

COMP6771

Advanced C++ Programming

Week 1

Part 3: Scope, I/O and Functions

2017

www.cse.unsw.edu.au/~cs6771

Functions

Functions allow us to define our own operations. They should be used to structure code into meaningful and reusable chunks. Generally a function should be self-contained and perform a single logical operation.

A function has:

- a declaration (prototype) and a definition
- a name
- a return type
- zero or more formal parameters

Formal vs Actual Parameters

The *formal parameters* are those that appear in the function definition, specifying the type and the name of each parameter, while *actual parameters* are those used when calling the function.

Functions

A function must specify a return type, which may be void.

Functions may not return arrays or other functions!

```
foo( etc ); // error: no return type  
void bar( etc ); // OK
```

A function may have an empty parameter list:

```
int foo(); // OK: implicit void parameter list  
void bar(void); // OK: explicit void parameter list
```

Call by Value

The actual argument is copied into a location being used to hold the formal parameter's value during method/function execution.

```
1  #include <iostream>
2
3  void swap(int y, int x) {
4      int tmp;
5      tmp = x;
6      x = y;
7      y = tmp;
8  }
9
10 int main() {
11     int i = 1, j = 2;
12     std::cout << i << " " << j << std::endl;
13     swap(i, j);
14     std::cout << i << " " << j << std::endl;
15 }
```

Note: is there a problem with this code?

Call by Value

The actual argument is copied into a location being used to hold the formal parameter's value during method/function execution.

```
1  #include <iostream>
2
3  void swap(int *x, int *y) {
4      int tmp;
5      tmp = *x;
6      *x = *y;
7      *y = tmp;
8  }
9
10 int main() {
11     int i = 1, j = 2;
12     std::cout << i << " " << j << std::endl;
13     swap(&i, &j);
14     std::cout << i << " " << j << std::endl;
15 }
```

Call by Reference

The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.

```
1  #include <iostream>
2
3  void swap(int &x, int &y) {
4      int tmp;
5      tmp = x;
6      x = y;
7      y = tmp;
8  }
9
10 int main() {
11     int i = 1, j = 2;
12     std::cout << i << " " << j << std::endl;
13     swap(i, j);
14     std::cout << i << " " << j << std::endl;
15 }
```

Parameter Passing

When is passing arguments by-value not a good idea?

- when the function changes the value of the argument
- when the argument is a large object
- when the parameter type has no copy operation

So what do we do? We use pointers (C style) or references (C++ style). For example, to swap two integers:

```
void swap(int i, int j);    // 1st-year style
void swap(int *i, int *j); // C style
void swap(int &i, int &j);  // C++ style
```

Of course, you should use the C++ style wherever possible!

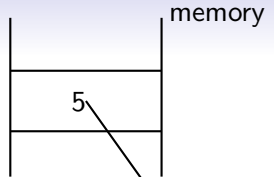
Lvalue vs Rvalue

- Consider

```
int i = 5;  
i = i + 1;
```

lvalue
(memory address)

200



add the **rvalue** 5 and 1 and
store 6 into **lvalue** 0x2000

- The concepts apply to class objects as well:

```
vector<int> v;
```

rvalue: the content in the vector

lvalue: address of the vector, i.e., &v

lvalues & rvalues

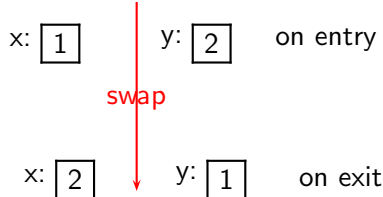
As the name suggests, rvalues may only appear on the RHS of an assignment, while lvalues may appear on both the LHS and RHS. But these concepts are more complex in C++ as per later lectures.

Parameter Passing

- **Call by value:**
 - The **rvalue** of an actual argument is passed
 - Cannot access/modify the actual argument in the callee
- **Call by reference:**
 - The **lvalue** of an actual argument is passed
 - May access/modify directly the actual argument
 - Eliminates the overhead of passing a large object
- Java and C: call by value
- C++: call by value and call by reference

Call by Value

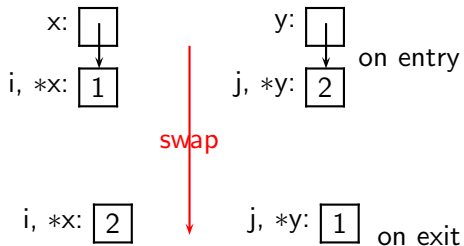
```
#include <iostream>
void swap(int y, int x) {  x: [1]      y: [2]      on entry
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
int main() {              i: [1]      j: [2]
    int i = 1, j = 2;
    std::cout << i << " " << j << std::endl;
    swap(i, j);
    std::cout << i << " " << j << std::endl;
}
```



The diagram illustrates the call by value mechanism. A red arrow labeled "swap" points from the initial state to the final state. The initial state shows variables x and y with values 1 and 2 respectively, labeled "on entry". The final state shows variables x and y with values 2 and 1 respectively, labeled "on exit".

