Object-Oriented Principles and Practice / C++

Alice E. Fischer

September 26, 2016
Global vs. Class

Static

Parameters

Move Semantics
Global Functions

Global functions do not have implicit arguments. They have only the explicit arguments that you see in the prototypes.

`<<` is a global operator whose first argument is an `ostream&`

That is why you cannot define a method for `<<` inside a class. Every function defined in a class has an implicit parameter, of the class type, and `<<` does not.

For the same reason, you cannot define the usual 2-argument `swap` function inside a class.
The implicit argument

Every class member function has an *implicit argument* in addition to the explicit arguments shown in the prototype.

When a class function is called, the implicit argument is the object whose name is written before the dot in the function call.

Class functions can see and modify the implicit argument.
Implicit argument example

```cpp
class MyExample {
private:
    int count; // data member
public:
    void advance(int n) { count += n; }
    ...
};
```

In this example, the function `advance` has two arguments, the integer `n` and the implicit argument `ex`.

```cpp
MyExample ex;
ex.advance(3); // Add 3 to ex.count
```
this: Chapter 10.1

The implicit argument is passed by pointer.

In the call `ex.advance(3)`, the implicit argument is `ex`, and a pointer to `ex` is passed to `advance()`.

The implicit argument can be referenced explicitly from within a member function using the keyword `this`.

Within the definition of `advance()`, `count` and `this->count` are synonymous.

There is rarely a need to use `this`. Almost always, there is a better way to access the implicit argument.
Static Class Members

Chapter 6.2
Static Class Functions
Static Data Members
Static means “stays in memory”.

- Anything declared to be static is created and initialized at load time and ready to use as soon as the program starts execution.
- Static objects remain available, and in the same memory locations, until the program terminates.
- All kinds things can be static: class functions, class data members, and local variables inside functions.
- However, static does NOT mean the same thing as “global”. Global objects are visible everywhere in the program. Static objects are visible ONLY within their proper, defined scope.
Static != Global

A static object is visible only within its proper, defined scope.

- A static local variable inside a function is visible only within that function.
- A public static class member is visible everywhere.
- Functions are sometimes declared public static.
- A private static class member is visible only to functions in the same class.
- Data members are sometimes private static.
Static Class Functions

Section 6.2
A static class function can be called before the first class object is created.

- It can have explicit parameters, but it does not have an implied parameter. (It does not have a "this" pointer.)
- So it cannot use the non-static class members.
- Such functions have very limited usefulness. I have used them for instructions and data validation prior to calling a constructor.
Static Data Members

Section 6.2
A static data member becomes part of every class object.

► It can be used to share information between all class instances.
► For example, an integer counter could be made static. If the constructor increments it, then it tells us how many class objects have ever been created by that constructor.
► Static data members are called class variables, while non-static data members are called instance variables.
Parameters

Parameter Passing Mechanisms
  Call by value
  Call by pointer
  Call by reference
  Call by r-value reference

Choosing Parameter Types
Parameter Passing Mechanisms

Chapter 5, Section 2

- Call by value: the argument value is copied into the parameter variable.
- Call by pointer: the pointer-value is copied into the pointer-parameter.
- Call by reference: the address of the argument is copied into the reference-parameter. At the hardware level, this is like passing a pointer. However, the meaning is different.
- Call by r-value reference: the address of the argument is copied into the & & parameter. This is like passing a & parameter. However, inside the compiler, it is a different type and that allows us more control over the meaning.
Call by value

Like C, C++ passes parameters by value, unless otherwise specified by the function prototype.

```cpp
void f( int y ) { ... y=4; ... };
...
int x=3;
f(x);
```

- x and y are independent variables.
- y is created when f is called and destroyed when it returns.
- At the call, the value of x (=3) is used to initialize y.
- The assignment y=4; inside of f has no effect on x.
Call by pointer

Like C, pointer-R-values (which I call reference values) are the things that can be stored in pointer variables and passed as arguments to functions having corresponding pointer parameters.

```cpp
void g( int* p ) { ... (*p)=4; ... };

...  
int x=3;
g(&x);
```

- p is created when g is called and destroyed when it returns.
- At the call, the value of &x, a reference value, is used to initialize p.
- The assignment (*p)=4; inside of g changes the value of x.
Call by reference

```cpp
void g( int& p ) { ... p=4; ... };
...
int x=3;
g(x);
```

- This has the same thing as using a pointer parameter.
  However, to use a reference parameter, we write `&` not `*` or `->`
- Within the body of `g`, `p` is a synonym or alias for `x`. So the assignment `p=4` changes the value of `x`.
- For example, `&p` and `&x` are identical reference values. `p=4` changes the value of `x`.

Reference parameters **must be used** when the argument is a class object with dynamically allocated parts.

Stream parameters are **always** passed by reference.
I/O uses reference parameters

- The first argument to `<<` has type `ostream&` because the `ostream` object has dynamically allocated parts that must not be deallocated after each output operation.
- The first arguments has type `istream&` instead of `const istream&` because the act of doing output changes members of the stream object.
- Similarly, the first argument to `>>` has type `istream&`.
- `<<` returns a reference to its first argument, which permits it to be chained.
- `cout << x << y;` is the same as `(cout << x) << y;`
Three Odd Functions

These functions illustrate the ways in which the parameter-passing mechanisms differ. All three compute the average of the two parameters, then increment the parameters (directly or indirectly).

- odd1 uses call by value, which protects the caller from actions of the called function. The ++ operations increment local copies of the arguments and do not (can not) affect the caller.

- odd2 uses call by pointer; it can do two different kinds of ++. When it increments aa, the pointer is moved to point at the next int in the array. When it increments *bb, the int in main’s array is incremented.

