Data Conversion

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

September 29, 2024

Data Conversion

- The "=" operator will assign a value from one variable to another, in statements like
  `intvar1 = intvar2;`
  where intvar1 and intvar2 are integer variables.
- If "=" is overloaded correctly it would do the same for the value of one user-defined object to another
  `dist3 = dist1 + dist2;`
  provided they are of the same type.
- 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.

Data Conversion

- But what happens when the variables on different sides of the ”=” are of different types?
- One might think it represents poor programming practice to convert routinely from one type to another.
- 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 very complicated question (?)
- To answer this lets first review how the compiler handles the conversion of basic types, which it does automatically.

Conversions Between Basic Types

- When we write a statement like
  `intvar = floatvar;`
- 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 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.
- To explicitly do a conversion we use the cast operator.
- For instance, to convert `float` to `int`, we can say
  `intvar = static_cast<int>(floatvar);`
- 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
- we must write these routines ourselves.
- the next example shows how to convert between a basic type and a user-defined type, where the user-defined type is English Distance class and the basic type is float
- see "englconv.cpp"
- 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;
```

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.
- the function

```
Distance(float meters)
{
    float fltfeet = MTF * meters;
...
```

- 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 `operator float()`

```
{
float fracfeet = inches/12;
...
```

- 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.
- This operator can be called with an explicit cast

```
mtrs = static_cast<float>(dist1);
```

- or with a simple assignment

```
mtrs = dist2;
```

Conversions Between Objects of Different Classes

- What about converting between objects of different user-defined classes?
- we can use a one-argument constructor
- or we 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 say
  `objecta = objectb;`
  - where `objecta` is a member of class `A`
  - `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)?

Two Kinds of Time

- Our example programs will convert between two ways of measuring time: 12-hour time and 24-hour time.
- the "`time12`" class will represent civilian time, as used in digital clocks
- 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.

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

Routine in Source Object

- ”times1.cpp” shows a conversion routine located in the source class.
- 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
- in this program the conversion operator is a member of the `time24` class
- `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`.

Routine in Destination Object

- how the same conversion is carried out when the conversion routine is in the destination class.
- it's common to use a one-argument constructor
- things are complicated by the fact that the constructor in the destination class must be able to access the data in the source class to perform the conversion.
- The data in `time24`—hours, minutes and seconds—is private,
- so we must provide special member functions in `time24` to allow direct access to it.
- These are called `getHrs()`, `getMins()`, and `getSecs()`
- The conversion routine is the one-argument constructor from the `time12` class.
- This function sets the object of which it is a member to values that correspond to the `time24` values of the object received as an argument.

Routine in Destination Object

- It works in much the same way as the conversion operator in TIMES1, except that it must work a little harder to access the data in the `time24` object, using `getHrs()` and similar functions.
- The `main()` part of TIMES2 is the same as that in TIMES1. The one-argument constructor again allows the time24-to-time12 conversion to take place in the statement
  `time12 t12 = t24;`
- The output is similar as well.
- The difference is behind the scenes, where the conversion is handled by a constructor in the destination object rather than a conversion operator in the source object.