Chapter 7: C++ Bells and Whistles

A number of C++ features are introduced in this chapter: default parameters, const class members, and operator extensions.

7.1 Optional Parameters

Purpose and Rules. Default parameters are never necessary but they are a great convenience. They allow us to write a single function definition that defines two or more function variations with different numbers of parameters. A function with one default parameter is really two functions that share a name. (This is sometimes called overloading a name.) A function with several default parameters is a shorthand for several function definitions. The basic requirements are:

- A default value for a parameter is given in the class declaration, not the function definition. It is written (like an initializer) after the parameter name in the function prototype.

  The default values may either be given or omitted in the function definitions that are written in a separate .cpp file. However, if the function definition *does* gives a default parameter value, it must match the one given in the class declaration.
- Some or all of the parameters of any function may have default values.
- All parameters with default values must come after the parameters without defaults.
- When a function is called, the programmer may omit writing arguments if the default values are acceptable.
- Documentation should always explain clearly which parameters have default values and what the values are.

The most familiar function defined with optional arguments is istream::get(). It can be called to read a string with either three or two arguments:

```
cin.get( input, MAX, ',' );
cin.get( input, MAX );
```

The calls both read characters into an array named input. Both stop after MAX-1 characters have been read. The first call will also stop if a comma is read; the second call will stop when the default character (newline) is read. The corresponding prototype of istream::get() might look something like this:

```
void cin.get( char* buf, int max, char c='\n' );
```

7.1.1 A Short Example

Consider the short program that follows. It constructs and prints three instances of a Box class, each with a different number of parameters. The definition of the Box constructor makes this possible by using default parameters.

- By using default parameters, we can replace two or more function definitions by one (line 26). An earlier version of this program had two constructors:

  ```
  Box( int ln, int wd, int ht ){ length=ln; width=wd; high=ht; }
  Box(){ length = width = high = 1; }
  ```
- This single definition is equivalent to a family of four function methods, and allows us to omit the final zero, one, two, or three arguments when calling the constructor:

```
Box( int ln, int wd, int ht );
Box( int ln, int wd );
Box( int ln );
Box( );
```

```
 1  //---------------------------------------------------------------------------
 2  // Array construction example, October 8, 2000                 file: BoxM.cpp
 3  //---------------------------------------------------------------------------
 4  #include "tools.hpp"
 5  #include "box.hpp"
 6  //---------------------------------------------------------------------------
 7  int main( void )
 8  {   cout << "\nTesting the default parameters\n";
 9      Box B1;             // Make a default Box.
10      Box B2(2);          // Make Boxes with various parameters.
11      Box B3(2, 3);
12      cout <<"\nDumping the boxes:\n" << B1 << B2 << B3 <<endl;
13  }

14  //---------------------------------------------------------------------------
15  // Demonstrate syntax and usage of default parameters.
16  // A. Fischer February 10, 2009                                  file: box.hpp
17  //---------------------------------------------------------------------------
18  #pragma once;
19  #include "tools.hpp"
20  //---------------------------------------------------------------------------
21  class Box {
22    private:
23      int length, width, high;    // The three dimensions of a box.
24
25    public:
26      Box(int ln=1, int wd=1, int ht=1){
27          length=ln; width=wd; high=ht;
28      }
29
30      ~Box() {}
31
32      ostream& print( ostream& out = cout ){
33          return out <<length <<" by " <<width <<" by " <<high <<".  ";
34      }
35  };
36
37  //---------------------------------------------------------------------------
38  inline ostream& operator<< (ostream& out, Box& B) { return B.print( out ); }
39
```

Usage. Clearly, optional parameters are useful in constructors. They also save time and writing whenever a function must sometimes handle an unusual case. The parameter connected to the unusual situation is coded last and omitted from most calls. For example, suppose you wanted a print function to (normally) write output to `cout`, but occasionally you want to write to a file. You can accomplish this with an optional parameter. Suppose B is a Box object and file_out is an open output stream. Then you could call the Box::print function (line 32) three ways:

```
B.print( cout );     // Write to the screen, but we do not need the parameter.
B.print();           // Means the same thing as the line above.
B.print( file_out ); // Write to an output file.
```

