6.16 — Explicit type conversion (casting) and static_cast

BY ALEX ON APRIL 16TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

In the previous lesson **6.15 -- Implicit type conversion (coercion)**, 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 promote a value from one data type to a larger similar data type, using `implicit type conversion` is fine.

However, many new programmers try something like this: `float f = 10 / 4;`. However, because `10` and `4` are both integers, no promotion takes place. Integer division is performed on `10 / 4`, resulting in the value of 2, which is then implicitly converted to `2.0` and assigned to variable `f`!

In the case where you are using literal values (such as `10`, or `4`), replacing one or both of the integer literal value with a floating point literal value (`10.0` or `4.0`) will cause both operands to be converted to floating point values, and the division will be done using floating point math (and thus retain the fractional component).

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

```
int i1 { 10 };
int i2 { 4 };
float f { i1 / i2 };
```

Variable `f` will end up with the value of 2. How do we tell the compiler that we want to use floating point division instead of integer division? The answer is by using a `type casting operator` (more commonly called a `cast`) to tell the compiler to do **explicit type conversion**. A **cast** represents an request by the programmer to do an `explicit type conversion`.

Type casting

In C++, there are 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. `Dynamic casts` we'll save until after we cover `pointers` and `inheritance` in future lessons.

`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 to placed inside the parenthesis. For example:

```
int i1 { 10 };
int i2 { 4 };
float f { (float)i1 / i2 };
```

In the above program, we use a float C-style cast to tell the compiler to convert `i1` to a floating point value. Because the left operand of operator/ now evaluates to a floating point value, the right operator 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:

```
int i1 { 10 };
int i2 { 4 };
float f { float(i1) / i2 };
```

This performs identically to the prior example.

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.

As an aside...

If you're curious, **this article** has more information on how C-style casts actually work.

Warning

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`:

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

The `static_cast` operator takes a single value as input, and outputs the same value converted to the type specified inside the angled brackets. `Static_cast` is best used to convert one fundamental type into another.

```
int i1 { 10 };
int i2 { 4 };

// convert an int to a float so we get floating point division rather than integer division
float f { static_cast<float>(i1) / i2 };
```

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

Best practice

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

Using casts to make implicit type conversions clear

Compilers will often complain when an unsafe implicit type conversion is performed. For example, consider the following program:

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

Casting an int (4 bytes) to a char (1 byte) is potentially unsafe (as the compiler can't tell whether the integer will overflow the range of the char or not), and so the compiler will typically complain.

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 assignment to variable `ch` doesn't generate any type mismatches, and hence no warnings.

In the following program, 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


6.17 -- Unnamed and inline namespaces


Index


6.15 -- Implicit type conversion (coercion)

C++ TUTORIAL | PRINT THIS POST

105 comments to 6.16 — Explicit type conversion (casting) and static_cast

« Older Comments 1 2

Gabe
February 5, 2020 at 10:16 am · Reply

How would a typecast like this work:

```
1 uintptr_t  address = 0x23B30; (BYTE*)address;
```

I have seen this done many times but it still evades me as to how you can cast a non-pointer to a pointer. How would this work.

nascardriver
February 6, 2020 at 9:15 am · Reply

A pointer is just an address. You can cast them to integers and back. If the address you're casting doesn't point to an object of the pointer type, you're causing undefined behavior.

C-style casts are unsafe, they allow conversions that cause UB. You can do the same with a `reinterpret_cast`, but it's more obvious that this isn't a safe cast.

Gabe
February 6, 2020 at 9:50 am · Reply

uintptr_t is not a pointer in and of itself, but capable of storing a pointer. I think it was misleading what I wrote.

```
1 uintptr_t number = 22; (BYTE*)number;
```

so uintptr_t is a regular data type, unsigned long i believe.

But this doesn't make sense to me, because it is never taught how a non-pointer to a pointer cast works.

nascardriver
February 7, 2020 at 8:18 am · Reply

It's not taught, because it's unsafe and rarely useful. Unless you're doing low-level memory operation (Potentially intentionally invoking UB), there's no need to cast pointers to integers and back.

