaces (will talk about later today)
- Types (classes and enumerations; see Chapter 9)
- Templates (see Chapter 19)

Header Files and the Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components.
- #include "std_lib_facilities.h"

  is a “preprocessor directive” that adds declarations to our program

  - Typically, the header file is simply a text (source code) file

Header Files and the Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components.
- #include "std_lib_facilities.h"

  is a “preprocessor directive” that adds declarations to our program
  - Typically, the header file is simply a text (source code) file
- A header gives us access to functions, types, etc. that we want to use in our programs.
  - Usually, we don’t really care about how they are written.
  - The actual functions, types, etc. are defined in other source code files
    - Often as part of libraries

Scope

- A *scope* is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between { … } braces)
  - Statement scope (e.g. in a for-statement)
- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name (“can’t look ahead” rule)
  - Class members can be used within the class before they are declared

Scope

- A scope keeps "things" local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have many thousands of entities
  - Locality is good!
    - Keep names as local as possible

Scope : examples

```
void f(int x)     // f is global, x is local to f
{
   int z = x+7;   // z is local
}

int g(int x)      // g is global, x is local to g
{
   int f = x+2;   // f is local
   return 2*f;
}
```

Scope : examples

```
int x;  // global variable – avoid those where you can
int y;  // another global variable

int f()
{
 int x; // local variable (Note – now there are two x’s)
 x = 7; // local x, not the global x
 {
   int x = y;// another local x, initialized by the global y
             // (Now there are three x’s)
   ++x;     // increment the local x in this scope
 }
}
// avoid such complicated nesting and hiding: keep it simple!
```

Scope : examples

```
#include "std_lib_facilities.h" // get max and abs from here
// no r, i, or v here
class My_vector {
   vector<int> v; // v is in class scope
public:
   int largest()  // largest is in class scope
   {
   int r = 0;          // r is local
   for (int i = 0; i<v.size(); ++i)   // i is in statement scope
   r = max(r,abs(v[i]));
   // no i here
   return r;
   }
   // no r here
};
// no v here
```

Recap: Why functions?

- Chop a program into manageable pieces
  - “divide and conquer”
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug

Functions

- General form:
  - return_type *name* (formal arguments); // a declaration
  - return_type *name* (formal arguments) body // a definition
  - For example

```
double f(int a, double d) { return a*d; }
```

- Formal arguments are often called parameters/formal parameters
- If you don't want to return a value give void as the return type

```
void increase_power_to(int level);
```

void means "doesn't return a value"

Functions

- A body is a block or a try block
  - For example:

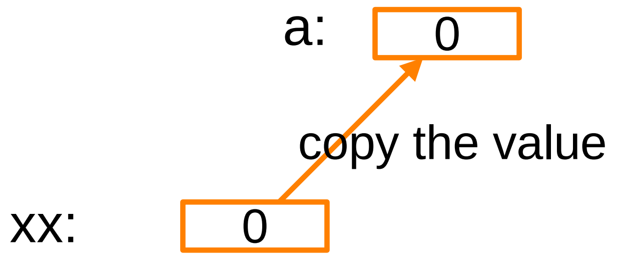
```
{ /* code */ }   // a block
```

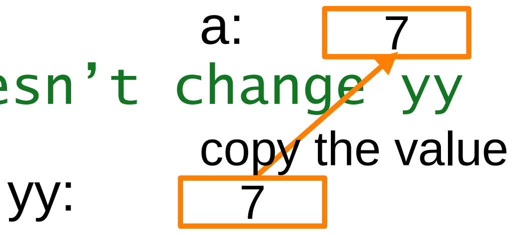
```
try // a try block
{
   /* code */
 }
catch(exception& e) { /* code */ }
```

- Functions represent/implement computations/calculations

Functions: Call by Value

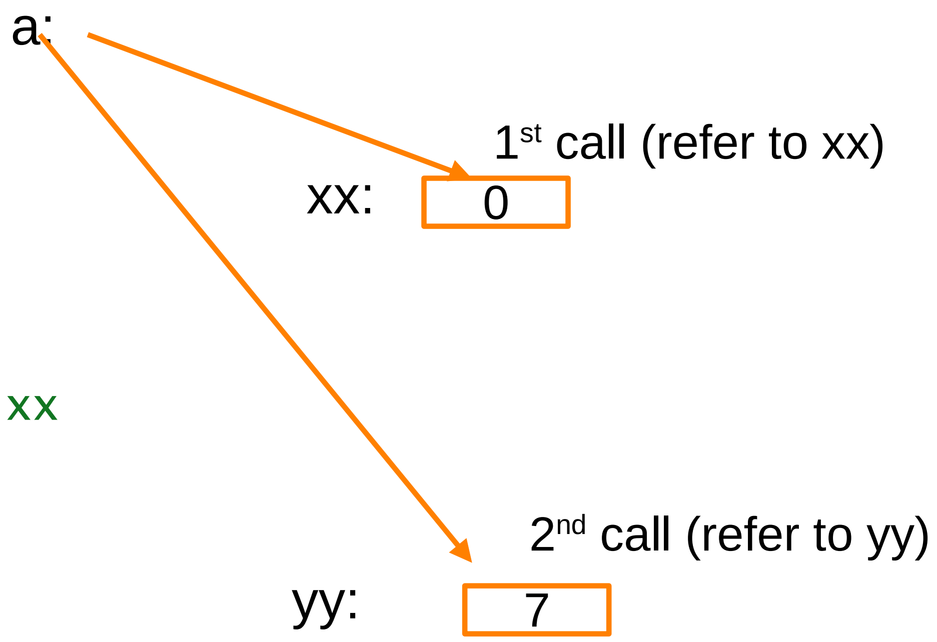
```
// call-by-value (send the function a copy of the argument's
value)
int f(int a) {
    a = a+1;
    return a; }
int main()
{
   int xx = 0;
   cout << f(xx) << '\n';    // writes 1
   cout << xx << '\n';       // writes 0; f() doesn't change xx
   int yy = 7;
   cout << f(yy) << '\n'; // writes 8;  f() doesn't change yy
   cout << yy << '\n';       // writes 7
}
```





