10.1 — Implicit type conversion

Introduction to type conversion

The value of an object is stored as a sequence of bits, and the data type of the object 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` might be stored as binary `0000 0000 0000 0000 0000 0000 0000 0011`, whereas floating point value `3.0` might be stored as binary `0100 0000 0100 0000 0000 0000 0000 0000`.

So what happens when we do something like this?

```
float f{ 3 }; // initialize floating point variable with int 3
```

In such a case, the compiler can't just copy the bits representing the `int` value `3` into the memory allocated for `float` variable `f`. Instead, it needs to convert the integer value `3` to the equivalent floating point value `3.0`, which can then be stored in the memory allocated for `f`.

The process of producing a new value of some type from a value of a different type is called a **conversion**.

Key insight

Conversions do not change the value or type being converted. Instead, a new value with the desired type is created as a result of the conversion.

Type conversion can be invoked in one of two ways: either implicitly (as needed by the compiler), or explicitly (when requested by the programmer). We'll cover implicit type conversion in this lesson, and explicit type conversions (casting) in upcoming lesson 10.6 -- Explicit type conversion (casting) and static_cast.

Implicit type conversion

Implicit type conversion (also called **automatic type conversion** or **coercion**) is performed automatically by the compiler when one data type is required, but a different data type is supplied. The vast majority of type conversions in C++ are implicit type conversions. For example, implicit type conversion happens in all of the following cases:

When initializing (or assigning a value to) a variable with a value of a different data type:

```
double d{ 3 }; // int value 3 implicitly converted to type double
d = 6; // int value 6 implicitly converted to type double
```

When the type of a return value is different from the function's declared return type:

```
float doSomething()
{
    return 3.0; // double value 3.0 implicitly converted to type float
}
```

When using certain binary operators with operands of different types:

```
double division{ 4.0 / 3 }; // int value 3 implicitly converted to type double
```

When using a non-Boolean value in an if-statement:

```
if (5) // int value 5 implicitly converted to type bool
{
}
```

When an argument passed to a function is a different type than the function parameter:

```
void doSomething(long l)
{
}

doSomething(3); // int value 3 implicitly converted to type long
```

What happens when a type conversion is invoked

When a type conversion is invoked (whether implicitly or explicitly), the compiler will determine whether it can convert the value from the current type to the desired type. If a valid conversion can be found, then the compiler will produce a new value of the desired type. Note that type conversions don't change the value or type of the value or object being converted.

If the compiler can't find an acceptable conversion, then the compilation will fail with a compile error. Type conversions can fail for any number of reasons. For example, the compiler might not know how to convert a value between the original type and the desired type. In other cases, statements may disallow certain types of conversions. For example:

```
int x { 3.5 }; // brace-initialization disallows conversions that result in data loss
```

Even though the compiler knows how to convert a `double` value to an `int` value, such conversions are disallowed when using brace-initialization.

There are also cases where the compiler may not be able to figure out which of several possible type conversions is unambiguously the best one to use. We'll see examples of this in lesson 11.3 -- Function overload resolution and ambiguous matches.

So how does the compiler actually determine whether it can convert a value from one type to another?

The standard conversions

The C++ language standard defines how different fundamental types (and in some cases, compound types) can be converted to other types. These conversion rules are called the **standard conversions**.

The standard conversions can be broadly divided into 4 categories, each covering different types of conversions:

- Numeric promotions (covered in lesson 10.2 -- Floating-point and integral promotion)
- Numeric conversions (covered in lesson 10.3 -- Numeric conversions)
- Arithmetic conversions (covered in lesson 10.5 -- Arithmetic conversions)
- Other conversions (which includes various pointer and reference conversions)

When a type conversion is needed, the compiler will see if there are standard conversions that it can use to convert the value to the desired type. The compiler may apply zero, one, or more than one standard conversions in the conversion process.

As an aside…

How do you have a type conversion with zero conversions? As an example, on architectures where `int` and `long` both have the same size and range, the same sequence of bits is used to represent values of both types. Therefore, no actual conversion is needed to convert a value between those types -- the value can simply be copied.

The full set of rules describing how type conversions work is both lengthy and complicated, and for the most part, type conversion "just works". In the next set of lessons, we'll cover the most important things you need to know about type conversions. If finer detail is required for some uncommon case, the full rules are detailed in technical reference documentation for implicit conversions.

Let's get to it!