# 10.2 — Floating-point and integral promotion

ALEX[1] JULY 31, 2024

In lesson 4.3 -- Object sizes and the sizeof operator (https://www.learncpp.com/cpp-tutorial/object-sizes-and-the-sizeof-operator/)[2], we noted that C++ has minimum size guarantees for each of the fundamental types. However, the actual size of these types can vary based on the compiler and architecture.

This variability was allowed so that the `int` and `double` data types could be set to the size that maximizes performance on a given architecture. For example, a 32-bit computer will typically be able to process 32-bits of data at a time. In such cases, an `int` would likely be set to a width of 32-bits, since this is the "natural" size of the data that the CPU operates on (and likely to be the most performant).

### A reminder

The number of bits a data type uses is called its width. A wider data type is one that uses more bits, and a narrower data type is one that uses less bits.

But what happens when we want our 32-bit CPU to modify an 8-bit value (such as a `char`) or a 16-bit value? Some 32-bit processors (such as 32-bit x86 CPUs) can manipulate 8-bit or 16-bit values directly. However, doing so is often slower than manipulating 32-bit values! Other 32-bit CPUs (like 32-bit PowerPC CPUs), can only operate on 32-bit values, and additional tricks must be employed to manipulate narrower values.

---

## Numeric promotion

Because C++ is designed to be portable and performant across a wide range of architectures, the language designers did not want to assume a given CPU would be able to efficiently manipulate values that were narrower than the natural data size for that CPU.

To help address this challenge, C++ defines a category of type conversions informally called the `numeric promotions`. A **numeric promotion** is the type conversion of certain narrower numeric types (such as a `char`) to certain wider numeric types (typically `int` or `double`) that can be processed efficiently.

All numeric promotions are value-preserving. A **value-preserving conversion** (also called a **safe conversion**) is one where every possible source value can be converted into an equal value of the destination type.

Because promotions are safe, the compiler will freely use numeric promotion as needed, and will not issue a warning when doing so.

---

## Numeric promotion reduces redundancy

Numeric promotion solves another problem as well. Consider the case where you wanted to write a function to print a value of type `int`:

```
#include <iostream>

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

While this is straightforward, what happens if we want to also be able to print a value of type `short`, or type `char`? If type conversions did not exist, we'd have to write a different print function for `short` and another one for `char`. And don't forget another version for `unsigned char`, `signed char`, `unsigned short`, `wchar_t`, `char8_t`, `char16_t`, and `char32_t`! You can see how this quickly becomes unmanageable.

Numeric promotion comes to the rescue here: we can write functions that have `int` and/or `double` parameters (such as the `printInt()` function above). That same code can then be called with arguments of types that can be numerically promoted to match the types of the function parameters.

---

## Numeric promotion categories

The numeric promotion rules are divided into two subcategories: `integral promotions` and `floating point promotions`. Only the conversions listed in these categories are considered to be numeric promotions.

---

## Floating point promotions

We'll start with the easier one.

Using the **floating point promotion** rules, a value of type `float` can be converted to a value of type `double`.

This means we can write a function that takes a `double` and then call it with either a `double` or a `float` value:

```
#include <iostream>

void printDouble(double d)
{
    std::cout << d << '\n';
}

int main()
{
    printDouble(5.0); // no conversion necessary
    printDouble(4.0f); // numeric promotion of float to double

    return 0;
}
```

In the second call to `printDouble()`, the `float` literal `4.0f` is promoted into a `double`, so that the type of argument matches the type of the function parameter.

---

## Integral promotions

The integral promotion rules are more complicated.

Using the **integral promotion** rules, the following conversions can be made:

- signed char or signed short can be converted to int.
- unsigned char, char8_t, and unsigned short can be converted to int if int can hold the entire range of the type, or unsigned int otherwise.
- If char is signed by default, it follows the signed char conversion rules above. If it is unsigned by default, it follows the unsigned char conversion rules above.
- bool can be converted to int, with false becoming 0 and true becoming 1.

Assuming an 8 bit byte and an `int` size of 4 bytes or larger (which is typical these days), the above basically means that `bool`, `char`, `signed char`, `unsigned char`, `signed short`, and `unsigned short` all get promoted to `int`.

There are a few other integral promotion rules that are used less often. These can be found at https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_promotion (https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_promotion)[3].

In most cases, this lets us write a function taking an `int` parameter, and then use it with a wide variety of other integral types. For example:

