Object-Oriented Principles and Practice / C++

Alice E. Fischer

April 27, 2015

Parameter Passing
- Choosing Parameter Types
- Parameter Passing Examples
- The Implicit Argument

Static Class Members
- Static Class Functions
- Static Data Members

Parameters

Parameter Passing Mechanisms
Call by value
Call by pointer
Call by 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.

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

Stream parameters are always passed by reference.

Call by value

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

```
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.

```
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

C++ has a new kind of parameter called a *reference* parameter.

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

- This does same thing as previous example; namely, the assignment `p=4` changes the value of `x`.
- Within the body of `g`, `p` is a synonym or alias for `x`.
- For example, `&p` and `&x` are *identical* reference values.

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 di 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: call by value

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

- call-by-value copies the data whereas call-by-pointer or call-by-reference copies only an address.
- If the data object is large, call-by-value is expensive of both time and space and should be avoided.
- If the data object is small (eg., an `int` or `double`), call-by-value is cheaper since it avoids the indirection of a reference.
- Call by value protects the caller's data from being inadvertently changed.

Using call-by-const-reference or const-pointer.

Call by reference or pointer copies only an address into the function, so it costs little space and little time.

- However, it allows the caller's data to be changed.
- Use `const` to protect the caller's data from inadvertent change. Example:
  `int func( const T& x )` or `int gunk( const T* p )`.
- *Prefer call by value* for small-sized input parameters.
- For large objects, *prefer call-by-reference* over call-by-pointer. because call-by-reference will work 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 avoids the need for the caller to place an ampersand in the call in front of the variable name.

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

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

```
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`.

```
MyExample ex;
ex.advance(3);  // Add 3 to ex.count
```

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

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.