10.4 — Narrowing conversions, list initialization, and constexpr initializers

ALEX[1] FEBRUARY 16, 2024

In the previous lesson (10.3 -- Numeric conversions (https://www.learncpp.com/cpp-tutorial/numeric-conversions/)[2]), we covered numeric conversions, which cover a wide range of different type conversions between fundamental types.

Narrowing conversions

In C++, a **narrowing conversion** is a potentially unsafe numeric conversion where the destination type may not be able to hold all the values of the source type.

The following conversions are defined to be narrowing:

- From a floating point type to an integral type.
- From a floating point type to a narrower or lesser ranked floating point type, unless the value being converted is constexpr and is in range of the destination type (even if the destination type doesn't have the precision to store all the significant digits of the number).
- From an integral to a floating point type, unless the value being converted is constexpr and whose value can be stored exactly in the destination type.
- From an integral type to another integral type that cannot represent all values of the original type, unless the value being converted is constexpr and whose value can be stored exactly in the destination type. This covers both wider to narrower integral conversions, as well as integral sign conversions (signed to unsigned, or vice-versa).

In most cases, implicit narrowing conversions will result in compiler warnings, with the exception of signed/unsigned conversions (which may or may not produce warnings, depending on how your compiler is configured).

Narrowing conversions should be avoided as much as possible, because they are potentially unsafe, and thus a source of potential errors.

Best practice

Because they can be unsafe and are a source of errors, avoid narrowing conversions whenever possible.

Make intentional narrowing conversions explicit

Narrowing conversions are not always avoidable -- this is particularly true for function calls, where the function parameter and argument may have mismatched types and require a narrowing conversion.

In such cases, it is a good idea to convert an implicit narrowing conversion into an explicit narrowing conversion using `static_cast`. Doing so helps document that the narrowing conversion is intentional, and will suppress any compiler warnings or errors that would otherwise result.

For example:

```cpp
void someFcn(int i)
{
}

int main()
{
    double d{ 5.0 };

    someFcn(d); // bad: implicit narrowing conversion will generate compiler warning

    // good: we're explicitly telling the compiler this narrowing conversion is
intentional
    someFcn(static_cast<int>(d)); // no warning generated

    return 0;
}
```

Best practice

If you need to perform a narrowing conversion, use `static_cast` to convert it into an explicit conversion.

Brace initialization disallows narrowing conversions

Narrowing conversions are disallowed when using brace initialization (which is one of the primary reasons this initialization form is preferred), and attempting to do so will produce a compile error.

For example:

```cpp
int main()
{
    int i { 3.5 }; // won't compile

    return 0;
}
```

Visual Studio produces the following error:

```
error C2397: conversion from 'double' to 'int' requires a narrowing conversion
```

If you actually want to do a narrowing conversion inside a brace initialization, use `static_cast` to convert the narrowing conversion into an explicit conversion:

```cpp
int main()
{
    double d { 3.5 };

    // static_cast<int> converts double to int, initializes i with int result
    int i { static_cast<int>(d) };

    return 0;
}
```

Some constexpr conversions aren’t considered narrowing

When the source value of a narrowing conversion isn’t known until runtime, the result of the conversion also can’t be determined until runtime. In such cases, whether the narrowing conversion preserves the value or not also can’t be determined until runtime. For example:

```cpp
#include <iostream>

void print(unsigned int u) // note: unsigned
{
    std::cout << u << '\n';
}

int main()
{
    std::cout << "Enter an integral value: ";
    int n{};
    std::cin >> n; // enter 5 or -5
    print(n);      // conversion to unsigned may or may not preserve value

    return 0;
}
```

In the above program, the compiler has no idea what value will be entered for `n`. When `print(n)` is called, the conversion from `int` to `unsigned int` will be performed at that time, and the results may be value-preserving or not depending on what value for `n` was entered. Thus, a compiler that has signed/unsigned warnings enabled will issue a warning for this case.

However, you may have noticed that most of the narrowing conversions definitions have an exception clause that begins with “unless the value being converted is constexpr and …”. For example, a conversion is narrowing when it is “From an integral type to another integral type that cannot represent all values of the original type, unless the value being converted is constexpr and whose value can be stored exactly in the destination type.”

When the source value of a narrowing conversion is constexpr, the specific value to be converted must be known to the compiler. In such cases, the compiler can perform the conversion itself, and then check whether the value was preserved. If the value was not preserved, the compiler can halt compilation with an error. If the value is preserved, the conversion is not considered to be narrowing (and the compiler can replace the entire conversion with the converted result, knowing that doing so is safe).

For example:

```cpp
#include <iostream>

int main()
{
    constexpr int n1{ 5 };    // note: constexpr
    unsigned int u1 { n1 };  // okay: conversion is not narrowing due to exclusion
clause

    constexpr int n2 { -5 }; // note: constexpr
    unsigned int u2 { n2 };  // compile error: conversion is narrowing due to value
change

    return 0;
}
```

Let's apply the rule "From an integral type to another integral type that cannot represent all values of the original type, unless the value being converted is constexpr and whose value can be stored exactly in the destination type" to both of the conversions above.

