

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

```
int I      = 12;  
double D  = 42.3;  
int J     = int(D);  
int K     = D;  
double E  = double(I);
```

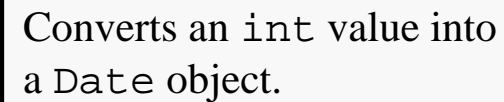
Although the use of explicit casts above does not alter the values that are ultimately assigned to J and E, the use of explicit casts is still good practice since it renders the implicit conversions supplied by C++ more visible.

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:

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



Converts an int value into a Date object.

```
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;
}
```

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:

```
Date::Date(int yyyyymmdd) {  
    Year   = yyyyymmdd / 10000;  
    Month  = (yyyyymmdd - Year * 10000) / 100;  
    Day    = yyyyymmdd - 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.

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

```
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 << endl;  
}
```

Conversion of int value into a Date object.

Looks just like a standard explicit cast.

We could also write:

```
a = 20020101;
```

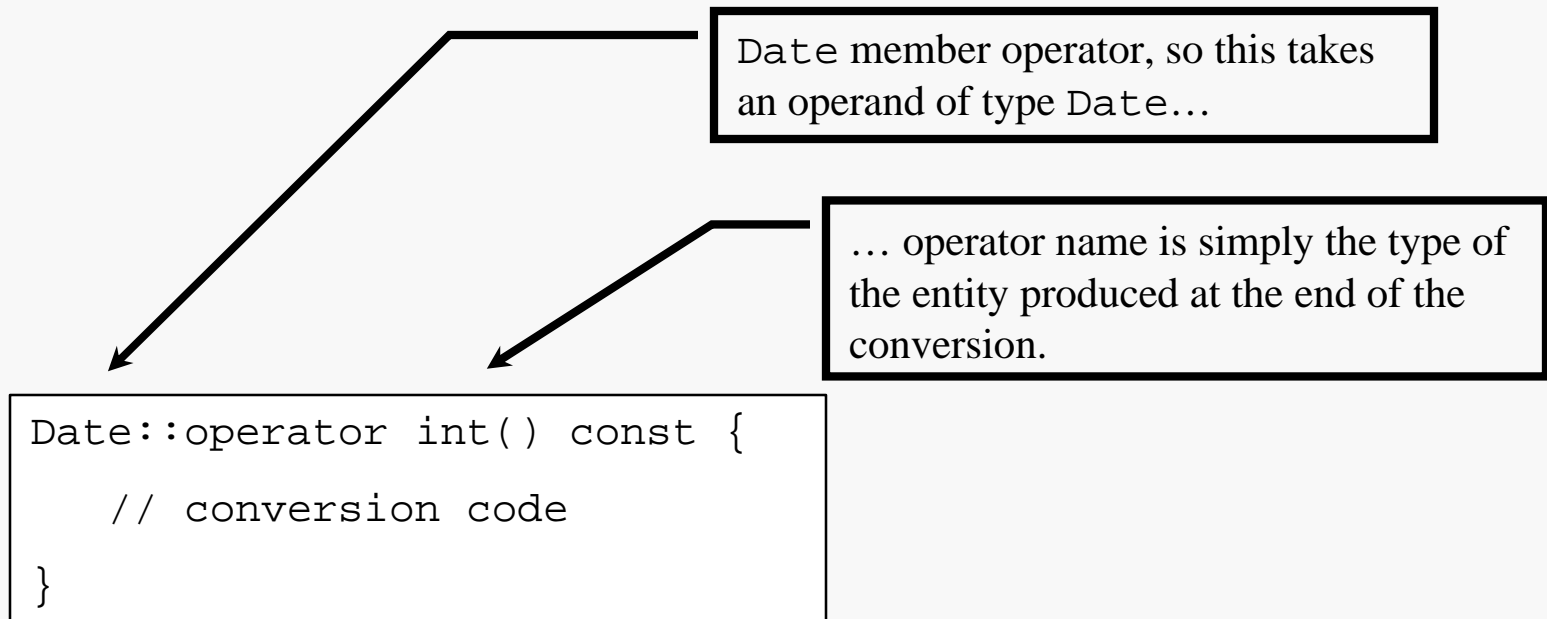
or even the old C-style:

```
a = (Date) 20020101;
```

Output

```
Date a is:  
03/10/1987  
Date a is now:  
01/01/2002
```

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:



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

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:

```
class Date {  
private:  
    int Month, Day, Year;  
public:  
    Date();  
    Date(int M, int D, int Y);  
    Date(int yyymmdd);  
    operator int()const;  
    void ShowDate(ostream& Out);  
};
```

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

Converts a Date object into an int.

# 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:

```
void main() {
    Date a(4, 1, 1999);
    int b;

    b = int(a);

    cout << "a's date is: ";
    a.ShowDate(cout);
    cout << endl
         << "This date, as an int, is: "
         << b << endl;;
}
```

Conversion of Date object into an int value.  
Looks just like a standard explicit cast.

Output

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

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:

```
// IntDate.h
...
class Date; // forward declaration

class IntDate {
private:
    int yyymmdd;

public:
    IntDate(int ymd = 0);
    operator Date(); // conversion op
    void ShowDate(ostream& Out);
};
```

Converts an IntDate object into a Date object.

Assumes Date has an appropriate constructor.

