6.15 — Implicit type conversion (coercion)

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

Previously, you learned that a value of a variable is stored as a sequence of bits, and the data type of the variable 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 and the float value 3.0 are stored as completely different binary patterns.

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 `float` variable `f`. Instead, it needs to convert the integer value 3 to a floating point number, which can then be assigned to `float` variable `f`.

The process of converting a value from one data type to another is called a **type conversion**. `Type conversions` can happen in many different cases:

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

```
double d{ 3 }; // initialize double variable with integer value 3
d = 6; // assign double variable the integer value 6
```

- When passing a value to a function where the function parameter is of a different data type:

```
void doSomething(long l)
{
}

doSomething(3); // pass integer value 3 to a function expecting a long parameter
```

- When returning a value from a function where the function return type is of a different data type:

```
float doSomething()
{
    return 3.0; // Return double value 3.0 back to caller through float return type
}
```

- Using a binary operator with operands of different types:

```
double division{ 4.0 / 3 }; // division with a double and an integer
```

In all of these cases (and quite a few others), C++ will use `type conversion` to convert one data type to another data type.

There are two basic types of `type conversion`: `implicit type conversion`, where the compiler automatically transforms one data type into another, and `explicit type conversion`, where the developer uses a casting operator to direct the conversion.

We'll cover `implicit type conversion` in this lesson, and `explicit type conversion` in the next.

Implicit type conversion

Implicit type conversion (also called **automatic type conversion** or **coercion**) is performed whenever one data type is expected, but a different data type is supplied. If the compiler can figure out how to do the conversion between the two types, it will. If it doesn't know how, then it will fail with a compile error.

All of the above examples are cases where `implicit type conversion` will be used.

There are two basic types of `implicit type conversion`: `promotions` and `conversions`.

Numeric promotion

Whenever a value from one fundamental data type is converted into a value of a larger fundamental data type from the same family, this is called a **numeric promotion** (or **widening**, though this term is usually reserved for integers). For example, an `int` can be widened into a `long`, or a `float` promoted into a `double`:

```
long l{ 64 }; // widen the integer 64 into a long
double d{ 0.12f }; // promote the float 0.12 into a double
```

While the term `numeric promotion` covers *any* type of promotion, there are two other terms with specific meanings in C++:

- **Integral promotion** involves the conversion of integer types narrower than `int` (which includes `bool`, `char`, `unsigned char`, `signed char`, `unsigned short`, and `signed short`) to an `int` (if possible) or an `unsigned int` (otherwise).
- **Floating point promotion** involves the conversion of a `float` to a `double`.

`Integral promotion` and `floating point promotion` are used in specific cases to convert smaller data types to `int`/`unsigned int` or `double`, because `int` and `double` are generally the most performant types to perform operations on.

The important thing to remember about promotions is that they are always safe, and no data loss will result.

For advanced readers

Under the hood, promotions generally involve extending the binary representation of a number (e.g. for integers, adding leading 0s).

Numeric conversions

When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion**. For example:

```
double d{ 3 }; // convert integer 3 to a double (between different types)
short s{ 2 }; // convert integer 2 to a short (from larger to smaller type within same type fam
```

Unlike promotions, which are always safe, conversions may or may not result in a loss of data. Because of this, code that causes an implicit conversion to be performed will often cause the compiler to issue a warning.

The rules for conversions are complicated and numerous, so we'll just cover the common cases here.

In *all* cases, converting a value into a type that doesn't have a large enough range to support the value will lead to unexpected results. For example:

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

    std::cout << static_cast<int>(c);

    return 0;
```

```
}
```

In this example, we've assigned a large integer to a char (that has range -128 to 127). This causes the char to overflow, and produces an unexpected result:

```
48
```

However, converting from a larger integral or floating point type to a smaller similar type 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
    std::cout << std::setprecision(9) << f << '\n'; // std::setprecision defined in iomanip hea
```

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 type. For example:

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

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
```

Conversions that could cause loss of information, e.g. floating point to integer, are called **narrowing conversions**. Since information loss is generally undesirable, brace initialization doesn't allow narrowing conversions.

```
double d{ 10.0 };
int i{ d }; // Error: A double can store values that don't fit into an int
```

Evaluating arithmetic expressions

When evaluating expressions, the compiler breaks each expression down into individual subexpressions. The arithmetic operators require their operands to be of the same type. To ensure this, the compiler uses the following rules:

- If an operand is an integer that is narrower than an int, it undergoes integral promotion (as described above) to int or unsigned int.
- If the operands still do not match, then the compiler finds the highest priority operand and implicitly converts the other operand to match.

The priority of operands is as follows:

- long double (highest)
- double
- float
- unsigned long long
- long long
- unsigned long
- long
- unsigned int
- int (lowest)

We can see the usual arithmetic conversion take place via use of the `typeid` operator (included in the <typeinfo> header), which can be used to show the resulting type of an expression.

In the following example, we add two shorts:

```
#include <iostream>
#include <typeinfo> // for typeid()

int main()
{
    short a{ 4 };
    short b{ 5 };
    std::cout << typeid(a + b).name() << " " << a + b << '\n'; // show us the type of a + b

    return 0;
}
```

Because `shorts` are integers, they undergo `integral promotion` to `ints` before being added. The result of adding two `ints` is an `int`, as you would expect:

```
int 9
```

Note: Your compiler may display something slightly different, as the format of typeid.name() is left up to the compiler.

Let's take a look at another case:

```
#include <iostream>
#include <typeinfo> // for typeid()

int main()
{
```

```
    double d{ 4.0 };
    short s{ 2 };
    std::cout << typeid(d + s).name() << ' ' << d + s << '\n'; // show us the type of d + s

    return 0;
}
```

In this case, the `short` undergoes `integral promotion` to an `int`. However, the `int` and `double` still do not match. Since `double` is higher on the hierarchy of types, the integer 2 gets converted to the `double` `2.0`, and the `doubles` are added to produce a `double` result.

```
double 6.0
```

This hierarchy can cause some interesting issues. For example, take a look at the following code:

```
std::cout << 5u - 10; // 5u means treat 5 as an unsigned integer
```

you might expect the expression `5u - 10` to evaluate to -5 since 5 - 10 = -5. But here's what actually happens:

```
4294967291
```

In this case, the `signed integer` `10` is promoted to an `unsigned integer` (which has higher priority), and the expression is evaluated as an `unsigned int`. Since `-5` can't be stored in an `unsigned int`, the calculation wraps around, and we get an answer we don't expect.

This is one of many good reasons to avoid `unsigned integers` in general.

Quiz time

Question #1

What's the difference between a numeric promotion and a numeric conversion?

Show Solution

6.16 -- Explicit type conversion (casting) and static_cast

Index

6.14 -- The auto keyword

C++ TUTORIAL | PRINT THIS POST

109 comments to 6.15 — Implicit type conversion (coercion)

« Older Comments 1 2

kavin
January 8, 2020 at 9:10 am · Reply

In this program why 2 shorts are getting converted to int? They are of same type and hold values within their limits (signed short(2 bytes)-32,768 to 32,767) Could you explain me this please ?

```
int main()
{
   short a{ 4 };
   short b{ 5 };
   std::cout << typeid(a + b).name() << " " << a + b << '\n'; // show us the type of a + b

   return 0;
}
```

OUTPUT: int 9

nascardriver
January 9, 2020 at 3:55 am · Reply

The built-in arithmetic operations, including `+`, don't support any types narrower than an `int`. To still allow narrower types to be added, they get promoted to an `int`.

Nice Joe
October 27, 2019 at 12:05 pm · Reply

Hello Alex,

Please forgive my nitpicking but I'm trying to make sure I'm not confused here.
In the last example you gave, you said one number was promoted to an unsigned integer and then a subtraction was performed. If I recall from an earlier lesson, you stated that signed integers overflow and unsigned integers wraparound.

In this case you are operating on two unsigned integers. So would this be a wraparound, or are you calling it overflow because it is happening in the negative direction? Or something else?

nascardriver
October 28, 2019 at 3:57 am · Reply

Nice, Joe!
It wraps around indeed, lesson updated.

spike
August 13, 2019 at 6:53 pm · Reply

Hi,

In the section "Evaluating arithmetic expressions", you said "If an operand is an integer that is narrower than an int, it undergoes integral promotion (as described above) to int or unsigned int.".

1.
May I ask that what situation the operand promotes to int and what situation it promotes to unsigned int?
It would be better if there is an example, thank you. :)

2.
How about the rule used in bit field arithmetic expression?

nascardriver
August 14, 2019 at 2:55 am · Reply

Hi,

1.
Here's a list
https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_promotion

Peter
June 20, 2019 at 3:05 pm · Reply

I'm pretty sure that converting the int 30000 to a char is actually not undefined. The result is 48, which is 30000 mod 256 (since char is 1 byte). It's rare that a programmer would ever want to do this, but is it technically undefined?

nascardriver
June 21, 2019 at 12:34 am · Reply

Hi Peter!

Until C++20, converting 30000 to a char caused implementation-defined behavior (ie. well defined, but varies between compilers).
Since C++20, the conversion is performed using modulus, ie. 30000 % 2^8, where ^ is the power operator, not xor.

This leaves open what happens if the resulting value is larger than the maximum of the target type. eg. 255 % 2^8 = 255 still doesn't fit into a char.
cppreference says that the number will be truncated, however, I could not find such a rule in the standard.

For anyone who wants to continue the search
https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_conversions
n4820 § 7.3.8

Whatever the outcome may be, it's safe to say that when applying a modulus manually, the code is easier to understand.

Peter
June 21, 2019 at 5:00 pm · Reply

Ok, thanks for clearing that up!

Alireza
February 23, 2019 at 11:04 am · Reply

Hi,
I'm using qt, why does it output following result:

```
1 i 9
```

I know it means that's an int data type, but why doesn't it output 'int' instead of ' i ' ?!! :|
The code is:

```
1     short a(4);
2     short b(5);
3
4     std::cout << typeid (a+b).name() << " " << a + b << '\n';
```

nascardriver
February 24, 2019 at 1:50 am · Reply

* Line 1, 2: Initialize your variables with brace initializers. You used direct initialization.

Everything in @std::type_info (The return type of @typeid) is implementation defined.

Benur21
January 25, 2019 at 3:00 pm · Reply

"In this case, the signed integer (10) is promoted to an unsigned integer (which has higher priority), and the expression is evaluated as an unsigned int. Overflow results, and we get an answer we don’t expect."

Why overflow happens? 10 is 1010 in binary, but both signed and unsigned int have 4 bytes, so the conversion shouldn't cause overflow. Can someone clarify this to me please? Thanks.

nascardriver
January 26, 2019 at 2:04 am · Reply

Hi!

The overflow doesn't occur in the promotion. The overflow occurs when we're calculating
5u - 10
The result of unsigned - signed is unsigned.

Benur21
January 26, 2019 at 2:31 am · Reply

Ahh got it, thanks

Benur21
January 25, 2019 at 1:23 pm · Reply

> the integer value 3 and the float value 3.0 are stored as completely different binary patterns.
3.0 is double by default

I'm in love with alex's tutor
November 14, 2018 at 12:55 am · Reply

Peace for you Mr. Alex!
Can you nominate us a book or website that explains in which header file(library) a built in and pre-compiled functions are found and what they do? like pow(), std::setprecision(),fail() and a like....
B/s knowing them is a best way to shorten our code and to know more about C++ functionality. Can i get them in Visual studio help desk?
God be always with you !!!

nascardriver
November 14, 2018 at 5:53 am · Reply

https://en.cppreference.com/w/cpp
http://www.cplusplus.com/reference/

Use the search function on either one of those sites.
cpluscplus doesn't use https and is slower, but has better examples than cppreference in my opinion.

I'm in love with Alex's tutorial
November 14, 2018 at 6:54 am · Reply

Thank you so much Mr Nas!!!
I always proud of your helpful and prompt responses.
I really feel blessed to have you both!
Thanks much indeed!

Yaroslav
October 9, 2018 at 10:10 am · Reply

Hello.
Here is a text.
"Integral promotion involves the conversion of integer types narrower than int (which includes bool, char, unsigned char, signed char, unsigned short, signed short) to an integer (if possible) or an unsigned int."
what does "to an integer" means? They are all integers. Does it a typo of int as another word for signed int or what? I think it's a bit unclear.

Alex
October 9, 2018 at 1:36 pm · Reply

Replaced "integer" with "int". Thanks for pointing out the confusing wording!

Max Ludero
August 17, 2018 at 8:06 pm · Reply

Hello. Why do you need to #include <typeinfo> for typeid when typeid is a C++ keyword? Thanks!

Alex
August 19, 2018 at 12:50 pm · Reply

Because typeid returns an object of type std::type_info, which is defined in the typeinfo header.

It's a bit of a strange design.

Max Ludero
August 19, 2018 at 8:20 pm · Reply

Thank you very much, Alex.
I was a little confused, because when I originally tried it, it worked without #include <typeinfo>. Then I realized that it was being included with <iostream>. Same with some of the other headers, like <cstdint>. I understand that they should all be #included anyway as a matter of good programming practice. I am using XCode.

Vikram
July 4, 2018 at 9:25 am · Reply

this is not ambiguous, conversion does not work on pointers and addresses, except void and 0(zero)

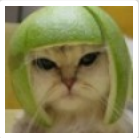
```
1  void func(char* a) { }
2  void func(double *a){}
3  int main()
4  {
5      int a =10;
6      func(&a);
7  }
```

Vikram
July 4, 2018 at 9:21 am · Reply

this is also ambiguous.
int can be demoted to char and promoted to double also.
This int demotion to char is not said in this article.
Please put some lines for demotion from int to char or some other smaller data types

```
1  void func(char a) { }
2  void func(double a){}
3
4  int main()
5  {
6      int a =10;
```

```
7       func(a);
8   }
```



Alex
July 9, 2018 at 1:31 pm · Reply

"When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion**"

The numeric conversions section has an example of an int (typically 32 bits) being converted to a short (typically 16 bits).

Infinite
June 15, 2018 at 9:26 am · Reply

In the last example, why does it overflow instead of becoming 15?

nascardriver
June 15, 2018 at 9:36 am · Reply

Hi Infinite!

Are you talking about this?

```
1   std::cout << 5u - 10;
```

Why would it be 15? It's subtraction, not addition.

Infinite
June 15, 2018 at 10:11 am · Reply

But the - 10 was changed to unsigned, so shouldn't it become 10?

nascardriver
June 15, 2018 at 11:05 am · Reply

That's not how unsigned works. If an unsigned number goes below 0 it will start over at the maximum value.

```
1   unsigned int a{ 5u };
2   unsigned int b{ 10u };
3
4   std::cout << (a - b) << std::endl;
```

This is the same as Alex' code, but it's clearer that 10 is positive and unsigned. The result is unsigned too, but an unsigned int can't hold negative numbers.

Let's do the calculation in binary, because that's what your computer does and that's what causing the result. I'm using 8bit integers, because they're shorter:

```
1     0000 0101  (5)
2   - 0000 1010 (10)
3   ----------------
4   = 1111 1011 (-5)
```

With regular ints, the first bit indicates whether or not the number is negative. 1 means negative. You can convert a negative number to decimal by flipping all bits and adding 1, that will give you the positive number.

```
    1111 1011
flip 0000 0100
+1   0000 0101 (This is a positive 5)
```

Now, the subtraction in unsigned will yield the same 1111 1011, but unsigned integers don't use the first bit as a sign bit, the use it as part of the number.
Using the first bit as part of the number gives us 1111 1011 = 251 in decimal.
32 bits looks the same, just with more leading ones and zeroes.
You don't need to understand how this works on a binary level, but you need to know that when an unsigned number tries going below 0 it will start over at the highest number.

DonnieDarko
June 20, 2018 at 3:39 am · Reply

Binary operator and integer sign is different.
Here: 5u - 10 doesn't mean x = 5u and y = -10;
You can see it as 5u - (10), here 10 is of singed integer and [if it would have been variable its range would have been from (-2147483648 to 2147483647)] and '-' is the binary operator between 5u and 10.
10 will convert to unsigned integer implicitly [getting the range from (0 to 4294967295) if it would have been variable].

hrmn
June 13, 2018 at 3:18 am · Reply

These issue warning? , as we studied earlier short s=2; works fine . should i use it in my code?

```
double d = 3; // convert integer 3 to a double (between different types)
short s = 2; // convert integer 2 to a short (from larger to smaller type)
```

or these issue warning after crossing range.I was thinking to use it in for variables with short range constraints.
By default numbers 1,2,3.. represent int integer?
Thanks.
These tuts are great!

nascardriver
June 13, 2018 at 6:27 am · Reply

Hi hrmn!

Neither g++ nor clang++ issue a conversion warning for your code. You should only get a warning when the values are out-of-range.

> By default numbers 1,2,3.. represent int integer?
Correct, that's why you shouldn't use them to initialize a double. "short" is short for "short int" so it's fine there.

> should i use it in my code?
Use uniform initialization and numbers that match your type.

```
double d{ 3.0 };
short s{ 2 };
```

hrmn
June 13, 2018 at 7:39 am · Reply

""short" is short for "short int" so it's fine there."- i didn't get what it means.
int integer(2 or 4 byte) is converted to short int(2 byte) (might be loss),
in case of double int integer is promoted (and no loss)
whats the difference , why shouldn't we use them to initialize the double.

Thanks for reply.

nascardriver
June 13, 2018 at 8:29 am · Reply

> i didn't get what it means.
Whether you write

```
short s{ 0 };
// or
short int s{ 0 };
```

doesn't matter, "short" is just there so you don't have to write "short int" all the time. A "short" is an "int", so you can initialize it with integers.

> why shouldn't we use them to initialize the double.
When your teacher tells you do bring a checkered piece of paper, do you bring one with lines and draw on the vertical ones? It's the same in the end, but one way is better than the other. Not the best of analogies.
You'll learn about the auto-keyword later, it deduces the type from the value you give it. When you pass an int it will be an int, if you want a double you need to give it a double. If you decide to initialize doubles with int you'll end up with mixed notations.

hrmn
June 13, 2018 at 8:34 am · Reply

default number like 1,2,3 are which type of int ?
short int , int ,long,long long.

sorry i am confused , thinking short integer and int integer separate

nascardriver
June 13, 2018 at 8:38 am · Reply

"short int", "int", "long int" and so on are different, but they're all integers. You can initialize all of them with integer numbers(1, 2, 3, ...) and that's the proper way to initialize them. The only thing you need to care about with integers are their range.

hrmn
June 14, 2018 at 10:17 am · Reply

and the default from "short int", "int", "long int" , is int , i get it now!
Thanks

Cosmin
February 19, 2018 at 10:38 am · Reply

What's happening wrong in this program?

```
int a = numeric_limits<int>::max();
std::cout << typeid(a * 2) << " " << (a * 2) << "\n"; // int -2

unsigned long long b = a * 2;
std::cout << b << "\n"; // Prints correctly
```

I understand why a * 2 evaluates to int, but why does unsigned long long b = a * 2 work correctly?

nascardriver
February 19, 2018 at 10:46 am · Reply

Hi Cosmin!

```
int main()
{
  int a{ std::numeric_limits<int>::max() }; // Missing std::
  std::cout << typeid(a * 2).name() << " " << (a * 2) << "\n"; // Missing .name()

  unsigned long long b{ a * 2 }; // Use uniform initializers
  std::cout << b << "\n";

  return 0;
}
```

The uniform initializer will cause your compiler to to print a warning, because a * 2 is not an unsigned long long.

Frederico Machado
January 31, 2018 at 11:14 am · Reply

I think the first example in numeric conversion is actually doing a promotion.

Alex
February 1, 2018 at 9:37 am · Reply

Nope. Numeric promotion is converting a smaller integral (or floating point) type to a larger integral (or floating point) type.

The first example in numeric conversion is converting an integral number to a floating point value, which is a conversion, not a promotion.

Promotions generally involve extending the binary representation of a number (e.g. for integers, adding leading 0s), whereas conversions require converting the underlying binary representation to a different format.

WiseFool
March 7, 2018 at 9:55 pm · Reply

Hi, Alex
That explains it very succinctly and clearly. I have a little suggestion that maybe you

should make it a little quiz question at the end of this section (4.4) of "Explain the difference between numeric promotion and numeric conversion." with your above explanation as the answer. That way it won't be lost as older comments are cleared. ;)

Alex
March 8, 2018 at 8:37 pm · Reply

Good idea. Done! Thanks for the suggestion.

« Older Comments 1 2