# 10.3 — Numeric conversions



In the previous lesson (10.2 -- 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**. These numeric conversions cover additional type conversions between fundamental types.

Key insight

Any type conversion covered by the numeric promotion rules (10.2 -- 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):

```
short s = 3; // convert int to short
long l = 3; // convert int to long
char ch = s; // convert short to char
unsigned int u = 3; // convert int to unsigned int
```

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

```
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:

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

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

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

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

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

As an aside…

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

Safe and potentially unsafe conversions

Unlike a numeric promotion (which is always value-preserving and thus “safe”), some numeric conversions are not value-preserving in certain cases. Such conversions are said to be “unsafe” (even though “potentially unsafe” is more accurate, as these conversions are value-preserving in other cases).

Numeric conversions fall into one of three safety categories:

1. *Value-preserving conversions* are safe numeric conversions where the destination type can precisely represent all the values in the source type.

For example, `int` to `long` and `short` to `double` are safe conversions, as the source value can always be converted to an equivalent value of the destination type.

```
int main()
{
    int n { 5 };
    long l = n; // okay, produces long value 5

    short s { 5 };
    double d = s; // okay, produces double value 5.0

    return 0;
}
```

Compilers will typically not issue warnings for implicit value-preserving conversions.

A value converted using a value-preserving conversion can always be converted back to the source type, resulting in a value that is equivalent to the original value:

```
#include <iostream>

int main()
{
    int n = static_cast<int>(static_cast<long>(3)); // convert int 3 to long and back
    std::cout << n << '\n';                         // prints 3

    char c = static_cast<char>(static_cast<double>('c')); // convert 'c' to double
and back
    std::cout << c << '\n';                               // prints 'c'

    return 0;
}
```

2. *Reinterpretive conversions* are potentially unsafe numeric conversions where the result may be outside the range of the source type. Signed/unsigned conversions fall into this category.

For example, when converting a `signed int` to an `unsigned int`:

```
int main()
{
    int n1 { 5 };
    unsigned int u1 { n1 }; // okay: will be converted to unsigned int 5 (value
preserved)

    int n2 { -5 };
    unsigned int u2 { n2 }; // bad: will result in large integer outside range of
signed int

    return 0;
}
```

In the case of `u1`, the signed int value `5` is converted to unsigned int value `5`. Thus, the value is preserved in this case.

In the case of `u2`, the signed int value `-5` is converted to an unsigned int. Since an unsigned int can't represent negative numbers, the result will be a large integral value that is outside the range of a signed int. The value is not preserved in this case.

Such value changes are typically undesirable, and will often cause the program to exhibit unexpected or implementation-defined behavior.

Warning

Even though reinterpretive conversions are potentially unsafe, most compilers leave implicit signed/unsigned conversion warnings disabled by default.

This is because in some areas of modern C++ (such as when working with standard library arrays), signed/unsigned conversions can be hard to avoid. And practically speaking, the majority of such conversions do not actually result in a value change. Therefore, enabling such warnings can lead to many spurious warnings for signed/unsigned conversions that are actually okay (drowning out legitimate warnings).

If you choose to leave such warnings disabled, be extra careful of inadvertent conversions between these types (particularly when passing an argument to a function taking a parameter of the opposite sign).

Values converted using a reinterpretive conversion can be converted back to the source type, resulting in a value equivalent to the original value (even if the initial conversion produced a value out of range of the source type).

```
#include <iostream>

int main()
{
    int u = static_cast<int>(static_cast<unsigned int>(-5)); // convert '-5' to
unsigned and back
    std::cout << u << '\n'; // prints -5

    return 0;
}
```

For advanced readers

Prior to C++20, converting an unsigned value that is out-of-range of the signed value is technically undefined behavior (due to the allowance that signed integers could use a different binary representation than unsigned integers). In practice, this was a non-issue on modern system.

C++20 now requires that signed integers use 2s complement. As a result, the conversion rules were changed so that the above is now well-defined as a reinterpretive conversion.

Note that while such conversions are well defined, signed arithmetic overflow (which occurs when an arithmetic operation produces a value outside the range that can be stored) is still undefined behavior.

3. *Lossy conversions* are potentially unsafe numeric conversions where some data may be lost during the conversion.

For example, `double` to `int` is a conversion that may result in data loss:

```
int i = 3.0; // okay: will be converted to int value 3 (value preserved)
int j = 3.5; // data lost: will be converted to int value 3 (fractional value 0.5
lost)
```

Conversion from `double` to `float` can also result in data loss:

```
float f = 1.2;        // okay: will be converted to float value 1.2 (value preserved)
float g = 1.23456789; // data lost: will be converted to float 1.23457 (precision
lost)
```

Converting a value that has lost data back to the source type will result in a value that is different than the original value:

```
#include <iostream>

int main()
{
    double d { static_cast<double>(static_cast<int>(3.5)) }; // convert double 3.5 to
int and back
    std::cout << d << '\n'; // prints 3

    double d2 { static_cast<double>(static_cast<float>(1.23456789)) }; // convert
double 1.23456789 to float and back
    std::cout << d2 << '\n'; // prints 1.23457

    return 0;
}
```

For example, if `double` value `3.5` is converted to `int` value `3`, the fractional component `0.5` is lost. When `3` is converted back to a `double,` the result is `3.0`, not `3.5`.

Compilers will generally issue a warning (or in some cases, an error) when an implicit lossy conversion would be performed at runtime.

Warning

Some conversions may fall into different categories depending on the platform.

For example, `int` to `double` is typically a safe conversion, because `int` is usually 4 bytes and `double` is usually 8 bytes, and on such systems, all possible `int` values can be represented as a `double`. However, there are architectures where both `int` and `double` are 8 bytes. On such architectures, `int` to `double` is a lossy conversion!

We can demonstrate this by casting a long long value (which must be at least 64 bits) to double and back:

```
#include <iostream>

int main()
{
    std::cout << static_cast<long long>(static_cast<double>(10000000000000001LL));

    return 0;
}
```

This prints:

```
10000000000000000
```

Note that our last digit has been lost!

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:

```
int main()
{
    int i{ 30000 };
    char c = i; // chars have range -128 to 127

    std::cout << static_cast<int>(c) << '\n';

    return 0;
}
```

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
```

Remember that overflow is well-defined for unsigned values and produces undefined behavior for signed values.

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:

```
int i{ 2 };
short s = i; // convert from int to short
std::cout << s << '\n';

double d{ 0.1234 };
float f = d;
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:

```
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:

```
int i{ 10 };
float f = i;
std::cout << f << '\n';
```

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 << '\n';
```

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

```
3
```

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).