```
IntDate::operator Date() {
    int M, D, Y;
    Y = yyymmdd / 10000;
    M = (yyymmdd - Y*10000) / 100;
    D = yyymmdd - Y*10000 - M*100;
    return Date(M, D, Y);
}
```

# Update the Date Class Declaration

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

```
// Date.h
. . .
class IntDate; // forward declaration

class Date {
private:
    int Month, Day, Year;

public:
    Date(int M = 7, int D = 4, int Y = 2001);
    operator IntDate(); // conversion op
    void ShowDate(ostream& Out);
};
```

Converts a Date object into an IntDate object.

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

Assumes IntDate has an appropriate constructor.

# Using the Conversions

This makes the conversions between the user-defined types as simple as an explicit cast of one built-in type to another built-in type.

```
#include "Date.h"
#include "IntDate.h"

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);

    cout << endl << "as an IntDate object this date is: ";
    d.ShowDate(cout);

    // continues . . .
}
```

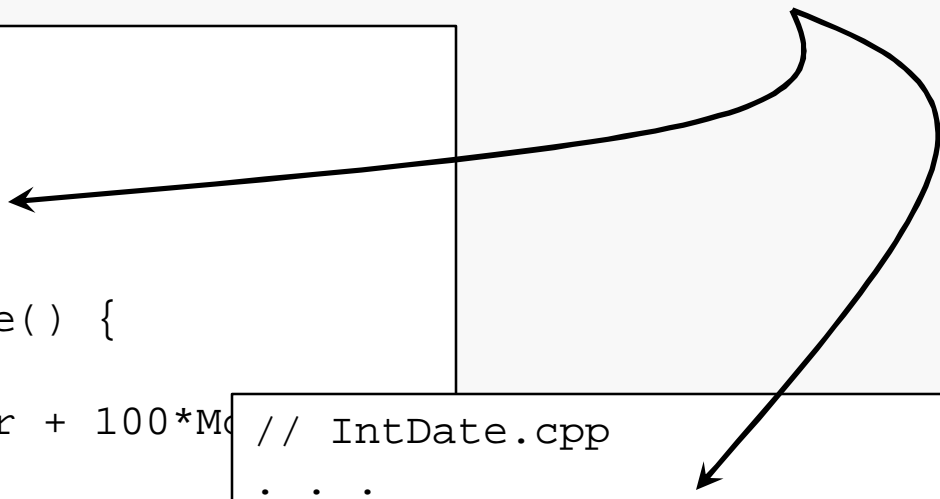
Conversions of IntDate object into a Date object and of a Date object into an IntDate object look just like standard explicit casts.

Each implementation file will include the appropriate class declarations:

```
// Date.cpp
. . .
#include "Date.h"
#include "IntDate.h"
. . .
Date::operator IntDate() {
    int Temp;
    Temp = 10000 * Year + 100*M
Day;
    return IntDate(Temp);
}
```

```
// IntDate.cpp
. . .
#include "IntDate.h"
#include "Date.h"
. . .
IntDate::operator Date() {
    int M, D, Y;

    Y = yyyyymmdd / 10000;
    M = (yyyyymmdd - Y*10000) / 100;
    D = yyyyymmdd - Y*10000 - M*100;
    return Date(M, D, Y);
}
```



# Using the Conversions

```
// . . . continued

cout << endl << "c's date is: ";
c.ShowDate(cout);

cout << endl << "as a Date object this date is: ";
b.ShowDate(cout);
cout << endl << endl;
}
```

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
```

## A Final Example

Recall the `Person/Employee` hierarchy defined earlier. The following assignment of a base object to a derived object would normally be illegal:

```
Employee Jill(. . .);  
Staff      Jillian(. . .);  
Jillian = Jill;    // derived <-- base type assign.
```

However, with the implementation shown before, this statement is legal. Why?

Recall that the class `Staff` has the following constructor:

```
Staff::Staff(const Employee& E, double R = 0.0) : Employee(E) {  
    HourlyRate = R;  
}
```

**QTP:** why does the inclusion of this constructor make the assignment above legal?