4.4 — Implicit type conversion (coercion)

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 8TH, 2018

Previously, you learned that a value of a variable is stored as a sequence of bits, and the data type of the variable tells the compiler how to interpret those bits into meaningful values. Different data types may represent the "same" number differently -- for example, the integer value 3 and the float value 3.0 are stored as completely different binary patterns.

So what happens when we do something like this?

```cpp
float f = 3; // initialize floating point variable with integer 3
```

In such a case, the compiler can't just copy the bits representing the value 3 into float f. Instead, it needs to convert the integer 3 to a floating point number, which can then be assigned to variable f.

The process of converting a value from one data type to another is called a **type conversion**. Type conversions can happen in many different cases:

Assigning to or initializing a variable with a value of a different data type:

```cpp
double d(3); // initialize double variable with integer value 3
d = 6; // assign double variable with integer value 6
```

Passing a value to a function where the function parameter is of a different data type:

```cpp
void doSomething(long l)
{
}

doSomething(3); // pass integer value 3 to a function expecting a long parameter
```

Returning a value from a function where the function return type is of a different data type:

```cpp
float doSomething()
{
    return 3.0; // Return double value 3.0 back to caller through float return type
}
```

Using a binary operator with operands of different types:

```cpp
double division = 4.0 / 3; // division with a double and an integer
```

In all of these cases (and quite a few others), C++ will use type conversion to convert data from one type to another.

There are two basic types of type conversion: **implicit type conversion**, where the compiler automatically transforms one fundamental data type into another, and **explicit type conversions**, where the developer uses a casting operator to direct the conversion.

We'll cover implicit type conversion in this lesson, and explicit type conversion in the next.

Implicit type conversion

Implicit type conversion (also called **automatic type conversion** or **coercion**) is performed whenever one fundamental data type is expected, but a different fundamental data type is supplied, and the user does not explicitly tell the compiler how to perform this conversion (via a cast).

All of the above examples are cases where implicit type conversion will be used.

There are two basic types of implicit type conversion: promotions and conversions.

Numeric promotion

Whenever a value from one type is converted into a value of a larger similar data type, this is called a **numeric promotion** (or **widening**, though this term is usually reserved for integers). For example, an int can be widened into a long, or a float promoted into a double:

```
long l(64); // widen the integer 64 into a long
double d(0.12f); // promote the float 0.12 into a double
```

While the term “numeric promotion” covers *any* type of promotion, there are two other terms with specific meanings in C++:

- **Integral promotion** involves the conversion of integer types narrower than int (which includes bool, char, unsigned char, signed char, unsigned short, signed short) to an integer (if possible) or an unsigned int.
- **Floating point promotion** involves the conversion of a float to a double.

Integral promotion and floating point promotion are used in specific cases to convert smaller data types to int/unsigned int or double, because int and double are generally the most performant types to perform operations on.

The important thing to remember about promotions is that they are always safe, and no data loss will result. Under the hood, promotions generally involve extending the binary representation of a number (e.g. for integers, adding leading 0s).

Numeric conversions

When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion**. For example:

```
double d = 3; // convert integer 3 to a double (between different types)
short s = 2; // convert integer 2 to a short (from larger to smaller type)
```

Unlike promotions, which are always safe, conversions may or may not result in a loss of data. Because of this, any code that does an implicit conversion will often cause the compiler to issue a warning (on the other hand, if you do an explicit conversion using a cast, the compiler will assume you know what you’re doing and not issue a warning). Under the hood, conversions typically require converting the underlying binary representation to a different format.

The rules for conversions are complicated and numerous, so we’ll just cover the common cases here.