- odd4 the address of the argument is copied into the reference-parameter. At the hardware level, this is like passing a pointer. However, the meaning is different.
Calling the Odd Functions

Chapter 5, page 7

- Call by value: the argument value is copied into the parameter variable.
- Call by pointer: the pointer-value is copied into the pointer-parameter.
- Call by reference: the address of the argument is copied into the reference-parameter. At the hardware level, this is like passing a pointer. However, the meaning is different.

Reference parameters must be used when the argument is a class object with dynamically allocated parts.

Stream parameters are always passed by reference.
How should one choose the parameter type?

Often, in C++, we use class members instead of parameters.

Parameters are used for two main purposes:

- To send data (large or small) to a function.
- To receive data from a function.
Sending data to a function.

For sending data to a function, all three major parameter-passing mechanisms work.

- Call by value protects the caller’s data from being inadvertently changed.
- call-by-value copies the data whereas call-by-pointer or call-by-reference copies only an address.
- If the data object is small (e.g., an `int` or `double`), call-by-value is better since it avoids an indirect reference on every use.
- If the data object is large, call-by-value is expensive in both copying-time and space and should be avoided.
Passing an address.

- Call by reference or pointer copies the address of the argument into the parameter variable, so it costs little space and little copying time.
- However, every use causes a second trip to memory.
- Call by reference or pointer allows the caller’s data to be changed. Using `const` protects the caller’s data from inadvertent change. Example:

  ```cpp
  int func( const T& x ) or int gunk( const T* p ).
  ```
Reference vs. Pointer

- Use call-by-value for small arguments (1 – 8 bytes).
- Use call-by-pointer when working with linked data structures (trees, lists, graphs).
- Otherwise, for large objects, prefer call-by-reference over call-by-pointer.
- When the size and nature of the parameter is unknown, use call-by-reference because it works with more kinds of arguments. For example: `func( 234 )` works but `gunk( &234 )` does not. Reason: 234 is not a variable and hence can not be passed to a pointer parameter.
Receiving data from a function

An output parameter is one that should be changed by the function.

Both call-by-reference and call-by-pointer work for output parameters.

Declaration: int f( int& x ) or int g( int* p ).

Calls: f( result ) or g( &result ).

Call by reference is generally preferred since it is easier to type and easier to read.
Returning Pointers and References

Section 5.3
The return value of a function can be a value, a pointer, or a reference.

▶ A non-const pointer or reference permits the caller to store data in the address returned by the function.

▶ A const-pointer or const-reference return value gives read-only access to the result.

▶ It looks a little strange to put a function call on the left side of an assignment, but it works.

▶ `operator[]` returns a non-const reference. The result can be used to fetch or store data in the array.
RValue References

Default Class Operations: C++ 11
Control of Defaults: C++ 11
RValue References

In addition to values, pointer, and references, C++-11 introduced rvalue-references as a fourth “flavor” that any type can have.

- The purpose of this fourth syntax appears to be giving us more options about how operations should be done.
- An rvalue-reference is used to implement move semantics in addition to assignment semantics.
- When you assign an object your copy it, and the value then exists in two locations.
- When you move a thing, it is no longer in the old location.
RValue References

An rvalue reference can bind to an rvalue (but not to an lvalue):

- \( X \) \ a;
  \( X& \ r1 = a; \) // bind r1 to a (an L-value)
  \( X&& \ rr1 = f(); \) // bind rr1 to the result of f()
  \( X&& \ rr2 = a; \) // error: a is an L-value

- This idea can be used to speed up execution of any operation that moves values around.

- If \( X \) is a type for which copying is expensive (string, vector) a simple swap becomes an expensive operation.
RValue Swapping

But when we swap, we don’t really want new copies at all. We just want to rebind the existing copies.

```cpp
void swap(T& a, T& b) {
    T tmp = move(a);    // Save a’s value.
    a = move(b);        // Move b’s value to a.
    b = move(tmp);      // Move the saved value to b.
}
```

- Moving is faster than copying for many types because it does not construct a new object.
- This is very important when you design a template because the template parameter could be any primitive or class type.
Default Class Operations: C++ 11

By default, a class has these five related operations:

- **destructor**: By default, a null destructor.

- **copy constructor**: `X(const X&)`
  Initialize a new object from an old one of the same type. The contents of the source are unchanged. By default, it is a shallow copy.

- **copy assignment**: `Y& operator=(const Y&)`
  Copy one object into another of the same type. The contents of the source are unchanged. By default, it is a shallow copy.
Move Operations: C++ 11

By default, a class has these five related operations:

- move constructor: `X(const X&&)`
  Initialize a new object by moving the contents of an old object into it. After the move, the contents of the source are set to the initial constructed state.

- move assignment: `Y& operator=(Y&&);`
  Move one object into another. After the move, any pointers in the source should be set to nullptr.
Default Class Operations: C++ 11

Copy, assignment, move, and delete are closely related operations.

- You can redefine all of them,
- But only a few combinations make sense.
- If you declare any of them you must explicitly define or default all the others.
- If you define any one, movers will not be generated automatically. Copiers will be generated automatically, but this is deprecated.
- Move constructor and move assignment take non-const &&. They can, and usually do, write to their argument
Control of Defaults: C++ 11

The keywords `delete` and `default` and can be used to define methods.

```cpp
▷ class X {
    // These definitions disallow copying.
    X& operator=(const X&) = delete;
    X(const X&) = delete;
};
▷ class X {
    // These define the default copy behavior.
    X& operator=(const X&) = default;
    X(const X&) = default;
};
```