Avoid Ambiguity

Suppose you use both a one-argument constructor and a conversion operator to perform the same conversion (`time24` to `time12`, for example). How will the compiler know which conversion to use? It won't. The compiler does not like to be placed in a situation where it doesn't know what to do, and it will signal an error. So avoid doing the same conversion in more than one way.

Not All Operators Can Be Overloaded

The following operators cannot be overloaded: the member access or dot operator (`.`), the scope resolution operator (`::`), and the conditional operator (`?:`). Also, the pointer-to-member operator (`->`), which we have not yet encountered, cannot be overloaded. In case you wondered, no, you can't create new operators (like `*&`) and try to overload them; only existing operators can be overloaded.

Keywords `explicit` and `mutable`

Let's look at two unusual keywords: `explicit` and `mutable`. They have quite different effects, but are grouped together here because they both modify class members. The `explicit` keyword relates to data conversion, but `mutable` has a more subtle purpose.

Preventing Conversions with `explicit`

There may be some specific conversions you have decided are a good thing, and you've taken steps to make them possible by installing appropriate conversion operators and one-argument constructors, as shown in the TIME1 and TIME2 examples. However, there may be other conversions that you don't want to happen. You should actively discourage any conversion that you don't want. This prevents unpleasant surprises.

It's easy to prevent a conversion performed by a conversion operator: just don't define the operator. However, things aren't so easy with constructors. You may want to construct objects using a single value of another type, but you may not want the implicit conversions a one-argument constructor makes possible in other situations. What to do?

Standard C++ includes a keyword, `explicit`, to solve this problem. It's placed just before the declaration of a one-argument constructor. The EXPLICIT example program (based on the ENGLCON program) shows how this looks.

```
//explicit.cpp
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
```

```
   {
   private:
      const float MTF;          //meters to feet
      int feet;
      float inches;
   public:                      //no-args constructor
      Distance() : feet(0), inches(0.0), MTF(3.280833F)
         {  }
                                //EXPLICIT one-arg constructor
      explicit Distance(float meters) : MTF(3.280833F)
         {
         float fltfeet = MTF * meters;
         feet = int(fltfeet);
         inches = 12*(fltfeet-feet);
         }
      void showdist()           //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   void fancyDist(Distance);    //declaration
   Distance dist1(2.35F);       //uses 1-arg constructor to
                                //convert meters to Distance

// Distance dist1 = 2.35F;      //ERROR if ctor is explicit
   cout << "\ndist1 = "; dist1.showdist();

   float mtrs = 3.0F;
   cout << "\ndist1 ";
// fancyDist(mtrs);             //ERROR if ctor is explicit

   return 0;
   }
//--------------------------------------------------------------
void fancyDist(Distance d)
   {
   cout << "(in feet and inches) = ";
   d.showdist();
   cout << endl;
   }
```

This program includes a function (`fancyDist()`) that embellishes the output of a `Distance` object by printing the phrase "(in feet and inches)" before the feet and inches figures. The argument to this function is a `Distance` variable, and you can call `fancyDist()` with such a variable with no problem.

The tricky part is that, unless you take some action to prevent it, you can also call `fancyDist()` with a variable of type `float` as the argument:

```
fancyDist(mtrs);
```

The compiler will realize it's the wrong type and look for a conversion operator. Finding a `Distance` constructor that takes type `float` as an argument, it will arrange for this constructor to convert `float` to `Distance` and pass the `Distance` value to the function. This is an *implicit* conversion, one which you may not have intended to make possible.

However, if we make the constructor *explicit*, we prevent implicit conversions. You can check this by removing the comment symbol from the call to `fancyDist()` in the program: the compiler will tell you it can't perform the conversion. Without the explicit keyword, this call is perfectly legal.

As a side effect of the explicit constructor, note that you can't use the form of object initialization that uses an equal sign

```
Distance dist1 = 2.35F;
```

whereas the form with parentheses

```
Distance dist1(2.35F);
```

works as it always has.

Changing `const` Object Data Using `mutable`

Ordinarily, when you create a `const` object (as described in Chapter 6), you want a guarantee that none of its member data can be changed. However, a situation occasionally arises where you want to create `const` objects that have some specific member data item that needs to be modified despite the object's `const`ness.

As an example, let's imagine a window (the kind that Windows programs commonly draw on the screen). It may be that some of the features of the window, such as its scrollbars and menus, are *owned* by the window. Ownership is common in various programming situations, and indicates a greater degree of independence than when one object is an attribute of another. In such a situation an object may remain unchanged, except that its owner may change. A scrollbar retains the same size, color, and orientation, but its ownership may be transferred from one window to another. It's like what happens when your bank sells your mortgage to another bank; all the terms of the mortgage are the same, but the owner is different.

Let's say we want to be able to create `const` scrollbars in which attributes remain unchanged, except for their ownership. That's where the `mutable` keyword comes in. The MUTABLE program shows how this looks.

```
//mutable.cpp
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class scrollbar
   {
   private:
      int size;                     //related to constness
      mutable string owner;         //not relevant to constness
   public:
      scrollbar(int sz, string own) : size(sz), owner(own)
         {  }
      void setSize(int sz)             //changes size
         { size = sz; }
      void setOwner(string own) const  //changes owner
         { owner = own; }
      int getSize() const              //returns size
         { return size; }
      string getOwner() const          //returns owner
         { return owner; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   const scrollbar sbar(60, "Window1");

// sbar.setSize(100);              //can't do this to const obj
   sbar.setOwner("Window2");       //this is OK
                                   //these are OK too
   cout << sbar.getSize() << ", " << sbar.getOwner() << endl;
   return 0;
   }
```

The `size` attribute represents the scrollbar data that cannot be modified in `const` objects. The `owner` attribute, however, can change, even if the object is `const`. To permit this, it's made `mutable`. In `main()` we create a `const` object `sbar`. Its size cannot be modified, but its owner can, using the `setOwner()` function. (In a non-`const` object, of course, both attributes could be modified.) In this situation, `sbar` is said to have *logical* `const`*ness*. That means that in theory it can't be modified, but in practice it can, in a limited way.

Summary

In this chapter we've seen how the normal C++ operators can be given new meanings when applied to user-defined data types. The keyword `operator` is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.

Closely related to operator overloading is the issue of *type conversion*. Some conversions take place between user-defined types and basic types. Two approaches are used in such conversions: A one-argument constructor changes a basic type to a user-defined type, and a conversion operator converts a user-defined type to a basic type. When one user-defined type is converted to another, either approach can be used.

Table 8.2 summarizes these conversions.

TABLE 8.2 Type Conversions

	Routine in Destination	*Routine in Source*
Basic to basic	**(Built-In Conversion Operators)**	
Basic to class	Constructor	N/A
Class to basic	N/A	Conversion operator
Class to class	Constructor	Conversion operator

A constructor given the keyword `explicit` cannot be used in implicit data conversion situations. A data member given the keyword `mutable` can be changed, even if its object is `const`.

UML class diagrams show classes and relationships between classes. An association represents a conceptual relationship between the real-world objects that the program's classes represent. Associations can have a direction from one class to another; this is called navigability.

Questions

Answers to these questions can be found in Appendix G.

1. Operator overloading is
   a. making C++ operators work with objects.
   b. giving C++ operators more than they can handle.
   c. giving new meanings to existing C++ operators.
   d. making new C++ operators.
2. Assuming that class `X` does not use any overloaded operators, write a statement that subtracts an object of class `X`, `x1`, from another such object, `x2`, and places the result in `x3`.