```
#include <iostream>

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

int main()
{
    printInt(2);

    short s{ 3 }; // there is no short literal suffix, so we'll use a variable for this one
    printInt(s); // numeric promotion of short to int

    printInt('a'); // numeric promotion of char to int
    printInt(true); // numeric promotion of bool to int

    return 0;
}
```

There are two things worth noting here. First, on some architectures (e.g. with 2 byte ints) it is possible for some of the unsigned integral types to be promoted to `unsigned int` rather than `int`.

Second, some narrower unsigned types (such as `unsigned char`) may be promoted to larger signed types (such as `int`). So while integral promotion is value-preserving, it does not necessarily preserve the signedness (signed/unsigned) of the type.

---

## Not all widening conversions are numeric promotions

Some widening type conversions (such as `char` to `short`, or `int` to `long`) are not considered to be numeric promotions in C++ (they are `numeric conversions`, which we'll cover shortly in lesson 10.3 -- Numeric conversions (https://www.learncpp.com/cpp-tutorial/numeric-conversions/)[4]). This is because such conversions do not assist in the goal of converting smaller types to larger types that can be processed more efficiently.

The distinction is mostly academic. However, in certain cases, the compiler will favor numeric promotions over numeric conversions. We'll see examples where this makes a difference when we cover function

overload resolution (in upcoming lesson 11.3 -- Function overload resolution and ambiguous matches (https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/)[5]).


**Next lesson**
10.3 Numeric conversions

4

**Back to table of contents**

6

**Previous lesson**
10.1 Implicit type conversion

7

8




77 COMMENTS Newest

NordicPeace

December 11, 2024 10:54 pm PST

his variability was allowed so that the int and double data types could be set to the size that maximizes performance on a given architecture. For example, a 32-bit computer will typically be able to process 32-bits of data at a time. In such cases, an int would likely be set to a width of 32-bits, since this is the “natural” size of the data that the CPU operates on (and likely to be the most performant).

Chat- gpt :

Imagine a bakery that bakes loaves of bread. This bakery has a bread oven with 32 loaf-sized trays that can bake up to 32 loaves at a time. If the bakery decides to bake only 8 loaves at once (using just a quarter of the oven), the oven is underutilized, and it takes just as much energy and time to heat the oven for fewer loaves.

Similarly, a 32-bit computer is like this bakery oven designed to process 32 bits of data at a time. If you ask it to process only 8 bits (smaller data), it's not using its full capacity. The computer still takes the same amount of effort to handle the smaller piece, and thus it's less efficient.

Is this correct?

2 Reply

**Alex** Author

Reply to NordicPeace [11] December 16, 2024 1:39 pm PST

I like that analogy a lot for basic understanding. But it's perhaps incomplete, as it leaves the reader with the impression that the issue is purely related to (under)utilization.

On some architectures, performing operations on 8-bit or 16-bit data is actually SLOWER than performing operations on 32-bit data. This can be because the operations are actually performed as 32-bit operations, and then the compiler needs to perform an additional operation to mask/transform the data back to the smaller number of bits. Or also because some types of instructions can process an multiple pieces of 32-bit data in parallel, but not multiple pieces of 16-bit or 8-bit data.

14 Reply

**NordicPeace**

Reply to Alex [12] December 16, 2024 11:12 pm PST

Thanks for clarification :)

0 Reply

**Jan Schultke**

May 13, 2024 2:48 am PDT

`char8_t` and `bool` are not subject to integral promotions. See https://eel.is/c++draft/conv.prom#2

They can be converted to `int`, but as part of integral conversions (https://eel.is/c++draft/conv.integral), not as part of promotions. This does matter for overload resolution, C variadics, and other places.

0 Reply

**Alex** Author

Reply to Jan Schultke [13] May 13, 2024 12:21 pm PDT

This does not appear to be true.

https://eel.is/c++draft/conv.prom#6 covers charX_t promotions and https://eel.is/c++draft/conv.prom#7 covers bool promotions.

You can also validate this experimentally by overloading `foo(int)` and `foo(long)`:

```
#include <iostream>

void foo(int)
{
    std::cout << "int\n";
}

void foo(long)
{
    std::cout << "long\n";
}

int main()
{
    // If these are promotions, foo(int) should be favored
    // If these are conversions, should be ambiguous
    foo(true);
    foo(char8_t { 'a' });
}
```

This compiles.

2 Reply

**Jan Schultke**

Reply to Alex [14] May 13, 2024 12:28 pm PDT

Ah yes, sorry. I have missed the latter paragraphs in [conv.prom].

1 Reply

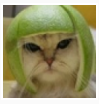
**Swaminathan R**

May 2, 2024 3:36 am PDT

“A numeric promotion is the type conversion of certain narrower numeric types (such as a char) to certain wider numeric types (typically int or double) that can be processed efficiently and is *less likely* to have a result that overflows“

Less likely:: Since it is converting to a wider length, shouldn’t it always be able to accommodate the source value?

0 Reply



**Alex** Author

Reply to Swaminathan R [15] May 3, 2024 1:46 pm PDT

When I say "a result", I don't mean a result of the promotion, I mean a result of the operation being applied.

For example, consider a signed char that has range -128 to 127. If we add two chars, without promotion the result could easily overflow the range of char. However, if we promote both chars to int

and return an int, we won't encounter overflow.

That said, I've removed that part of the sentence, as that's really more specific to the usual arithmetic conversions rather than the promotion process itself.

0 Reply

**Swaminathan R**

Reply to Alex [16] May 3, 2024 11:41 pm PDT

Thank you ❤️

0 Reply

**Life failure**

September 26, 2023 4:24 am PDT

```
#include <iostream>
#include <typeinfo>
#include <string>

void printInt(int x, std::string a);

int main()
{
printInt(2, typeid(2).name() );

short s{ 3 }; // there is no short literal suffix, so we'll use a variable for this one
printInt(s, typeid(s).name()); // numeric promotion of short to int

printInt('a', typeid('a').name()); // numeric promotion of char to int
printInt(true, typeid(true).name()); // numeric promotion of bool to int

return 0;
}
void printInt(int x, std::string a)
{
std::cout << x << ":\t" << a << '\n';
}
```

sorry for my bad eng teacher ^.^

for short : Numeric promotion categories
are used in fonction calls so we don't have to type a specified fonction for every type, anything else is considered
conversion right?

*Last edited 1 year ago by Life failure*

0 Reply

**Life Failure**

Reply to Life failure [17] September 26, 2023 1:36 pm PDT

after reading next lessons now i understand promotions. better sorry for bothering you master

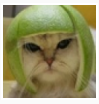
5 Reply

### Ali

August 11, 2023 11:22 am PDT

numeric promotion is about performance and numeric conversion is about memory ?
if not, what distinguishes promotion from conversion ?

0 Reply



### Alex Author

Reply to Ali [18] August 12, 2023 4:50 pm PDT

Numeric promotions are just a special set of conversions that are guaranteed to be safe and take precedence over other kinds of conversions when matching overloaded functions.

2 Reply

### j t

July 5, 2023 3:56 am PDT

Hi, I'm a little unclear on this part of the section on `Numeric promotion`:

> A numeric promotion is the type conversion of certain narrower numeric types (such as a char) to certain wider numeric types (typically int or double) that can be processed efficiently and **is less likely to have a result that overflows.**

I'm not sure I understand how numeric promotion reduces the likelihood of overflow. Say we're adding two `char`s, then even if the `char`s are processed as `int`s, their result will still be stored in a `char`-sized object, right? So, doesn't overflow occur regardless of whether the `char`s are promoted?

0 Reply

### Alex Author

Reply to j t [19] July 7, 2023 1:53 pm PDT

> Say we're adding two chars, then even if the chars are processed as ints, their result will still be stored in a char-sized object, right?

No, the result is returned as an int.

0 Reply

### Emeka Daniel

Reply to Alex [20] August 13, 2023 7:33 am PDT

Okayyy, that's why performing bitwise negation on an unsigned char type returns a signed integer, because I downloaded a 32bit compiler(mingw32) that assumes that 32bit is the fastest data size for my computer[correct if wrong].

My computer's processor is actually 64bit though(and x86 compatible), so if I switched to a 64bit compiler like in visual studio would it return a signed long long?

0 Reply

**Alex** Author

Reply to Emeka Daniel [21] August 17, 2023 4:31 pm PDT

No, integral promotion is always to `int` (or occasionally `unsigned int`). Your ints may be 32 or 64 bits depending on architecture.

0 Reply

**Emeka Daniel**

Reply to Alex [22] August 18, 2023 2:12 am PDT

Okay.
Since we are in the talks about architecture natural data sizes, I have a question.

Some months ago, I compiled and ran a third party library application on my X64 bits MSVC compiler and though my computer is X64 bits, it didn't not function correctly, it would start and end abruptly. But when I compiled on my X86 bits MSVC compiler, it ran without issue. Why is it so?

0 Reply

**Alex** Author

Reply to Emeka Daniel [23] August 23, 2023 4:18 pm PDT

When you say "library", what format was it in? AFAIK, 64-bit applications can't load 32-bit DLLs (or vice-versa).

0 Reply

**Emeka Daniel**

Reply to Alex [24] August 24, 2023 2:41 am PDT

`When you say "library", what format was it in?`
It was distributed in a Visual Studio Solution folder, just the code was made available for users to compile.

`AFAIK, 64-bit applications can't load 32-bit DLLs (or vice-versa).`
Come to think of it, it had nothing to do with the processor's data size, it was this my NORTON ANTIVIRUS, it kept blocking its X64 bit dll for reasons unbeknownst to me then. Once I ran it the Antivirus would start acting up saying something malicious[the X64 bit dll] is running and would then forcefully shutdown the application, when I then went to

rerun the software, It wouldn't be there because Norton have already quarantined the "problem". It's all so clear now, because of this debugger problem you helped me solve. The Norton Antivirus even cleaned out my entire recently downloaded compiler /bin folder the previous day because of the same reason(well maybe not the same, but it was generic).

I would sue that Antivirus for grievances caused but unfortunately it was my ignorance that ultimately let the software have free rein. Thanks again for the debugger help that time, it meant alot.

0 Reply

**Krishnakumar**
June 28, 2023 10:58 am PDT

>the compiler will freely use numeric promotion as needed, and will not issue a warning when doing so.

However, a static analyser like `clang-tidy` will e.g with a message like "implicit conversion increases floating-point precision: 'float' to 'double'"

1 Reply

**LoudWard**
June 20, 2023 12:08 pm PDT

as far as i understand, promotion is for different size and conversion same size?

0 Reply

**Alex** Author
Reply to LoudWard [25] June 23, 2023 8:51 pm PDT

Promotions are usually smaller to larger converisons between specific types. But short to int is always a promotion, regardless of whether short and int are the same size or not.

A conversion is everything that isn't a promotion. It can be smaller to larger, larger to smaller, or same size.

4 Reply

**brandnewdrako**
April 20, 2023 11:55 pm PDT

short yy = 32769; // here 32769 is an int value so when i complile this the result comes out to be -32767 why does the sign changed ?

1 Reply

**Alex** Author

Reply to brandnewdrako [26] April 23, 2023 3:58 pm PDT

Because the range of a 16-bit signed variable is -32768 to 32767. Since 32769 is outside this range, data is lost and the result you see is whatever remains.

1 Reply

**Bharath**

April 11, 2023 8:27 pm PDT