Gabe
February 7, 2020 at 11:14 am · Reply

Vice-versa, i mean. Integers to pointers. This is what I need to know. I understand now cstyle casts should generally be avoided.. But I need to know how this particular operation works to see if there is a better cast for it.

And yes this is dealing with low-level memory operations.

nascardriver
February 8, 2020 at 3:29 am · Reply

Enter undefined behavior land. Virtually every compiler does what you want, but it doesn't have to.
A pointer is nothing but a number (The memory address). When you use a pointer, the memory at that address is read from, or written to.

```
// 0x12345678 is an address in your process (I suppose the process you
std::uintptr_t uiHealth{ 0x12345678 };

// Use a C-style cast, or reinterpret_cast (preferred), we can
// turn the integral address into a pointer to whatever type we want.
auto* pHealth{ reinterpret_cast<int*>(uiHealth) };

// And use the pointer as if it was valid.
*pHealth = 100;
```

Unlike a `static_cast`, a `reinterpret_cast` changes only the type, not the data.

```
auto* pHealth{ reinterpret_cast<int*>(uiHealth) };

// Cast an int* to a float*. Doesn't change the data.
auto* pHealth2{ reinterpret_cast<float*>(pHealth) };
```

kitabski
November 7, 2019 at 2:03 am · Reply

First of all, Thanks for awesome tutorial!!!
Can anyone help me?
So the result of the follwing is 2.5

```
#include <iostream>
using namespace std;
int main()
{
    int i1 {10};
    int i2 {4};
    float f = static_cast <float> (i1)/i2;
    cout << f << endl;
```

```
    return 0;
}
```

But if i change it a bit and put i1 and i2 into brackets (i1/i2), the result is 2.
Why??????

nascardriver
November 7, 2019 at 2:52 am · Reply

```
static_cast<float>(i1 / i2)
```

`i1 / i2` is performed first. Since both are `int`, the result is an `int` (2). You're then casting the 2 to `float`, but the precision has already been lost.

```
static_cast<float>(i1) / i2
```

`i1` is cast to `float` first. Then you're dividing `float` by `int`, which produces a `float`.

kitabski
November 7, 2019 at 3:10 am · Reply

Thank you very much for clarification!!!

Radioga
October 7, 2019 at 9:55 am · Reply

I have a doubt about this sentence: "Because C-style casts are not checked by the compiler at compile time, C-style casts can be inherently misused". In my experience the C-style cast is checked by the compiler; for example this little program cannot compile:

```
int main()
{
    float f{ 5.0f };
    char** s{ nullptr };
    s = (char**)f;
}
```

the output console of Visual Studio 2017 says me:

error C2440: 'type cast': cannot convert from 'float' to 'char **'

Alex
October 10, 2019 at 12:59 pm · Reply

I've updated the article a bit with better reasons not to use C-style casts, and added a link with more detail on how C-style casts work for the curious. Thanks!

daniel jo
June 27, 2019 at 7:25 am · Reply

"Static_cast takes as input a values" this bit confuses me.. can u explain what u mean

nascardriver

June 27, 2019 at 7:30 am · Reply

Yes he does. @Alex

daniel jo
June 27, 2019 at 8:43 am · Reply

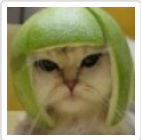
?

nascardriver
June 27, 2019 at 8:52 am · Reply

Uhm, sorry, my reply doesn't make a lot of sense when I read the message wrong.
He means
"static_cast takes as input a value".
Single value, not values.

daniel jo
June 27, 2019 at 9:09 am · Reply

ahhh.. makes sense now thank you

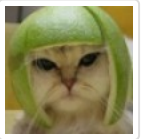


Alex
June 28, 2019 at 2:39 pm · Reply

Typo fixed. Thanks!

