4.12 — Introduction to type conversion and static_cast

Implicit type conversion

Consider the following program:

```
#include <iostream>

void print(double x) // print takes a double parameter
{
	std::cout << x << '\n';
}

int main()
{
	print(5); // what happens when we pass an int value?

	return 0;
}
```

In the above example, the `print()` function has a parameter of type `double` but the caller is passing in the value `5` which is of type `int`. What happens in this case?

In most cases, C++ will allow us to convert values of one fundamental type to another fundamental type. The process of converting a value from one type to another type is called **type conversion**. Thus, the int argument `5` will be converted to double value `5.0` and then copied into parameter `x`. The `print()` function will print this value, resulting in the following output:

```
5
```

A reminder

By default, floating point values whose decimal part is 0 print without the decimal places (e.g. `5.0` prints as `5`).

When the compiler does type conversion on our behalf without us explicitly asking, we call this **implicit type conversion**. The above example illustrates this -- nowhere do we explicitly tell the compiler to convert integer value `5` to double value `5.0`. Rather, the function is expecting a double value, and we pass in an integer argument. The compiler will notice the mismatch and implicitly convert the integer to a double.

Here’s a similar example where our argument is an int variable instead of an int literal:

```
#include <iostream>

void print(double x) // print takes a double parameter
{
	std::cout << x << '\n';
}

int main()
{
	int y { 5 };
	print(y); // y is of type int

	return 0;
}
```

This works identically to the above. The value held by int variable `y` (`5`) will be converted to double value `5.0`, and then copied into parameter `x`.

Type conversion produces a new value

Even though it is called a conversion, a type conversion does not actually change the value or type of the value being converted. Instead, the value to be converted is used as input, and the conversion results in a new value of the target type (via direct initialization).

In the above example, the conversion does not change variable `y` from type `int` to `double`. Instead, the conversion uses the value of `y` (`5`) as input to create a new double value (`5.0`). This double value is then passed to function `print`.

Key insight

Type conversion uses direct initialization to produce a new value of the target type from a value of a different type.

Implicit type conversion warnings

Although implicit type conversion is sufficient for most cases where type conversion is needed, there are a few cases where it is not. Consider the following program, which is similar to the example above:

```
#include <iostream>

void print(int x) // print now takes an int parameter
{
	std::cout << x << '\n';
}

int main()
{
	print(5.5); // warning: we're passing in a double value

	return 0;
}
```

In this program, we've changed `print()` to take an `int` parameter, and the function call to `print()` is now passing in `double` value `5.5`. Similar to the above, the compiler will use implicit type conversion in order to convert double value `5.5` into a value of type `int`, so that it can be passed to function `print()`.

Unlike the initial example, when this program is compiled, your compiler will generate some kind of a warning about a possible loss of data. And because you have "treat warnings as errors" turned on (you do, right?), your compiler will abort the compilation process.

Tip

You'll need to disable "treat warnings as errors" temporarily if you want to compile this example. See lesson 0.11 -- Configuring your compiler: Warning and error levels for more information about this setting.

When compiled and run, this program prints the following:

```
5
```

Note that although we passed in value `5.5`, the program printed `5`. Because integral values can't hold fractions, when double value `5.5` is implicitly converted to an `int`, the fractional component is dropped, and only the integral value is retained.

Because converting a floating point value to an integral value results in any fractional component being dropped, the compiler will warn us when it does an implicit type conversion from a floating point to an integral value. This happens even if we were to pass in a floating point value with no fractional component, like `5.0` -- no actual loss of value occurs during the conversion to integral value `5` in this specific case, but the compiler may still warn us that the conversion is unsafe.

Key insight

Some type conversions are always safe to make (such as `int` to `double`), whereas others may result in the value being changed during conversion (such as `double` to `int`). Unsafe implicit conversions will typically either generate a compiler warning, or (in the case of brace initialization) an error.

This is one of the primary reasons brace initialization is the preferred initialization form. Brace initialization will ensure we don't try to initialize a variable with an initializer that will lose value when it is implicitly type converted:

```
int main()
{
    double d { 5 }; // okay: int to double is safe
    int x { 5.5 }; // error: double to int not safe

    return 0;
}
```

Related content

Implicit type conversion is a meaty topic. We dig into this topic in more depth in future lessons, starting with lesson 10.1 -- Implicit type conversion.

An introduction to explicit type conversion via the static_cast operator

Back to our most recent `print()` example, what if we *intentionally* wanted to pass a double value to a function taking an integer (knowing that the converted value would drop any fractional component?) Turning off "treat warnings as errors" just to make our program compile is a bad idea, because then we'll have warnings every time we compile (which we will quickly learn to ignore), and we risk overlooking warnings about more serious issues.

C++ supports a second method of type conversion, called explicit type conversion. **Explicit type conversion** allow us (the programmer) to explicitly tell the compiler to convert a value from one type to another type, and that we take full responsibility for the result of that conversion. If such a conversion results in the loss of value, the compiler will not warn us.

To perform an explicit type conversion, in most cases we'll use the `static_cast` operator. The syntax for the `static_cast` looks a little funny:

```
static_cast<new_type>(expression)
```

static_cast takes the value from an expression as input, and returns that value converted into the type specified by *new_type* (e.g. int, bool, char, double).

