Data Conversion

You already know that the = operator will assign a value from one variable to another, in statements like intvar1 = intvar2; where intvar1 and intvar2 are integer variables. You may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like dist3 = dist1 + dist2;

where the result of the addition, which is type Distance, is assigned to another object of type Distance, dist3. Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler doesn't need any special instructions to use = for the assignment of user-defined objects such as Distance objects.

Thus, assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign. But what happens when the variables on different sides of the = are of different types? This is a more thorny question, to which we will devote the balance of this chapter. We'll first review how the compiler handles the conversion of basic types, which it does automatically. Then we'll explore several situations where the compiler doesn't handle things automatically and we need to tell it what to do. These include conversions between basic types and user-defined types, and conversions between different user-defined types.

You might think it represents poor programming practice to convert routinely from one type to another. After all, languages such as Pascal go to considerable trouble to keep you from doing such conversions. However, the philosophy in C++ (and C) is that the flexibility provided by

allowing conversions outweighs the dangers. This is a controversial issue; we'll return to it at the end of this chapter.

Conversions Between Basic Types

When we write a statement like intvar = floatvar; where intvar is of type int and floatvar is of type float, we are assuming that the compiler will call a special routine to convert the value of floatvar, which is expressed in floating-point format, to an integer format so that it can be assigned to intvar. There are of course many such conversions: from float to double, char to float, and so on. Each such conversion has its own routine, built into the compiler and called up when the data types on different sides of the equal sign so dictate. We say such conversions are implicit because they aren't apparent in the listing. Sometimes we want to force the compiler to convert one type to another. To do this we use the cast operator. For instance, to convert float to int, we can say

```
intvar = static_cast<int>(floatvar);
```

Casting provides explicit conversion: It's obvious in the listing that static_cast<int>() is intended to convert from float to int. However, such explicit conversions use the same built-in routines as implicit conversions.

Conversions Between Objects and Basic Types

When we want to convert between user-defined data types and basic types, we can't rely on built-in conversion routines, since the compiler doesn't know anything about user-defined types besides what we tell it. Instead, we must write these routines ourselves. Our next example shows how to convert between a basic type and a user-defined type. In this example the user-defined type is (surprise!) the English Distance class from previous examples, and the basic type is float, which we use to represent meters, a unit of length in the metric measurement system. The example shows

conversion both from Distance to float, and from float to Distance. Here's the listing for ENGLCONV:

```
// englconv.cpp
// conversions: Distance to meters, meters to Distance
#include <iostream>
class Distance //English Distance class
{
private:
const float MTF; //meters to feet
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0), MTF(3.280833F)
{ } //constructor (one arg)
Distance(float meters) : MTF(3.280833F)
{ //convert meters to Distance
float fltfeet = MTF * meters; //convert to float feet
feet = int(fltfeet); //feet is integer part
inches = 12*(fltfeet-feet); //inches is what's left
} //constructor (two args)
Distance(int ft, float in) : feet(ft),inches(in), MTF(3.280833F)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
```

```
{ cout << feet << "\'-" << inches << '\"'; }
operator float() const //conversion operator
{ //converts Distance to meters
float fracfeet = inches/12; //convert the inches
fracfeet += static_cast<float>(feet); //add the feet
return fracfeet/MTF; //convert to meters
}
};
//////////////////////////////////////////////////////////////
int main()
{
float mtrs;
Distance dist1 = 2.35F; //uses 1-arg constructor to
//convert meters to Distance
cout << "\ndist1 = "; dist1.showdist();
mtrs = static_cast<float>(dist1); //uses conversion operator
//for Distance to meters
cout << "\ndist1 = " << mtrs << " meters\n";
Distance dist2(5, 10.25); //uses 2-arg constructor
mtrs = dist2; //also uses conversion op
cout << "\ndist2 = " << mtrs << " meters\n";
// dist2 = mtrs; //error, = won't convert
return 0;
}
```

In main() the program first converts a fixed float quantity—2.35, representing meters—to feet and inches, using the one-argument

constructor:

Distance dist1 = 2.35F;

Going in the other direction, it converts a Distance to meters in the statements mtrs = static_cast<float>(dist2); and mtrs = dist2; Here's the output:

```
dist1 = 7'-8.51949" ←□□□ this is 2.35 meters
dist1 = 2.35 meters ←□□□ this is 7'–8.51949"
dist2 = 1.78435 meters ←□□□ this is 5'–10.25"
```

We've seen how conversions are performed using simple assignment statements in main().

Now let's see what goes on behind the scenes, in the Distance member functions. Converting a user-defined type to a basic type requires a different approach than converting a basic type to a user-defined type. We'll see how both types of conversions are carried out in ENGLCONV.

From Basic to User-Defined

To go from a basic type—float in this case—to a user-defined type such as Distance, we use a constructor with one argument. These are sometimes called conversion constructors. Here's how this constructor looks in ENGLCONV:

```
Distance(float meters)
{
float fltfeet = MTF * meters;
feet = int(fltfeet);
inches = 12 * (fltfeet-feet);
}
```

This function is called when an object of type Distance is created with a single argument. The function assumes that this argument represents meters. It converts the argument to feet and inches, and assigns the resulting values to the object. Thus the conversion from meters to Distance is carried out along with the creation of an object in the statement Distance dist1 = 2.35;

From User-Defined to Basic

What about going the other way, from a user-defined type to a basic type? The trick here is to create something called a conversion operator. Here's where we do that in ENGLCONV:

```
operator float()
{
float fracfeet = inches/12;
fracfeet += float(feet);
return fracfeet/MTF;
}
```

This operator takes the value of the Distance object of which it is a member, converts it to a float value representing meters, and returns this value.