```
1  #include <iostream>
2
3  void printInt(int x)
4  {
5      std::cout << x << '\n';
6  }
7
8  int main()
9  {
10     printInt(2);
11
12     short s{ 3 }; // there is no short literal suffix, so we'll use a variable for
13 this one
14     printInt(s); // numeric promotion of short to int
15
16     printInt('a'); // numeric promotion of char to int
17     printInt(true); // numeric promotion of bool to int
18
19     return 0;
   }
```

the line 10, Is it comes under integral promotion of int to int ?

0 Reply

**Alex** Author

Reply to Bharath [27] April 13, 2023 11:19 pm PDT

No. 2 is an int literal, and printInt() is expecting an int, so no conversion or promotion is necessary.

1 Reply

**Bharath**

Reply to Alex [28] April 15, 2023 8:16 pm PDT

Ohh thanks alex

0 Reply

# Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/object-sizes-and-the-sizeof-operator/
3. https://en.cppreference.com/w/cpp/language/implicit_conversion#Integral_promotion
4. https://www.learncpp.com/cpp-tutorial/numeric-conversions/
5. https://www.learncpp.com/cpp-tutorial/function-overload-resolution-and-ambiguous-matches/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/
8. https://www.learncpp.com/floating-point-and-integral-promotion/
9. https://www.learncpp.com/cpp-tutorial/common-semantic-errors-in-c/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-605161
12. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-605321
13. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-597046
14. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-597060
15. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-596556
16. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-596616
17. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-587762
18. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-585400
19. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-583238
20. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-583368
21. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-585545
22. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-585707
23. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-585760
24. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-585982
25. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-582209
26. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-579567
27. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-579221
28. https://www.learncpp.com/cpp-tutorial/floating-point-and-integral-promotion/#comment-579285
29. https://g.ezoic.net/privacy/learncpp.com