Key insight

Whenever you see C++ syntax (excluding the preprocessor) that makes use of angled brackets (<>), the thing between the angled brackets will most likely be a type. This is typically how C++ deals with code that need a parameterized type.

Let's update our prior program using `static_cast`:

```
#include <iostream>

void print(int x)
{
	std::cout << x << '\n';
}

int main()
{
	print( static_cast<int>(5.5) ); // explicitly convert double value 5.5 to an int

	return 0;
}
```

Because we're now explicitly requesting that double value `5.5` be converted to an `int` value, the compiler will not generate a warning about a possible loss of data upon compilation (meaning we can leave "treat warnings as errors" enabled).

Related content

C++ supports other types of casts. We talk more about the different types of casts in future lesson 10.6 -- Explicit type conversion (casting) and static_cast.

Using static_cast to convert char to int

In the lesson on chars 4.11 -- Chars, we saw that printing a char value using `std::cout` results in the value being printed as a char:

```
#include <iostream>

int main()
{
    char ch{ 97 }; // 97 is ASCII code for 'a'
    std::cout << ch << '\n';

    return 0;
}
```

This prints:

```
a
```

If we want to print the integral value instead of the char, we can do this by using `static_cast` to cast the value from a `char` to an `int`:

```
#include <iostream>

int main()
{
    char ch{ 97 }; // 97 is ASCII code for 'a'
    std::cout << ch << " has value " << static_cast<int>(ch) << '\n'; // print value
of variable ch as an int

    return 0;
}
```

This prints:

```
a has value 97
```

It's worth noting that the argument to *static_cast* evaluates as an expression. When we pass in a variable, that variable is evaluated to produce its value, and that value is then converted to the new type. The variable itself is *not* affected by casting its value to a new type. In the above case, variable `ch` is still a char, and still holds the same value even after we've cast its value to an `int`.

Converting unsigned numbers to signed numbers

To convert an unsigned number to a signed number, you can also use the `static_cast` operator:

```
#include <iostream>

int main()
{
    unsigned int u { 5 };
    int s { static_cast<int>(u) }; // return value of variable u as an int

    std::cout << s << '\n';
    return 0;
}
```

The `static_cast` operator doesn't do any range checking, so if you cast a value to a type whose range doesn't contain that value, undefined behavior will result. Therefore, the above cast from `unsigned int` to `int` will yield unpredictable results if the value of the `unsigned int` is greater than the maximum value a signed `int` can hold.

Warning

The static_cast operator will produce undefined behavior if the value being converted doesn't fit in range of the new type.

std::int8_t and std::uint8_t likely behave like chars instead of integers

As noted in lesson 4.6 -- Fixed-width integers and size_t, most compilers define and treat `std::int8_t` and `std::uint8_t` (and the corresponding fast and least fixed-width types) identically to types `signed char` and `unsigned char` respectively. Now that we've covered what chars are, we can demonstrate where this can be problematic:

```
#include <cstdint>
#include <iostream>

int main()
{
    std::int8_t myInt{65};      // initialize myInt with value 65
    std::cout << myInt << '\n'; // you're probably expecting this to print 65

    return 0;
}
```

Because `std::int8_t` describes itself as an int, you might be tricked into believing that the above program will print the integral value `65`. However, on most systems, this program will print `A` instead (treating `myInt` as a `signed char`). However, this is not guaranteed (on some systems, it may actually print `65`).

If you want to ensure that a `std::int8_t` or `std::uint8_t` object is treated as an integer, you can convert the value to an integer using `static_cast`:

```
#include <cstdint>
#include <iostream>

int main()
{
    std::int8_t myInt{65};
    std::cout << static_cast<int>(myInt) << '\n'; // will always print 65

    return 0;
}
```

In cases where `std::int8_t` is treated as a char, input from the console can also cause problems:

```
#include <cstdint>
#include <iostream>

int main()
{
    std::cout << "Enter a number between 0 and 127: ";
    std::int8_t myInt{};
    std::cin >> myInt;

    std::cout << "You entered: " << static_cast<int>(myInt) << '\n';

    return 0;
}
```

A sample run of this program:

```
Enter a number between 0 and 127: 35
You entered: 51
```

Here's what's happening. When `std::int8_t` is treated as a char, the input routines interpret our input as a sequence of characters, not as an integer. So when we enter `35`, we're actually entering two chars, `'3'` and `'5'`. Because a char object can only hold one character, the `'3'` is extracted (the `'5'` is left in the input stream for possible extraction later). Because the char `'3'` has ASCII code point 51, the value `51` is stored in `myInt`, which we then print later as an int.

In contrast, the other fixed-width types will always print and input as integral values.

Quiz time

Question #1

Write a short program where the user is asked to enter a single character. Print the value of the character and its ASCII code, using `static_cast`.

The program's output should match the following:

```
Enter a single character: a
You entered 'a', which has ASCII code 97.
```

Show Solution

Question #2

Modify the program you wrote for quiz #1 to use implicit type conversion instead of `static_cast`. How many different ways can you think of to do this?

Note: You should favor explicit conversions over implicit conversions, so don’t actually do this in real programs -- this is just to test your understanding of where implicit conversions can occur.

Show Solution