Conversions between Objects of Different Classes

What about converting between objects of different user-defined classes? The same two methods just shown for conversions between basic types and user-defined types also apply to conversions between two user-defined types. That is, you can use a one-argument constructor or you can use a conversion operator. The choice depends on whether you want to put the conversion routine in the class declaration of the source object or of

the destination object. For example, suppose you say objecta = objectb; where objecta is a member of class A and objectb is a member of class B. Is the conversion routine located in class A (the destination class, since objecta receives the value) or class B (the source class)? We'll look at both cases.

Two Kinds of Time

Our example programs will convert between two ways of measuring time: 12-hour time and 24-hour time. These methods of telling time are sometimes called **civilian time** and **military time**. Our time12 class will represent civilian time, as used in digital clocks and airport flight departure displays. We'll assume that in this context there is no need for seconds, so time12 uses only hours (from 1 to 12), minutes, and an "a.m." or "p.m." designation. Our time24 class, which is for more exacting applications such as air navigation, uses hours (from 00 to 23), minutes, and seconds. Table 8.1 shows the differences.

TABLE 8.1 12-Hour and 24-Hour Time

12-Hour Time	*24-Hour Time*
12:00 a.m. (midnight)	00:00
12:01 a.m.	00:01
1:00 a.m.	01:00
6:00 a.m.	06:00
11:59 a.m	11:59
12:00 p.m. (noon)	12:00
12:01 p.m.	12:01
6:00 p.m.	18:00
11:59 p.m.	23:59

Note that 12 a.m. (midnight) in civilian time is 00 hours in military time. There is no 0 hour in civilian time.

Routine in Source Object

The first example program shows a conversion routine located in the source class. When the conversion routine is in the source class, it is commonly implemented as a conversion operator.Here's the listing for TIMES1:

```
//converts from time24 to time12 using operator in time24
#include <iostream>
#include <string>
class time12
{
private:
bool pm; //true = pm, false = am
int hrs; //1 to 12
int mins; //0 to 59
public: //no-arg constructor
time12() : pm(true), hrs(0), mins(0)
{ }
//3-arg constructor
time12(bool ap, int h, int m) : pm(ap), hrs(h), mins(m)
{ }
void display() const //format: 11:59 p.m.
{
cout << hrs << ':';
if(mins < 10)
cout << '0'; //extra zero for "01"
```

```
cout << mins << ' ';
string am_pm = pm ? "p.m." : "a.m.";
cout << am_pm;
}
};
class time24
{
private:
int hours; //0 to 23
int minutes; //0 to 59
int seconds; //0 to 59
public: //no-arg constructor
time24() : hours(0), minutes(0), seconds(0)
{ }
time24(int h, int m, int s) : //3-arg constructor
hours(h), minutes(m), seconds(s)
{ }
void display() const //format: 23:15:01
{
if(hours < 10) cout << '0';
cout << hours << ':';
if(minutes < 10) cout << '0';
cout << minutes << ':';
if(seconds < 10) cout << '0';
cout << seconds;
}
operator time12() const; //conversion operator
};
```

```
//-----------------------------------------------------------
time24::operator time12() const //conversion operator
{
int hrs24 = hours;
bool pm = hours < 12 ? false : true; //find am/pm
//round secs
int roundMins = seconds < 30 ? minutes : minutes+1;
if(roundMins == 60) //carry mins?
{
roundMins=0;
++hrs24;
if(hrs24 == 12 || hrs24 == 24) //carry hrs?
pm = (pm==true) ? false : true; //toggle am/pm
}
int hrs12 = (hrs24 < 13) ? hrs24 : hrs24-12;
if(hrs12==0) //00 is 12 a.m.
{ hrs12=12; pm=false; }
return time12(pm, hrs12, roundMins);
}
int main()
{
int h, m, s;
while(true)
{ //get 24-hr time from user
cout << "Enter 24-hour time: \n";
cout << " Hours (0 to 23): "; cin >> h;
if(h > 23) //quit if hours > 23
return(1);
```

```
cout << " Minutes: "; cin >> m;
cout << " Seconds: "; cin >> s;
time24 t24(h, m, s); //make a time24
cout << "You entered: "; //display the time24
t24.display();
time12 t12 = t24; //convert time24 to time12
cout << "\n12-hour time: "; //display equivalent time12
t12.display();
cout << "\n\n";
}
return 0;
}
```

In the main() part of TIMES1 we define an object of type time24, called t24, and give it values for hours, minutes, and seconds obtained from the user. We also define an object of type time12, called t12, and initialize it to t24 in the statement time12 t12 = t24; Since these objects are from different classes, the assignment involves a conversion, and—as we specified—in this program the conversion operator is a member of the time24 class. Here's its declarator:

time24::operator time12() const //conversion operator } This function transforms the object of which it is a member to a time12 object, and returns this object, which main() then assigns to t12. Here's some interaction with TIMES1:

```
Enter 24-hour time:
Hours (0 to 23): 17
Minutes: 59
Seconds: 45
You entered: 17:59:45
12-hour time: 6:00 p.m.
```