In *all* cases, converting a value into a type that doesn’t have a large enough range to support the value will lead to unexpected results. This should be avoided. For example:

```
int main()
{
    int i = 30000;
    char c = i;

    std::cout << static_cast<int>(c);

    return 0;
}
```

In this example, we’ve assigned a large integer to a char (that has range -128 to 127). This causes the char to overflow, and produces an undefined result:

```
48
```

However, converting from a larger integral or floating point type to a smaller similar type will generally work so long as the value fits in the range of the smaller type. For example:

```
    int i = 2;
    short s = i; // convert from int to short
    std::cout << s;

    double d = 0.1234;
    float f = d;
    std::cout << f;
```

This produces the expected result:

```
2
0.1234
```

In the case of floating point values, some rounding may occur due to a loss of precision in the smaller type. For example:

```
    float f = 0.123456789; // double value 0.123456789 has 9 significant digits, but float can only supp
ort about 7
    std::cout << std::setprecision(9) << f; // std::setprecision defined in iomanip header
```

In this case, we see a loss of precision because the float can't hold as much precision as a double:

```
0.123456791
```

Converting from an integer to a floating point number generally works as long as the value fits within the range of the floating type. For example:

```
    int i = 10;
    float f = i;
    std::cout << f;
```

This produces the expected result:

```
10
```

Converting from a floating point to an integer works as long as the value fits within the range of the integer, but any fractional values are lost. For example:

```
    int i = 3.5;
    std::cout << i;
```

In this example, the fractional value (.5) is lost, leaving the following result:

```
3
```

Evaluating arithmetic expressions

When evaluating expressions, the compiler breaks each expression down into individual subexpressions. The arithmetic operators require their operands to be of the same type. To ensure this, the compiler uses the following rules:

- If an operand is an integer that is narrower than an int, it undergoes integral promotion (as described above) to int or unsigned int.
- If the operands still do not match, then the compiler finds the highest priority operand and implicitly converts the other operand to match.

The priority of operands is as follows:

- long double (highest)
- double
- float
- unsigned long long
- long long
- unsigned long
- long
- unsigned int
- int (lowest)

We can see the usual arithmetic conversion take place via use of the typeid() operator (included in the typeinfo header), which can be used to show the resulting type of an expression.

In the following example, we add two shorts:

```
#include <iostream>
#include <typeinfo> // for typeid()

```

```
int main()
{
    short a(4);
    short b(5);
    std::cout << typeid(a + b).name() << " " << a + b << std::endl; // show us the type of a + b

    return 0;
}
```

Because shorts are integers, they undergo integral promotion to ints before being added. The result of adding two ints is an int, as you would expect:

```
int 9
```

Note: Your compiler may display something slightly different as the format of typeid.name() is left up to the compiler.

Let’s take a look at another case:

```
#include <iostream>
#include <typeinfo> // for typeid()

int main()
{
    double d(4.0);
    short s(2);
    std::cout << typeid(d + s).name() << " " << d + s << std::endl; // show us the type of d + s

    return 0;
}
```

In this case, the short undergoes integral promotion to an int. However, the int and double still do not match. Since double is higher on the hierarchy of types, the integer 2 gets converted to the double 2.0, and the doubles are added to produce a double result.

```
double 6.0
```

This hierarchy can cause some interesting issues. For example, take a look at the following code:

```
std::cout << 5u - 10; // 5u means treat 5 as an unsigned integer
```

you might expect the expression `5u - 10` to evaluate to -5 since 5 - 10 = -5. But here’s what actually happens:

```
4294967291
```

In this case, the signed integer (10) is promoted to an unsigned integer (which has higher priority), and the expression is evaluated as an unsigned int. Overflow results, and we get an answer we don’t expect.

This is one of many good reasons to avoid unsigned integers in general.

Warning: Microsoft’s Visual C++ 2005 does not seem to issue warnings for unsafe signed/unsigned conversions.

Quiz time!

1) What’s the difference between a numeric promotion and a numeric conversion?

Hide Solution

Numeric promotion is converting a smaller (typically integral or floating point) type to a larger similar (integral or floating point) type. Promotions generally involve extending the binary representation of a number (e.g. for integers, adding leading 0s)

Numeric conversion is converting a larger type to a smaller type, or between different types. Conversions require converting the underlying binary representation to a different format.