7.2 Const Qualifiers

Purpose and usage. The proper use of const can be a powerful documentation technique. When you are using classes defined by someone else, you often see only the declarations and prototypes of the public members of the class. It is very important to know which of the members are constant and which functions do not modify their parameters. Here are a few rules and guidelines:

- You rarely need to use const. If a program will work with it, it will work without it.
- Use const to explicitly state your intentions: if your intention is that the variable will not change, then you should define it as a const.
- The most common use of const is to restrict the usage of reference and pointer parameters and return values, so that they give read-only access to another program unit.
- In class functions, const can also be used with the implied parameter; in this case it declares that no statement in the function changes the value of any member of `*this`.
- The other common use of const is at the beginning of a function, to declare a local variables whose value depends on one of the parameters, but whose value will be constant throughout the execution of one call on that function.
- If they are used at all, global variables should be const. It is not bad style to use a global const variable because you can do things with it that you cannot do with #define (initialize a constant object of a structured type).

Finally, be aware that compilers vary with respect to const. It is the job of the compiler (not the programmer) to make sure that the const rules are never broken. Writing the code that enforces the const declarations is difficult. Some compilers enforce the constant rules very strictly; others do not, so your program might compile without errors on a sloppy compiler but fail on a compiler that checks the const rules carefully.

Syntax for const. Everywhere a prototype contains the keyword const, the matching function definition must have a matching const. Here are the rules that apply to writing const in a function prototype and definition. There are four places the keyword const can appear:

1. The const can be written before the function's return type. This is used only when the value returned by the function is a * or an & type.
   ```
   const int* f( int x );  // A pointer to a constant integer.
   const int& g( int x );  // A reference to a constant integer
   ```
   The return value of f() points at a constant integer (possibly part of an array). It can be stored in a non-constant pointer variable. You could increment that pointer to point at the next element in an array of constant ints, but you could not change any of them.

   The return value from g() is a reference to a constant int. It gives read access but not write access to that integer.
2. The const can be written before the type of one of the parameters in the parameter list. This is used only when the parameter is a * or a &. It means that the function cannot modify the caller's variable that corresponds to that parameter.
   ```
   void List::Search( const Item& s );
   void List::Search( const char* name );
   ```
   Here, we declare that the Search function will not modify the Item (in the first prototype) or the string (in the second prototype) that is its argument. Class objects are usually passed by reference to avoid triggering the class destructor when the function exits. In this case, the const plays an important role in isolating the caller from the actions of the function.
3. The const can be written between the end of the parameter list and the ; that ends the prototype or the { that opens the function body. In this position, const means that the function does not modify the implied parameter.
   ```
   void Stack::Print() const;
   int Stack::Space() const { return Max-N; };
   ```
   In this location, the const is used as a form of documentation. It declares that the Print and Space functions use the members of the Stack class but do not modify them. This is obvious if you look at the function's code, but can be helpful documentation when only the prototype is available (not code itself).
4. Within the body of a function, const can be applied to any local variable. This is often done when the variable depends on one of the parameters, but does not change within the function.

7.3 Operator Extensions

Purpose and usage. The built-in arithmetic operators in C and C++ are generic; they are defined for all built-in numeric types. One of the purposes of an object-oriented language is to enable the programmer to define new classes and use them in the same ways that the built-in classes are used. Combining these two facts, we see a need to *extend* the built-in generic operators to work on new classes. An operator extension implements a built-in function for a new type in such a way that the intention and character of the original operator are preserved.

The assignment operator and a copy constructor are automatically defined for all types, as member-by-member shallow copies. Both can be redefined, if needed, to do something else; the new definition *overrides* the default definition. This is appropriate when the programmer wants to eliminate the possibility of an accidental use of the built-in definition, and even more appropriate if something useful can be implemented instead.

A third kind of operator definition *overloads* the operator by giving it a new meaning that is only distantly related to its original meaning. For example, the `+=` operator is extended, below, to add a new item to a list of items. This is not at all like numeric addition, so it is an overload of `+=`, not an extension of the original meaning of the operator (add and store back).

C has both binary and unary operators. When a unary operator (such as increment or type cast) is extended, the definition is always given as part of the relevant class. When a binary operator is extended, the definition is normally given as part of the class of the left operand. If that is not possible, it is given globally (outside all classes). A general rule for an operator definition is that the the number of operands and precedence must match the built-in operator. Default arguments are illegal in these definitions.