Q
June 2, 2019 at 7:03 pm · Reply

Can you give more examples of why c++ style cast should be preferred? C style casts have always worked fine for c and java for that matter. I don't understand why they took something simple from C and replaced it with this over complicated ugly, and cumbersome mess. What ever happened to if it ain't broke don't fix it?



Alex
June 11, 2019 at 2:29 pm · Reply

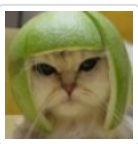
C++ style casts should be preferred because:
1) They allow you to better express your intent.
2) C-style casts might do any number of things (and sometimes different things depending on context), and it's not always clear which one is being invoked from reading the code.
3) They're easier to find in your code precisely because they're "ugly" and differentiated.
4) At least in the case of dynamic_cast<>, they can do things that C-style casts can't.

There's nothing that says you have to give up C-style casts if you want to, but C++ styles casts are safer. https://anteru.net/blog/2007/c-background-static-reinterpret-and-c-style-casts/ has a little bit of additional information as well as an example of where C-style casts can go wrong.

James
April 27, 2019 at 11:56 am · Reply

```
In order to announce to the compiler that you are explicitly doing something you recognize is potentially unsafe (but want to do anyway), you can use a cast
```

I believe that's not actually what happened there. "static_cast" is an unary operator with high precedence level(2) which takes in one expression of a type and return that expression with the type programmer desired. So, compiler giving a warning for doing unsafe conversion on type-converting operator which purpose is to convert type makes no sense, so it won't. And when we do "i = static_cast<int>(i / 2.5);", "static_cast<int>(i/2.5);" will be evaluated first since "static_cast" operator has higher precedence lever than "operator=". When compiler evaluate "operator=", it see an int being assigned to an int so no warning.
I've confirmed it by using -Wconversion I found in a comment by @nascardriver. Try something like "float x= 3; int y = static_cast<double>(x);" and see your compiler still give warning even when you're using "static_cast".



Alex
May 1, 2019 at 2:18 pm · Reply

I take your point -- the original wording left something to be desired. I've updated the lesson text to be more explanatory around what's happening without losing the "programmer is taking responsibility" angle.

Anonymous
March 5, 2019 at 2:49 am · Reply

Why should casting be avoided? Wouldn't it be better to use (static) casting always instead of letting it happen implicitly?

nascardriver
March 6, 2019 at 4:14 am · Reply

If you convert types, use a cast. What Alex is trying to say it that you should try to stay out of situations where you'd need a cast by using the same type.

Anonymous
March 6, 2019 at 9:27 am · Reply

Understood, thanks!
Also, is using smaller variables for values that are going to be a part of computation with longer variables redundant, since they are going to get promoted anyway?

nascardriver
March 6, 2019 at 10:06 am · Reply

Implicit conversion can be the cause of trouble. If you can, use the same type throughout your computation.

jih332
February 25, 2019 at 3:46 pm · Reply

Hi there,

You said that "In the following program, 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;
```

however I didn't find any issue by doing this on my side, I'm using

```
g++ main.cpp -o test -pedantic-errors -Wall -Weffc++ -Wextra -Wsign-conversion -Werror -std
```

to compile my code. what could be the reason for this? Thanks in advance.

nascardriver
February 26, 2019 at 3:27 am · Reply

Add

```
-Wconversion
```

@Alex

While you're add it, you can also use a newer standard

```
-std=c++2a
// Or
-std=c++17
```

jih332
February 26, 2019 at 7:01 pm · Reply

got it. thanks a lot!

NAMIT KHANDUJA
February 17, 2019 at 8:49 am · Reply

what is type cast operator?

```
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;    // calls constructor
```

```
  bar = foo;     // calls assignment
  foo = bar;     // calls type-cast operator
  return 0;
}
```

Benur21
February 10, 2019 at 2:18 pm · Reply

Can we use

```
1 int i1 = 10;
2 int i2 = 4;
3 float f = (float)i1 / i2;
```

in C++?

Also, what do you mean by getting rid of a const?

« Older Comments 1 2