Here is another similar example from Absolute C++ by Walter Savitch that I also use as a reference text…

**Overloading the Array Operator [ ]**

You can overload the square brackets, [], for a class so that they can be used with objects of the class. If you want to use [] in an expression on the left-hand side of an assignment operator, then the operator must be defined to return a reference. When overloading [], the operator [] must be a member function. It may help to review the syntax for the operator [], since it is different from any other operator we have seen. Remember that [] is overloaded as a member operator; therefore one thing in an expression using [] must be the calling object. In the expression a[2], a is the calling object and 2 is the argument to the member operator []. When overloading [], this “index” parameter must be an integer type, that is, enum, char, short, int, long, or an unsigned version of one of these types.

For example, in Display 8.7 we define a class called CharPair whose objects behave like arrays of characters with the two indexes 1 and 2 (not 0 and 1). Note that the expressions a[1] and a[2] behave just like array indexed variables. If you look at the definition of the overloaded operator [], you will see that a reference is returned and that it is a reference to a member variable, not to the entire CharPair object. This is because the member variable is analogous to an indexed variable of an array. When you change a[1] (in the sample code in Display 8.7), you want that to be a change to the member variable first. Note that this gives access to the private member variables to any program, for example, via a[1] and a[2] in the sample main function in Display 8.7. Although first and second are private members, the code is legal because it does not reference first and second by name but indirectly using the names a[1] and a[2]

See attach source code display 8.7

Overloading Based on L-Value versus R-Value

Although we will not be doing it in this book, you can overload a function name (or operator) so that it behaves differently when used as an l-value and when it is used as an r-value. (Recall that an l-value means it can be used on the left-hand side of an assignment statement.) For example, if you want a function f to behave differently depending on whether it is used as an l-value or an r-value, you can do so as follows:

class SomeClass {

public: int& f( ); // will be used in any l-value invocation

const int& f( ) const; // used in any r-value invocation . . .

};

The two parameter lists need not be empty, but they should be the same (or else you just get simple overloading). Be sure to notice that the second declaration of f has two occurrences of const. You must include both occurrences of const. The ampersand signs & are of course also required

Some other good to know tips from Savitch…

**References reference**

A reference is the name of a storage location.1 You can have a stand-alone reference, as in the following:

int robert; int& bob = robert;

This makes bob a reference to the storage location for the variable robert, which makes bob an alias for the variable robert. Any change made to bob will also be made to robert. Stated this way, it sounds like a stand-alone reference is just a way to make your code confusing and get you in trouble. In most instances, a stand-alone reference is just trouble, although there a few cases where it can be useful. We will not discuss stand-alone references anymore, nor will we use them.

As you may suspect, references are used to implement the call-by-reference parameter mechanism. So, the concept is not completely new to this chapter, although the phrase a reference is new.

We are interested in references here because returning a reference will allow you to overload certain operators in a more natural way. Returning a reference can be viewed as something like returning a variable or, more precisely, an alias to a variable. The syntactic details are simple. You add an & to the return type. For example,

double& sampleFunction(double& variable);

Since a type like double& is a different type from double, you must use the & in both the function declaration and the function definition. The return expression must be something with a reference, such as a variable of the appropriate type. It cannot be an expression, such as X + 5. Although many compilers will let you do it (with unfortunate results), you also should not return a local variable because you would be generating an alias to a variable and immediately destroying the variable. A trivial example of the function definition is

double& sampleFunction(double& variable)

{

return variable;

}

Of course, this is a pretty useless, even troublesome, function, but it illustrates the concept. For example, the following code will output 99 and then 42:

double m = 99;

cout << sampleFunction(m) << endl;

sampleFunction(m) = 42;

cout << m << endl;

\*If you know about pointers, you will notice that a reference sounds like a pointer. A reference is essentially, but not exactly, a constant pointer. There are differences between pointers and references, and they are not completely interchangeable.

We will only be returning a reference when defining certain kinds of overloaded operators.

**L-Values and R-Values**

The term l-value is used for something that can appear on the left-hand side of an assignment operator. The term r-value is used for something that can appear on the right-hand side of an assignment operator.

If you want the object returned by a function to be an l-value, it must be returned by reference.