Below, we consider some contexts in which an operator definition might make sense and give one or more examples of each.

7.3.1 Global Binary Operator Extensions

These are appropriate for extensions of an operator to a non-class type, or for a pre-defined class that cannot be modified. The syntax for this kind of extension prototype is:

```
return-type operator op-symbol (left-op-type, right-op-type );
```

Three extensions of the `<<` operator were defined in the Bargraph program. These are defined globally because we cannot add them to the predefined ostream class. The right operand (second parameter) belongs to the new class in each case.

```
ostream& operator << ( ostream& out, Item& T);
ostream& operator << ( ostream& out, Graph& G);
ostream& operator << ( ostream& out, Row& T);
```

The preferred way to write this kind of extension is as an `inline` function just below the bottom of the class declaration. If the function is too long for inline expansion, then the prototype should be given at the bottom of the .hpp file for the class, and the function definition should be in the main module, after the #include command for the class header and before the beginning of main().

7.3.2 Binary Operator Extensions in a Class.

Binary functions within a class are defined using this prototype syntax:

```
return-type operator op-symbol ( right-op-type );
```

Extending arithmetic and comparison. Suppose we defined a new numeric class such as `Complex`. It would be appropriate to extend all the basic arithmetic and comparison operators for this class, so that complex numbers could be used, like reals, in formulas. Assume that the class Complex has two data members, rp (the real part) and ip (the imaginary part). Several definitions are required to define all of the relevant operators; here are two samples:

```
Complex operator + ( Complex plex ){   // Add two complex numbers.
    return Complex( rp + plex.rp, ip + plex.ip );
}
bool operator == ( Complex plex ){  // Compare two complex numbers.
    return rp == plex.rp && ip == plex.ip;
}
```

In such definitions, the implied argument is the left operand and the explicit argument is the right operand. Any reference to the left operand is made simply by using the part name. References to the right operand are made using the parameter name and the part name.

Adding to a collection: an overload. Another appropriate extension is a += operator in a container class. The Stack::Push() function could be replaced by the following extension. (Only the function name would change, not the function's code.)

```
void Stack::operator += ( char c );    // Push an Item onto the Stack.
```

The Stack::push function is called from three loops in the main program. The first loop could be replaced by this intuitively appealing code:

```
do { cin >> car;
     west += car;
} while (car != QUIT);            // Engine has been entered.
```

Assignment: an override. Sometimes the default definition of assignment is changed. For example, here is a definition of = for Stacks that moves the data from one stack to another, rather than doing a shallow copy. This definition illustrates the use of the keyword `this` to call the += function (defined above) to push a character onto the implied Stack parameter:

```
void operator= ( Stack& z ) {    // Pop one stack, push result onto other.
    char c;
    while (!z.empty()) {
        c = z.pop();
        *this += c;
    }
}
```

Extending subscript. The remaining important use of operator extensions is to define subscript for data structures that function as arrays but have a more complex representation. A good example is the resizeable array data structure that was used to implement the unlimited storage capacity for the stack in the Stack program. A flexible array class needs an extension of the subscript operator so that the client program can refer to the object as if it were a normal array.

The subscript operator is a little different in two ways. First, it must return a reference to the selected array slot so that the client program can either read or write the selected element. Second, the syntax is a little different because the left operand is enclosed between the square brackets, not written after them. An added advantage of extending subscript is that a bounds check can be incorporated into the definition, as illustrated below. Here is the part of the class declaration and the extension of the subscript operator that could be part of a FlexString class (a resizeable array of characters).

```
class FlexString {
  private:
    int Max;                 // Current allocation size.
    int N;                   // Number of data chars stored in array.
    char* Buf;               // Pointer to dynamic array of char.
  public:
    char& operator[]( int k );
    ...
};

// ------------------------------------ access the kth char in the string.
char& FlexString ::  operator []( int k ) {
    if (k >= N) fatal(" FlexString bounds error.");
    return Buf[k];
}
```