Call by Value Using Pointers

```
#include <iostream>
void swap(int *x, int *y) {
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
int main() {
    int i = 1, j = 2;
    std::cout << i << " " << j << std::endl;
    swap(&i, &j);
    std::cout << i << " " << j << std::endl;
}
```



**x is an alias of i; *y is an alias of j*

Call by Reference

```
#include <iostream>
void swap(int &x, int &y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
int main() {
    int i = 1, j = 2;
    std::cout << i << " " << j << std::endl;
    swap(i, j);
    std::cout << i << " " << j << std::endl;
}
```

on entry

swap

on exit

x is an alias of i; y is an alias of j

- Can think of a reference as a dereferenced pointer
- References are almost always implemented as pointers

Return Values

Pointer and Reference Return Types

Functions may return pointers or references, but make sure these are not to local objects!

```
1 int* f() {  
2     int local = 5;  
3     return &local;  
4 }  
5 // get a pointer to a local variable!  
6 int *getlocal = f();  
7 std::cout << *getlocal << std::endl;
```

Pointer and Reference Return Types

This function will compile ok, but fail at run time! When the function ends, the storage in which local variables reside is freed. The return value points to memory that is no longer available to the program.

Default Values

Functions in C++ may use default arguments, which is used if an actual value is not specified when the function is called.

```
string rgb(short r = 0, short g = 0, short b = 0);  
rgb();           // rgb(0, 0, 0);  
rgb(100);        // rgb(100, 0, 0);  
rgb(100, 200);   // rgb(100, 200, 0)  
rgb(100, , 200); // error
```

NB

Default values are used for the *trailing* parameters of a function call! Order your function parameters carefully!

```
void f(int x = 0, int y, int z = 0); // error
```

Function Overloading

Function overloading refers to a family of functions, in the **same scope**, that have the **same name** (and hopefully perform a similar operation!) but **different formal parameters**. Function overloading can, and should, make code easier to write, understand and use.

```
void print(double d);      // (1)
int  print(string s);      // (2)
void print(char c);        // (3)

print(3.14);               // call (1)
print("hello World!");     // call (2)
print('A');                 // call (3)
```

Function Overloading

The process by which the compiler chooses the correct version of an overloaded function for execution is known as *overload resolution* (or *function matching*). A good program design should make overload resolution straightforward.

A three-step process:

- 1 identifying the candidate functions
- 2 selecting the viable functions
- 3 finding a best-match

If any step fails then a compile-time error occurs.

Unexpected results are produced by subtleties resulting from automatic type conversions, promotions, and so on.

Read §6.4 and §6.6 about some of these issues.

An Example on Overloading Resolution

```
void g();  
void f(int);  
void f(int, int);  
void f(double, double = 3.14);  
f(5.6); // calls f(double, double)
```

- ❶ **Candidates:** same name visible at the point of call:

```
void f(int);  
void f(int, int);  
void f(double, double);
```

- ❷ **Viable:** same number of paras as arguments and the type of each argument is convertible to its corresponding paras

```
void f(int);  
void f(double, double);
```

- ❸ **Best Match:** the type match no worse in each argument but better in at least one argument

```
void f(double, double);
```

Function Overloading

Due to **call by value**, top-level const has no effect on the objects passed to the function. A parameter that has a top-level const is indistinguishable from the one without.

- Top-level const ignored

```
// both cannot modify a phone object passed in
Record lookup(Phone p) { ... };
Record lookup(const Phone p) { ... }; // a redefinition
```

- Low-level const is significant

```
// one can but the other cannot
Record lookup(Phone &p) { ... };           // (1)
Record lookup(const Phone &p) { ... };     // (2)
```

```
Phone p1;
const Phone p2;
lookup(p1); // call (1)
lookup(p2); // call (2)
```

Inline Functions

Sometimes it is convenient to define small utility functions for simple operations, for example:

```
int min(int n, int m) {  
    return (n < m) ? n : m;  
}
```

But we would also wish to avoid the overhead of a function call for such a simple function.

C++ allows us to do this by specifying the `inline` keyword:

```
inline int min(int n, int m) { etc...
```

NB

Inline functions should be defined in header files! Why?

Inline Functions

A function specified as inline is expanded "in line" at each point in the program in which it is invoked, for example:

```
std::cout << min(i, j) << std::endl;
```

would be expanded during compilation into something like:

```
std::cout << ((n < m) ? n : m) << std::endl;
```

Of course, the compiler must have access to the function definition in the header file in order to expand.