10.6 — Explicit type conversion (casting) and static_cast

ALEX SEPTEMBER 11, 2023

In lesson 10.1 -- Implicit type conversion (https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/), we discussed that the compiler can implicitly convert a value from one data type to another through a system called `implicit type conversion`. When you want to numerically promote a value from one data type to a wider data type, using implicit type conversion is fine.

Many new C++ programmers try something like this:

```
double d = 10 / 4; // does integer division, initializes d with value 2.0
```

Because `10` and `4` are both of type `int`, integer division is performed, and the expression evaluates to `int` value `2`. This value then undergoes numeric conversion to `double` value `2.0` before being used to initialize variable `d`. Most likely, this isn't what was intended.

In the case where you are using literal operands, replacing one or both of the integer literals with double literals will cause floating point division to happen instead:

```
double d = 10.0 / 4.0; // does floating point division, initializes d with value 2.5
```

But what if you are using variables instead of literals? Consider this case:

```
int x { 10 };
int y { 4 };
double d = x / y; // does integer division, initializes d with value 2.0
```

Because integer division is used here, variable `d` will end up with the value of `2.0`. How do we tell the compiler that we want to use floating point division instead of integer division in this case? Literal suffixes can't be used with variables. We need some way to convert one (or both) of the variable operands to a floating point type, so that floating point division will be used instead.

Fortunately, C++ comes with a number of different **type casting operators** (more commonly called **casts**) that can be used by the programmer to request that the compiler perform a type conversion. Because casts are explicit requests by the programmer, this form of type conversion is often called an **explicit type conversion** (as opposed to implicit type conversion, where the compiler performs a type conversion automatically).

Type casting

C++ supports 5 different types of casts: `C-style casts`, `static casts`, `const casts`, `dynamic casts`, and `reinterpret casts`. The latter four are sometimes referred to as **named casts**.

We'll cover `C-style casts` and `static casts` in this lesson.

Related content

We discuss dynamic casts in lesson 25.10 -- Dynamic casting (https://www.learncpp.com/cpp-tutorial/dynamic-casting/), after we've covered other prerequisite topics.

`Const casts` and `reinterpret casts` should generally be avoided because they are only useful in rare cases and can be harmful if used incorrectly.

Warning

Avoid const casts and reinterpret casts unless you have a very good reason to use them.

C-style casts

In standard C programming, casts are done via the () operator, with the name of the type to convert the value placed inside the parentheses. You may still see these used in code (or by programmers) that have been converted from C.

For example:

```
#include <iostream>

int main()
{
    int x { 10 };
    int y { 4 };

    double d { (double)x / y }; // convert x to a double so we get floating point division
    std::cout << d << '\n'; // prints 2.5

    return 0;
}
```

In the above program, we use a C-style cast to tell the compiler to convert `x` to a `double`. Because the left operand of operator/ now evaluates to a floating point value, the right operand will be converted to a floating point value as well, and the division will be done using floating point division instead of integer division!

C++ will also let you use a `C-style cast` with a more function-call like syntax:

```
    double d { double(x) / y }; // convert x to a double so we get floating point division
```

This performs identically to the prior example, but has the benefit of parenthesizing the value being converted (making it easier to tell what is being converted).

Although a `C-style cast` appears to be a single cast, it can actually perform a variety of different conversions depending on context. This can include a `static cast`, a `const cast` or a `reinterpret cast` (the latter two of which we mentioned above you should avoid). As a result, `C-style casts` are at risk for being inadvertently misused and not producing the expected behavior, something which is easily avoidable by using the C++ casts instead.

Also, because C-style casts are just a type name, parenthesis, and variable or value, they are both difficult to identify (making your code harder to read) and even more difficult to search for.

Related content

If you're curious, this article (https://anteru.net/blog/2007/c-background-static-reinterpret-and-c-style-casts/) has more information on how C-style casts actually work.

Best practice

Avoid using C-style casts.

---

static_cast

C++ introduces a casting operator called **static_cast**, which can be used to convert a value of one type to a value of another type.

You've previously seen `static_cast` used to convert a `char` into an `int` so that std::cout prints it as an integer instead of a `char`:

```
#include <iostream>

int main()
{
    char c { 'a' };
    std::cout << c << ' ' << static_cast<int>(c) << '\n'; // prints a 97

    return 0;
}
```

The `static_cast` operator takes an expression as input, and returns the evaluated value converted to the type specified inside the angled brackets. `static_cast` is best used to convert one fundamental type into another.

```
#include <iostream>

int main()
{
    int x { 10 };
    int y { 4 };

    // static cast x to a double so we get floating point division
    double d { static_cast<double>(x) / y };
    std::cout << d << '\n'; // prints 2.5

    return 0;
}
```

The main advantage of `static_cast` is that it provides compile-time type checking, making it harder to make an inadvertent error.

```
    // a C-style string literal can't be converted to an int, so the following is an invalid conversion
    int x { static_cast<int>("Hello") }; // invalid: will produce compilation error
```

`static_cast` is also (intentionally) less powerful than `C-style casts`, so you can't inadvertently remove `const` or do other things you may not have intended to do.

```
int main()
{
    const int x{ 5 };
    int& ref{ static_cast<int&>(x) }; // invalid: will produce compilation error
    ref = 6;

    return 0;
}
```

Best practice

Favor static_cast when you need to convert a value from one type to another type.

Using static_cast to make narrowing conversions explicit

Compilers will often issue warnings when a potentially unsafe (narrowing) implicit type conversion is performed. For example, consider the following program:

```
int i { 48 };
char ch = i; // implicit narrowing conversion
```

Casting an `int` (2 or 4 bytes) to a `char` (1 byte) is potentially unsafe (as the compiler can't tell whether the integer value will overflow the range of the `char` or not), and so the compiler will typically print a warning. If we used list initialization, the compiler would yield an error.

To get around this, we can use a static cast to explicitly convert our integer to a `char`:

```
int i { 48 };

// explicit conversion from int to char, so that a char is assigned to variable ch
char ch { static_cast<char>(i) };
```

When we do this, we're explicitly telling the compiler that this conversion is intended, and we accept responsibility for the consequences (e.g. overflowing the range of a `char` if that happens). Since the output of this `static_cast` is of type `char`, the initialization of variable `ch` doesn't generate any type mismatches, and hence no warnings or errors.

Here's another example where the compiler will typically complain that converting a `double` to an `int` may result in loss of data:

```
int i { 100 };
i = i / 2.5;
```

To tell the compiler that we explicitly mean to do this:

```
int i { 100 };
i = static_cast<int>(i / 2.5);
```

Quiz time

Question #1

What's the difference between implicit and explicit type conversion?

Show Solution (javascript:void(0))

Next lesson
10.7 Typedefs and type aliases

Back to table of contents

Previous lesson
10.5 Arithmetic conversions

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

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

213 COMMENTS

Newest