7.3.3 Unary operators.

Extending type cast. The syntax for extending the type cast operator is:

operator `target-type()` { **return** `member-name;` }

An extension of the type cast operator is used to convert a value from a class type to another type. This is also called "member conversion", because it converts a class member to some other type. It is particularly useful in linked-list classes such as Cell (below) for converting from a Cell to a pointer type such as Item*, by extracting the pointer from the Cell.

```
class Cell {
  private:
    Item*  Data;
    Cell* Next;
  public:
    operator Item() { return *Data; } // Type cast from Cell to Item
    ...
};
```

Type coercion. Extensions of a type cast operator can be used explicitly, but they are also used by the system to coerce operands and parameters. For example, in a program that contained the above extension of type cast, if you used a Cell object in a context in which a Cell* was needed, the compiler would use the Next field of the object to carry out the operation.

Increment and decrement: extensions or overloads. The Prefix forms of increment and decrement are normal unary operators; The postfix forms are different from everything else in the language. The prototypes for these operators have the form:

Preincrement:	`return-type` **operator** ++();
Postincrement:	`return-type` **operator** ++(**int**)();

The "int" after the ++ in the postfix form serves only one purpose, and that is to distinguish the prefix and postfix forms. The "(int)" looks like a parameter list, but the parameter does not need a name because it is not used; it is just a dummy. Definitions of these two operators inside the complex class might look like this:

```
Complex operator ++ (){ rp++; return *this; }
Complex operator ++ (int){ Complex temp = *this; ++rp; return temp; }
```

Another use of an increment extension is to make the syntax for a linked list imitate the syntax for an array, so that the exact nature of a class implementation can be fully hidden. This technique is used extensively in implementations of container classes in the Standard Template Library, and can be applied in any list class that has a "current element" member. Here is an appropriate extension for a List class that has members named Head, Curr and Prior:

```
Cell* operator ++ (){ Curr = Curr->Next; return Curr; }
Cell* operator ++ (int){ Cell* tmp = Curr; Curr = Curr->Next; return tmp; }
```

Given this definition and an appropriate extension for <= in the Item class, we could write a List::Search function as follows:

```
bool List::Search( Item T ){   // Search for T in list with ascending order.
    for (Curr= Head; Curr!=NULL; )
        if (T <= Curr->Data) break;
        Prior = Curr++;
    }
    return T == Curr->Data     // true for exact match, false for >.
}
```

7.4 Static Operators in a Class

When working with some standard packages, such as `std::sort()` in the STL algorithms package, we sometimes pass a class operator or function as a parameter to another function. In such cases, the class operator must have the exact prototype declared for the standard algorithm.

For example, `std::sort()` has two methods. Both sort an array of arbitrary type, BT. The zero-parameter sort method uses the `operator<`, which must be defined for class BT, using the normal syntax for defining a binary operator inside a class. The other method for `std::sort()` requires a functional parameter which must have exactly the prototype:
`bool funcname ( const BT& a, const BT& b );`
The function must compare the two BT objects and return a true or false result.

Clearly, this should be defined in conjunction with the BT class. Since the comparison operator must use the private parts of the BT object. it must be inside the class. However, a function defined inside the class has an implied argument and must be called using a class object. For `std::sort()`, this does not work.

The conflict can be addressed in various ways, most of them ugly. The non-ugly solution is to define a `static` class function with two parameters that matches the required prototype. Because the function is static, it *lacks an implied parameter*, and thus, satisfies the needs of `std::sort()`. Example: In this class, we define a function that will be used to sort an array of `Freq` objects in descending order, according to the `pct` member.

```
class Freq {
private:
    char letter;
    double pct;
public:
    Freq( char c=' ', double d=0.0 ) { letter = c; pct = d; }
    ~Freq(){}

    void print (ostream& out) { out <<letter <<setw( 7 ) <<pct; }

    bool operator< (const Freq fr) const { return pct < fr.pct; }
    static  bool descend( const Freq& f1, const Freq& f2 ){  return f1.pct > f2.pct; }
};
```

To call this function or pass it as a parameter to another function, use its full name, `Freq::descend`:

```
    Freq array[100];
    int n;  // The number of data items actually stored in the array.
    bool result = Freq::descend( array[1],  array[2] );
    std::sort( array, array+n,  Freq::descend );
};
```