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.
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(ostream& Out); // 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;
}
```
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 << 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;
```

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 type casts:

```
Date::operator int() const {
    // conversion code
}
```

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:

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

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

Converts a Date object into an int.
As before, this also makes the conversion as simple as an explicit cast of one built-in type to another built-in type:

```c++
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
```
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…
Let’s implement a more space-efficient class for dates:

```cpp
// IntDate.h

... class Date; // forward declaration

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

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

Assumes `Date` has an appropriate constructor.

Converts an IntDate object into a Date object.
Update the **Date** Class Declaration

... 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);
    operator IntDate(); // conversion op
    void ShowDate(ostream& Out);
};
```

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

Converts a **Date** object into an **IntDate** object.

Assumes **IntDate** has an appropriate constructor.
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.

```cpp
#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:

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

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

    Y = yyyymmdd / 10000;
    M = (yyymmdd - Y*10000) / 100;
    D = yyyymmdd - Y*10000 - M*100;
    return Date(M, D, Y);
}
```
// ... 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;
}
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?