Conversions

Short Review of Simple Typecasting

Recall that C++ provides for explicit conversions among built-in types by use of predefined typecast operators:

\[
\begin{align*}
    \text{int } I &= 12; \\
    \text{double } D &= 42.3; \\
    \text{int } J &= \text{int}(D); \\
    \text{double } E &= \text{double}(I);
\end{align*}
\]

Although the use of explicit casts above does not alter the values that are ultimately assigned to \( J \) and \( E \), the use of explicit casts is still good practice since it renders the implicit conversions supplied by C++ more visible.

By making the conversions explicit, the programmer acknowledges that he/she is aware they will occur — and presumably that they are acceptable in the given context.

A Simple Date Class

Consider a simple class for representing dates:

```cpp
class Date {
private:
    int Month, Day, Year;
public:
    Date();
    Date(int M, int D, int Y);
    Date(int yyyymmdd); // conversion constructor
    void ShowDate(); // display function
};
```

Converts an int value into a Date object.

Simple Date Class Implementation

```cpp
Date::Date() {
    Month = 3;
    Day = 10;
    Year = 1987;
}

Date::Date(int M, int D, int Y) {
    Month = M;
    Day = D;
    Year = Y;
}

void Date::ShowDate(ostream &Out) {
    Out << setfill('0')
        << setw(2) << Month << '/'
        << setw(2) << Day << '/'
        << setw(2) << Year;
}
```
Converting Built-in to User-defined

The conversion of a built-in type to a user-defined type can be accomplished by the use of an appropriate constructor for the targeted user-defined type:

```cpp
Date::Date(int yyyymmdd) {
    Year = yyyymmdd / 10000;
    Month = (yyyymmdd - Year * 10000) / 100;
    Day = yyyymmdd - Year * 10000 - Month * 100;
}
```

The Date implementation should be improved by adding error-handling in case the parameter values simply could not represent a valid date.

Using the Conversion Constructor

This makes the conversion as simple as an explicit cast of one built-in type to another built-in type.

```cpp
void main() {
    Date a;
    cout << "Date a is:" << endl;
    a.ShowDate(cout);
    cout << endl;
    a = Date(20020101);
    cout << "Date a is now: " << endl;
    a.ShowDate(cout);
    cout << endl;
}
```

Conversion of int value into a Date object. Looks just like a standard (old-style) explicit cast.

Output

```
Date a is:
07/04/2001
Date a is now:
01/01/2002
```

Conversion Operators

A conversion operator function is simply an operator that takes a value of one type and produces a value of another type. The syntax is identical to that for the built-in typecasts:

```cpp
Date::operator int() {
    int yyyymmdd;
    yyyymmdd = Year * 10000 + Month * 100 + Day;
    return yyyymmdd;
}
```

Converts a Date object into an int.

Converting User-defined to Built-in

The conversion of a user-defined type to a built-in type can be accomplished by the use of an appropriate conversion operator as a member of the user-defined type:

```cpp
class Date {
    private:
        int Month, Day, Year;
    public:
        Date();
        Date(int M, int D, int Y);
        Date(int yyyymmdd);
        operator int();
        void ShowDate();
    };
```

Note that the type used for the operator name MUST be declared within the scope of the operator declaration.
**Using the Conversion Operator**

As before, this also makes the conversion as simple as an explicit cast of one built-in type to another built-in type:

```cpp
void main() {
    Date a(4, 1, 1999);
    int b;
    b = (int) a;
    cout << "a's date is: ";
    a.ShowDate();
    cout << endl
    << "This date, as an int, is: " << b << endl;
}
```

**Output**

```
a's date is: 04/01/1999
This date, as an int, is: 19990401
```

**Converting Between User-defined Types**

The conversion of a user-defined type to a user-defined type is also accomplished by the use of a member conversion operator.

In this case, it frequently makes sense to provide conversion operators "on both sides" to facilitate translation in both directions.

That, of course, poses a small problem since both type names must be declared prior to the declaration of the relevant operators...

... resolution is normally done by use of forward declarations...

**Add an IntDate Class**

Let's implement a more space-efficient class for dates:

```cpp
// IntDate.h
class IntDate {
private:
    int yyyymmdd;
public:
    IntDate(int ymd = 0); // conversion op
    operator Date(); // conversion op
    void ShowIntDate();
};
```

**Update the Date Class**

... and update the Date class for conversions also:

```cpp
// Date.h
class IntDate; // forward declaration
class Date {
private:
    int Month, Day, Year;
public:
    Date(int M = 7, int D = 4, int Y = 2001); // conversion op
    operator IntDate(); // conversion op
    void ShowDate();
};
```

```cpp
Date::operator IntDate() {
    int Temp;
    Temp = 10000 * Year + 100 * Month + Day;
    return IntDate(Temp);
}
```
Conversions of `IntDate` object into a `Date` object and of a `Date` object into an `IntDate` object look just like standard (old-style) explicit casts.

```cpp
void main() {
    Date a(4, 1, 1999), b;
    IntDate c(20011215), d;
    b = Date(c);
    d = IntDate(a);
    cout << "a's date is: ";
    a.ShowDate();
    cout << endl << "as an IntDate object this date is: ";
    d.ShowIntDate();
    // continues . . .
}
```

Output

```
a's date is: 04/01/1999
as an IntDate object this date is: 19990401
c's date is: 20011215
as a Date object this date is: 12/15/2001
```