**TIP: Returning Member Variables of a Class Type**

When returning a member variable of a class type, in almost all cases it is important to return the member value by const value. To see why, suppose you do not, as in the example outlined in what follows:   
class Employee {

public:

Money& getSalary( ) { return salary; } ...

private:

Money salary; ...

};

In this example, salary is a private member variable that should not be changeable except by using some accessor function of the class Employee. The getSalary function returns the variable salary which is of type Money. If we do not return salary by reference then a new, temporary copy of salary will be created and returned. We might wish to avoid this overhead by returning a reference to salary as indicated in the example. However, even though salary is declared as private, this privateness is easily circumvented as follows:

Employee joe;   
(joe.getSalary( )).input( );

The lucky employee named joe can now enter any salary she wishes! On the other hand, suppose getSalary returns its value by const value, as follows:

class Employee {

public:

const Money& getSalary( ) { return salary; }

private:

Money salary; ...

};

In this case, the following will give a compiler error message:   
(joe.getSalary( )).input( );

(The declaration for getSalary should ideally be

const Money& getSalary( ) const { return salary; }

but we did not want to confuse the issue with another kind of const.) In general, when a member function returns a member variable and that member variable is of some class type, then it should normally not be returned by reference to avoid external access to a private member variable. If you want to return by reference for efficiency reasons then adding const to the return value can help protect access to the member variable.

**TIP: What Mode of Returned Value to Use**

A function can return a value of type T in four different ways:

■ By plain old value, as in the function declaration T f( );

■ By constant value, as in the function declaration const T f( );

■ By reference, as in the function declaration T& f( );

■ By const reference, as in the function declaration const T& f( )

There is not unanimous agreement on which to use when. So, do not expect too much consistency in usage. Even when an author or programmer has a clear policy, they seldom manage to follow it without exception. Still, some points are clear. If you are returning a simple type, like int or char, there is no point in using a const when returning by value or by reference. So programmers typically do not use a const on the return type when it is a simple type. If you want the simple value returned to be allowed as an l-value, that is, to be allowed on the left-hand side of an assignment statement, then return by reference; otherwise return the simple type by plain old value. Class types are not so simple. The rest of this discussion applies to returning an object of a class type.

The decision on whether or not to return by reference has to do with whether or not you want to be able to use the returned object as an l-value. If you want to return something that can be used as an l-value, that is, that can be used on the left-hand side of an assignment operator, you must return by reference and so must use an ampersand & on the returned type. Returning a local variable (or other short-lived object) by reference, with or without a const, can produce problems and should be avoided. For class types, the two returned type specifications const T and const T& are very similar. They both mean that you cannot change the returned object by invoking some mutator function directly on the returned object, as in

f().mutator( );

The returned value can still be copied to another variable with an assignment operator and that other variable can have the mutator function applied to it. If you cannot decide between the const T& and const T, use const T (without the ampersand). A const T& is perhaps a bit more efficient than a const T2.

However, the difference is not typically that important and most programmers use const T rather than const T& as a returned type specification. As noted earlier, const T& can sometimes cause problems. The following summary may be of help. T is assumed to be a class type.

Copy constructors are not covered until Chapter 10, but we include details about them here for reference value. If you have not yet read Chapter 10, simply ignore all references to copy constructors. If a public member function returns a private class member variable, it should always have a const on the returned type, as we explained in the Tip section of this chapter entitled “Returning Member Variables of a Class Type.” (One exception to this rule is that programmers normally always return a value of type string by ordinary value, not by const value. This is presumably because the type string is thought of as a simple type like int and char, even though string is a class type.)

2This is because const T& does not call the copy constructor while const T does call the copy constructor. Copy constructors are discussed in Chapter 10.

The following summary may be of help. T is assumed to be a class type.

Simple returning by value, as in the function declaration T f( ); Cannot be used as an l-value, and the returned value can be changed directly as in f( ).mutator( ). Calls the copy constructor.

Returning by constant value, as in const T f( ); This case is the same as the previous case, but the returned value cannot be changed directly as in f( ).mutator( ).

Returning by reference as in T& f( ); Can be used as an l-value, and the returned value can be changed directly as in f( ).mutator( ). Does not call the copy constructor.

Returning by constant reference, as in const T& f( ); Cannot be used as an l-value, and the returned value cannot be changed directly as in f( ).mutator( ). Does not call the copy constructor. ■