Functions: Call by Reference

```
// call-by-reference (pass a reference to the argument)
int f(int& a) {
   a = a+1;
   return a; }
int main()
{
  int xx = 0;
  cout << f(xx) << '\n';  // writes 1
        // f() changed the value of xx
  cout << xx << '\n';   // writes 1
  int yy = 7;
  cout << f(yy) << '\n'; // writes 8
        // f() changes the value of yy
  cout << yy << '\n';   // writes 8
}
```

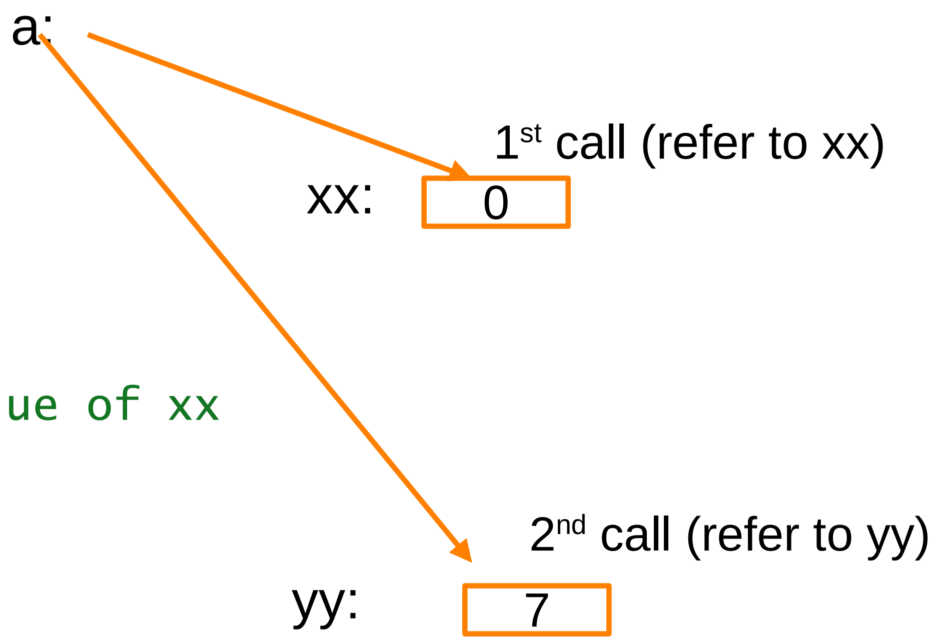


Functions: Call by Reference, with const

```
// call-by-reference (pass a reference to the argument)
int f(const int& a) {
   a = a+1;   // error, not allowed
   return a; }
int main()
{
   int xx = 0;
   cout << f(xx) << ’\n’;   // writes 1
          // f() changed the value of xx
   cout << xx << ’\n’;   // writes 1
   int yy = 7;
   cout << f(yy) << ’\n’; // writes 8
          // f() changes the value of yy
   cout << yy << ’\n’;   // writes 8
}
```

Functions: Call by Reference, with const

```
// call-by-reference (pass a reference to the argument)
int f(const int& a) {
   //a = a+1;
   return a+1; }
int main()
{
   int xx = 0;
   cout << f(xx) << '\n';  // writes 1
          // f() didn't change the value of xx
   cout << xx << '\n';  // writes 0
   int yy = 7;
   cout << f(yy) << '\n'; // writes 8
          // f() didn't change the value of yy
   cout << yy << '\n';  // writes 7
}
```



Function calls

- Why reference arguments?
  - No time spent on copying values
  - No space is used for copies of values
  - Can manipulate containers (like vector)
  - Can remedy the limitation of C++ to return only one value by updating the values of provided by reference variables
  - Remember: real programs have many thousands of entities
- Avoid non-const reference arguments when you do not plan to change/update their values

Function calls: some guidelines

- Use *pass-by-value* to pass very small object
- Use *pass-by-const-reference* to pass large objects that you don't need to modify
- Return a result rather than modifying the object through a reference argument (when possible)
- Use *pass-by-reference* only when you have to, i.e. for functions that need to change several objects

Example and in-class work

- Let’s take a look at a few examples I have in the file Example.cpp
- Then let’s grab the suggested code for the statistics program and incorporate a few suggestions into it:
  - We want to keep sorting, finding minimum, maximum, mean and median as one function, but we want to be able to return the smallest, the largest, the mean and the median back to the caller. What can we do?
  - We would also like to make “input” as a separate function.

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook