10.5 — Arithmetic conversions

ALEX[1] NOVEMBER 11, 2024

In lesson 6.1 -- Operator precedence and associativity (https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/)[2], we discussed how expressions are evaluated according to the precedence and associativity of their operators.

Consider the following expression:

```
int x { 2 + 3 };
```

Binary `operator+` is given two operands, both of type `int`. Because both operands are of the same type, that type will be used to perform the calculation, and the value returned will also be of this same type. Thus, `2 + 3` will evaluate to `int` value `5`.

But what happens when the operands of a binary operator are of different types?

```
??? y { 2 + 3.5 };
```

In this case, `operator+` is being given one operand of type `int` and another of type `double`. Should the result of the operator be returned as an `int`, a `double`, or possibly something else altogether?

In C++, certain operators require that their operands be of the same type. If one of these operators is invoked with operands of different types, one or both of the operands will be implicitly converted to matching types using a set of rules called the **usual arithmetic conversions**. The matching type produced as a result of the usual arithmetic conversion rules is called the **common type** of the operands.

The operators that require operands of the same type

The following operators require their operands to be of the same type:

- The binary arithmetic operators: +, -, *, /, %
- The binary relational operators: <, >, <=, >=, ==, !=
- The binary bitwise arithmetic operators: &, ^, |
- The conditional operator ?: (excluding the condition, which is expected to be of type `bool`)

For advanced readers

Overloaded operators are not subject to the usual arithmetic conversion rules.

The usual arithmetic conversion rules

The usual arithmetic conversion rules are somewhat complex, so we'll simplify a bit. The compiler has a ranked list of types that looks something like this:

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

The following rules are applied to find a matching type:

Step 1:

- If one operand is an integral type and the other a floating point type, the integral operand is converted to the type of the floating point operand (no integral promotion takes place).
- Otherwise, any integral operands are numerically promoted (see 10.2 -- Floating-point and integral promotion[3]).

Step 2:

- After promotion, if one operand is signed and the other unsigned, special rules apply (see below)
- Otherwise, the operand with lower rank is converted to the type of the operand with higher rank.

For advanced readers

The special matching rules for integral operands with different signs:

- If the rank of the unsigned operand is greater than or equal to the rank of the signed operand, the signed operand is converted to the type of the unsigned operand.
- If the type of the signed operand can represent all the values of the type of the unsigned operand, the type of the unsigned operand is converted to the type of the signed operand.
- Otherwise both operands are converted to the corresponding unsigned type of the signed operand.

Related content

You can find the full rules for the usual arithmetic conversions here (https://en.cppreference.com/w/cpp/language/usual_arithmetic_conversions)[4].

Some examples

In the following examples, we'll use the `typeid` operator (included in the <typeinfo> header), to show the resulting type of an expression.

First, let's add an `int` and a `double`:

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

int main()
{
    int i{ 2 };
    std::cout << typeid(i).name() << '\n'; // show us the name of the type for i

    double d{ 3.5 };
    std::cout << typeid(d).name() << '\n'; // show us the name of the type for d

    std::cout << typeid(i + d).name() << ' ' << i + d << '\n'; // show us the type of i + d

    return 0;
}
```

In this case, the `double` operand has the highest priority, so the lower priority operand (of type `int`) is type converted to `double` value `2.0`. Then `double` values `2.0` and `3.5` are added to produce `double` result `5.5`.

On the author's machine, this prints:

```
int
double
double 5.5
```

Note that your compiler may display something slightly different, as the names output by `typeid.name()` are implementation-specific.

Now let's add two values of type `short`:

```cpp
#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 neither operand appears on the priority list, both operands undergo integral promotion to type `int`. The result of adding two `ints` is an `int`, as you would expect:

```
int 9
```

Signed and unsigned issues

This prioritization hierarchy and conversion rules can cause some problematic issues when mixing signed and unsigned values. For example, take a look at the following code:

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

int main()
{
    std::cout << typeid(5u-10).name() << ' ' << 5u - 10 << '\n'; // 5u means treat 5
as an unsigned integer

    return 0;
}
```

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

```
unsigned int 4294967291
```

Due to the conversion rules, the `int` operand is converted to an `unsigned int`. And since the value `-5` is out of range of an `unsigned int`, we get a result we don't expect.

Here's another example showing a counterintuitive result:

```
#include <iostream>

int main()
{
    std::cout << std::boolalpha << (-3 < 5u) << '\n';

    return 0;
}
```

While it's clear to us that `5` is greater than `-3`, when this expression evaluates, `-3` is converted to a large `unsigned int` that is larger than `5`. Thus, the above prints `false` rather than the expected result of `true`.

This is one of the primary reasons to avoid unsigned integers -- when you mix them with signed integers in arithmetic expressions, you're at risk for unexpected results. And the compiler probably won't even issue a warning.

`std::common_type` and `std::common_type_t`

In future lessons, we'll encounter cases where it is useful to know what the common type of two type is. `std::common_type` and the useful type alias `std::common_type_t` (both defined in the <type_traits> header) can be used for just this purpose.

For example, `std::common_type_t<int, double>` returns the common type of `int` and `double`, and `std::common_type_t<unsigned int, long>` returns the common type of `unsigned int` and `long`.

We'll show an example where this is useful in lesson 11.8 -- Function templates with multiple template types (https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/)[5].

Next lesson
10.6 Explicit type conversion (casting) and static_cast

6

Back to table of contents

7

Previous lesson
10.4 Narrowing conversions, list initialization, and constexpr initializers

8

9

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[12] are connected to your provided email address.

62 COMMENTS

Newest

Cosmo
May 20, 2025 11:06 am PDT

After doing this and the last 5 lessons in this chapter I have got to ask, does the everyday c++ developer ever really remember every detail that is being mentioned here?

Last edited 1 month ago by Cosmo

1 Reply

Danel
Reply to Cosmo[13] June 30, 2025 9:53 pm PDT

Thats why practice is important, you will get accustomed to those details or at least take them into account

0 Reply

Fake Name
Reply to Cosmo 13 May 22, 2025 6:02 pm PDT

Having to learn about all of this math, arithmetic, and bitwise stuff so early into the lessons before even learning functions and classes is pretty unappealing. All the math stuff should be moved towards the end. 10 chapters in and we still haven't even gotten to classes and objects. There should be a warning telling people they can skip these and come back to them later if they just want to get into the syntax and oop faster.

6 Reply

Duck
April 13, 2025 8:37 pm PDT

Hey Mr. Alex
If in the first step of the usual arithemtic conversion, the first and second conditions are being unexecuted, the first step will be omitted?

E.g. when two operands have integral types and there is no possibility to do numeric promotions.

```
int i{ 1 };
long long ll{ 2 };

i + l;
```

If first and second conditions can be unexecuted I think there should be additionally the third condition (or 2 conditions with the third condition which I wrote bellow):

- The first case for **integral and floating point types**;
- The second case for **numer promotions** (after that go to the step 2);
- The third case for unmatched cases (do nothing and go to the step 2)

PS: I really enjoy your tutorial and I am thankful that you created this website with the high level of content. :)

Last edited 2 months ago by Duck

0 Reply

rkh
November 24, 2024 6:16 am PST

1. Are the concepts of "rank" and "size" the same?
2. If not, is it possible for two variables to have different ranks while maintaining the same size?

0 Reply

Alex Author

Reply to rkh [14] November 28, 2024 3:50 pm PST

1. No, as a long long is probably larger than a float.
2. Yes, as int and long are the same size on some architectures.

0 Reply

rkh

Reply to Alex [15] November 30, 2024 3:43 am PST

for this case (2)

unsigned int I{..}
signed long long L{..}

Since L cannot represent all unsigned values of I that have a rank lower than I. Therefore, the third rule can be applied, and the result is obtained as the unsigned type of the signed operand. Is this understanding correct?

0 Reply

Alex Author

Reply to rkh [16] December 2, 2024 5:51 pm PST

Per https://en.cppreference.com/w/cpp/language/types, there is no common architecture where `int` is larger than 32-bits or `long long` is smaller than 64-bits. Therefore, L can represent all values of I.

But let's say there is some esoteric architecture where both are 64-bits. In that case, I believe you are correct.

0 Reply

rkh

Reply to Alex [17] December 3, 2024 7:41 am PST

Thanks for reply

Sorry there was a typo
I meant
signed long L{..}

Is my understanding correct correct in this case?

0 Reply

Alex Author

Reply to rkh [18] December 3, 2024 7:57 pm PST

A signed long may or may not be wider than an unsigned int.

If they are the same width, then the third rule is applied.
If the signed long is wider, then it can represent all the values from the unsigned int, and the second rule is applied.

0 Reply

rkh
Reply to Alex [19] December 4, 2024 1:27 pm PST

Thanks Alex

0 Reply

r.kh
November 10, 2024 7:02 am PST

Is it possible to clarify the last rule as it applies to unsigned and signed integral types with an example?
For instance:

unsigned short a{5};
short b{-10};

Here, It seems the first rule could also apply!

0 Reply

Alex Author
Reply to r.kh [20] November 11, 2024 11:11 am PST

Assuming sizeof(short) < sizeof(int), then both operands are promoted to `int`. Since both types are now the same, the comparison is performed.

If sizeof(short) == sizeof(int), then `a` will be promoted to `unsigned int` and `b` will be promoted to `int`. Since these have the same rank, the first special rule applies, so the `b` is converted to unsigned int.

0 Reply

r.kh
Reply to Alex [21] November 13, 2024 12:39 pm PST

Thanks for the reply
If I'm not mistaken this was not a good case for applying the third rule

Please give me an example

For a 16-bit int
unsigned short a = 5;
signed int b = -2;

Here, can the third rule be applied to these variables?

0 Reply

Manas Ghosh
November 10, 2024 3:17 am PST

what about && and || operators? do they not require both the operands to be boolean type?

1 Reply

Alex Author
Reply to Manas Ghosh [22] November 10, 2024 10:13 pm PST

Those operators require the operands to be bool or convertible to bool. The arithmetic conversions don't apply.

0 Reply

r.kh
October 29, 2024 12:35 pm PDT

`If the rank of the unsigned operand is greater than the rank of the signed operand, the signed operand is converted to the type of the unsigned operand`

I think the term `equal to` should also be added
On the other hand, `-3<5u` confirms this.

0 Reply

Alex Author
Reply to r.kh [23] November 2, 2024 3:21 pm PDT

Corrected. Thanks for pointing this out.

1 Reply

EmtyC
Reply to r.kh [23] October 31, 2024 3:38 am PDT

I think for -3<5u it's the 3rd rule.

0 Reply

DddD
May 22, 2024 5:48 am PDT

It is very glad that compilers are getting smarter and even in the last example **Clang++ (17.0.6 version)** (with the **-Wconversion** flag) and **G++ (13.2.1 version)** (with the **-Wall** flag) show the warning:
G++:

```
warning: comparison of integer expressions of different signedness: 'int' and 'unsigned
```

```
int' [-Wsign-compare]
10 | std::cout << std::boolalpha << (-3 < 5U) << '\n';
| ~~~^~~~
```

Clang++:

```
warning: implicit conversion changes signedness: 'int' to 'unsigned int' [-Wsign-conversion]
10 | std::cout << std::boolalpha << (-3 < 5U) << '\n';
| ^~ ~
```

Always use the many useful compiler flags to detect such errors and I recommend using **Clang++** for more detailed errors!
I use such flags in **G++** and **Clang++,** in case it will be useful for someone:
-Wall -Wextra -Wshadow -Weffc++ -Wno-unused-result -Wlong-long -Wconversion -Wformat=2 -Wcast-align -ansi -pedantic -pedantic-errors -fsanitize=address -fsanitize=undefined -std=c++23

Last edited 1 year ago by DddD

2 Reply

Phargelm
March 30, 2024 8:46 am PDT

> Due to the conversion rules, the int operand is converted to an unsigned int. And since the value -5 is out of range of an unsigned int, we get a result we don't expect.

Is it because of this rule?
> Otherwise both operands are converted to the corresponding unsigned type of the signed operand.

0 Reply

Alex Author
Reply to Phargelm 24 March 30, 2024 1:25 pm PDT

Yep.

0 Reply

Vitalik
January 22, 2024 3:14 am PST

Short + short = int.
Why use the short type if it is implicitly converted to int?

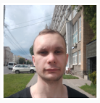
We haven't done anything to make implicit type conversion happen.

0 Reply

Alex Author
Reply to Vitalik 25 January 23, 2024 10:08 am PST

Because storing a short may take less memory. The fact that it gets promoted to int when used in certain cases is a transitory effect.

1 Reply



Vitalik

Reply to Alex 26 January 24, 2024 12:20 am PST

Got it, thanks for the reply.
It is clear that short takes less than int or long, but I've heard people say something like "it's just saving a couple bytes and it won't make a difference", is this true with current device capacities?

Or for example when creating health points for a character and knowing that the maximum there will be 150 HP should I create short?

0 Reply

Alex Author

Reply to Vitalik 27 January 25, 2024 3:16 pm PST

Saving a couple of bytes doesn't matter when allocating individual objects (which is 99% of the cases we encounter). It can matter when we're allocating an array of a million objects.

> Or for example when creating health points for a character and knowing that the maximum there will be 150 HP should I create short?

I wouldn't. Unless there's some reason you need your character to be as small as possible (e.g. because you're going to allocate a ton of them), I'd prefer `int` and avoid implicit conversion headaches.

3 Reply

Vigil

October 22, 2023 2:30 am PDT

Even -3 is converted to unsign int,it is 3,still smaller than 5,what you mean I guess it's better to take -6 or something else smaller than -5

0 Reply

Alex Author

Reply to Vigil 28 October 23, 2023 8:42 pm PDT

-3 converted to an unsigned int is not 3 -- it's (2^n)-3, where n is the number of bits in the unsigned number.

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/
3. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/
4. https://en.cppreference.com/w/cpp/language/usual_arithmetic_conversions
5. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/
6. https://www.learncpp.com/cpp-tutorial/explicit-type-conversion-casting-and-static-cast/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/
9. https://www.learncpp.com/arithmetic-conversions/
10. https://www.learncpp.com/cpp-tutorial/numeric-conversions/
11. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/
12. https://gravatar.com/
13. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-610239
14. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604463
15. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604622
16. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604702
17. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604788
18. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604820
19. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604838
20. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604014
21. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604080
22. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-604007
23. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-603654
24. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-595257
25. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-592676
26. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-592740
27. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-592791
28. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/#comment-589018
29. https://g.ezoic.net/privacy/learncpp.com