

8.3 — Numeric conversions

ALEX JULY 4, 2021

In the previous lesson ([8.2 -- Floating-point and integral promotion](#)), we covered numeric promotions, which are conversions of specific narrower numeric types to wider numeric types (typically `int` or `double`) that can be processed efficiently.

C++ supports another category of numeric type conversions, called **numeric conversions**, that cover additional type conversions not covered by the numeric promotion rules.

Key insight

Any type conversion covered by the numeric promotion rules ([8.2 -- Floating-point and integral promotion](#)) is a numeric promotion, not a numeric conversion.

There are five basic types of numeric conversions.

1. Converting an integral type to any other integral type (excluding integral promotions):

```
1 | short s = 3; // convert int to
   | short
   | long l = 3; // convert int to long
   | char ch = s; // convert short to
   | char
```

2. Converting a floating point type to any other floating point type (excluding floating point promotions):

```
1 | float f = 3.0; // convert double to float
   | long double ld = 3.0; // convert double to long
   | double
```

3. Converting a floating point type to any integral type:

```
1 | int i = 3.5; // convert double to
   | int
```

4. Converting an integral type to any floating point type:

```
1 | double d = 3; // convert int to
   | double
```

5. Converting an integral type or a floating point type to a bool:

```
1 | bool b1 = 3; // convert int to bool
   | bool b2 = 3.0; // convert double to
   | bool
```

As an aside...

Because brace initialization disallows some numeric conversions (more on this in a moment), we use copy initialization in this lesson (which does not have any such limitations) in order to keep the examples simple.

Narrowing conversions

Unlike a numeric promotion (which is always safe), a numeric conversion may (or may not) result in the loss of data or precision.

Some numeric conversions are always safe (such as `int` to `long`, or `int` to `double`). Other numeric conversions, such as `double` to `int`, may result in the loss of data (depending on the specific value being converted and/or the range of the underlying types):

```
1 | int i1 = 3.5; // the 0.5 is dropped, resulting in lost data
   | int i2 = 3.0; // okay, will be converted to value 3, so no data is
   | lost
```

In C++, a **narrowing conversion** is a numeric conversion that may result in the loss of data. Such narrowing conversions include:

- From a floating point type to an integral type.
- From a wider floating point type to a narrower floating point type, unless the value being converted is `constexpr` and is in range of the destination type (even if it can not be represented exactly).
- From an integer to a floating point type, unless the value being converted is `constexpr` and is in range of the destination type and can be converted back into the original type without data loss.
- From a wider integer type to a narrower integer type, unless the value being converted is `constexpr` and after integral promotion will fit into the destination type.

The good news is that you don't need to remember these. Your compiler will usually issue a warning (or error) when it determines that an implicit narrowing conversion is required.

Warning

Compilers will often *not* warn when converting a signed `int` to an unsigned `int`, or vice-versa, even though these are narrowing conversions. Be extra careful of inadvertent conversions between these types (particularly when passing an argument to a function taking a parameter of the opposite sign).

For example, when compiling the following program:

```
1 | int main()
   | {
2 |     int i =
3 |     3.5;
   | }
```

Visual Studio produces the following warning:

```
warning C4244: 'initializing': conversion from 'double' to 'int', possible loss of data
```

In general, narrowing conversions should be avoided, but there are situational cases where you might need to do one. In such cases, you should make the implicit narrowing conversion explicit by using `static_cast`. For example:

```

1 void someFcn(int i)
2 {
3 }
4
5 int main()
6 {
7     double d{ 5.0 };
8
9     someFcn(d); // bad: will generate compiler warning about narrowing conversion
    someFcn(static_cast<int>(d)); // good: we're explicitly telling the compiler this narrowing
    conversion is expected, no warning generated
10
    return 0;
}

```

Best practice

Avoid narrowing conversions whenever possible. If you do need to perform one, use `static_cast` to make it an explicit conversion.

Brace initialization disallows narrowing conversions

Narrowing conversions are strictly disallowed when using brace initialization (which is one of the primary reasons this initialization form is preferred):

```

1 int main()
2 {
3     int i { 3.5 }; // won't
    compile
}

```

Visual Studio produces the following error:

```
error C2397: conversion from 'double' to 'int' requires a narrowing conversion
```

More on numeric conversions

The specific rules for numeric conversions are complicated and numerous, so here are the most important things to remember.

In *all* cases, converting a value into a type whose range doesn't support that value will lead to results that are probably unexpected. For example:

```

1 int main()
2 {
3     int i{ 30000 };
4     char c = i; // chars have range -128 to
    127
5
6     std::cout << static_cast<int>(c);
7
8     return 0;
9 }

```

In this example, we've assigned a large integer to a variable with type `char` (that has range -128 to 127). This causes the char to overflow, and produces an unexpected result:

```
48
```

Converting from a larger integral or floating point type to a smaller type from the same family will generally work so long as the value fits in the range of the smaller type. For example:

```
1 | int i{ 2 };  
  | short s = i; // convert from int to  
2 | short  
  | std::cout << s << '\n';  
  
  | double d{ 0.1234 };  
  | float f = d;  
3 | std::cout << f << '\n';
```

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:

```
1 | float f = 0.123456789; // double value 0.123456789 has 9 significant digits, but float can only support  
  | about 7  
  | std::cout << std::setprecision(9) << f << '\n'; // 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 point type. For example:

```
1 | int i{ 10 };  
  | float f = i;  
2 | 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:

```
1 | int i = 3.5;  
  | std::cout << i <<  
2 | '\n';
```

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

While the numeric conversion rules might seem scary, in reality the compiler will generally warn you if you try to do something dangerous (excluding some signed/unsigned conversions).



Next lesson

8.4 Arithmetic conversions



Back to table of contents



Previous lesson

8.2 Floating-point and integral promotion


Leave a comment... Put C++ code between triple-backticks (markdown style): ``Your C++ code``

Name*

Email*

?

Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies: 

POST COMMENT

DP N N FOUT

Newest ▼