In the case of `n1` and `u1`, `n1` is an `int` and `u1` is an `unsigned int`, so this is a conversion from an integral type to another integral type that cannot represent all values of the original type. However, `n1` is constexpr, and its value `5` can be represented exactly in the destination type (as unsigned value `5`). Therefore, this is not considered to be a narrowing conversion, and we are allowed to list initialize `u1` using `n1`.

In the case of `n2` and `u2`, things are similar. Although `n2` is constexpr, its value `-5` cannot be represented exactly in the destination type, so this is considered to be a narrowing conversion, and because we are doing list initialization, the compiler will error and halt the compilation.

Strangely, conversions from a floating point type to an integral type do not have a constexpr exclusion clause, so these are always considered narrowing conversions even when the value to be converted is constexpr and fits in the range of the destination type:

```cpp
int n { 5.0 }; // compile error: narrowing conversion
```

Even more strangely, conversions from a constexpr floating point type to a narrower floating point type are not considered narrowing even when there is a loss of precision!

```cpp
constexpr double d { 0.1 };
float f { d }; // not narrowing, even though loss of precision results
```

Warning

Conversion from a constexpr floating point type to a narrower floating point type is not considered narrowing even when a loss of precision results.

List initialization with constexpr initializers

These constexpr exception clauses are incredibly useful when list initializing non-int/non-double objects, as we can use an int or double literal (or a constexpr object) initialization value.

This allows us to avoid:

- Having to use literal suffixes in most cases
- Having to clutter our initializations with a static_cast

For example:

```cpp
int main()
{
    // We can avoid literals with suffixes
    unsigned int u { 5 }; // okay (we don't need to use `5u`)
    float f { 1.5 };      // okay (we don't need to use `1.5f`)

    // We can avoid static_casts
    constexpr int n{ 5 };
    double d { n };       // okay (we don't need a static_cast here)
    short s { 5 };        // okay (there is no suffix for short, we don't need a
static_cast here)

    return 0;
}
```

This also works with copy and direct initialization.

One caveat worth mentioning: initializing a narrower or lesser ranked floating point type with a constexpr value is allowed as long as the value is in range of the destination type, even if the destination type doesn't have enough precision to precisely store the value!

Key insight

Floating point types are ranked in this order (greater to lesser):

- Long double
- Double
- Float

Therefore, something like this is legal and will not emit an error:

```cpp
int main()
{
    float f { 1.23456789 }; // not a narrowing conversion, even though precision lost!

    return 0;
}
```

However, your compiler may still issue a warning in this case (GCC and Clang do if you use the -Wconversion compile flag).

Next lesson
10.5 Arithmetic conversions

3

Back to table of contents

4

Previous lesson
10.3 Numeric conversions

2

5

44 COMMENTS Newest

alper
July 3, 2025 1:27 pm PDT

this chapter is soo boring

0 Reply

Yavuz
April 15, 2025 3:21 am PDT

Hey Alex! I get compiler warning about conversion in this example above. isn't static_cast should avoid warnings?

```cpp
void someFcn(int i)
{
}

int main()
{
    double d{ 5.0 };

    someFcn(d); // bad: implicit narrowing conversion will generate compiler warning

    // good: we're explicitly telling the compiler this narrowing conversion is
intentional
    someFcn(static_cast<int>(d)); // no warning generated

    return 0;
}
```

0 Reply

Pvt. Joker
February 28, 2025 9:00 pm PST

yo alex, i noticed that you mentioned converting an integral to a floating point type. isnt this actually a widening conversion?

1 Reply

Pvt. Joker
February 27, 2025 6:18 pm PST

everything was good and dandy until i took a glance at this line XD

"Even more strangely, conversions from a constexpr floating point type to a narrower floating point type are not considered narrowing even when there is a loss of precision!"

3 Reply

> **Jamison**
> Reply to Pvt. Joker 9 June 12, 2025 9:14 pm PDT
>
> IKR this feels like an interesting design choice for the language and something to remember. -Wconversion catches this but I wonder what is the historical reason for this being in the standard.
>
> 0 Reply

man98
February 1, 2025 6:14 am PST

Just sharing how I ran into a narrowing conversion error when creating the HiLo game in the previous lessons. `std::tolower()` returns an integer which was generating an error due to the implicit narrowing conversion in the brace initialization. I fixed that by wrapping `std::tolower(getChar())` in `char()` but maybe I should have used static_cast?

```
char getChar()
{
    // Gets char from cin...
}

bool playAgain() {

    std::cout << "Would you like to play again (y/n)? ";
    char playAgain{ std::tolower(getChar()) };
    // More code...

    return false;
}

int main()
{
    do {
        playHilo();
    } while (playAgain());
}
```

EDIT: I see this addressed in an upcoming lesson: "Avoid using C-style casts."

Last edited 5 months ago by man98

0 Reply

Tobito
January 20, 2025 8:02 pm PST

constexpr double d { 0.1 };
float f { d }; // not narrowing, even though loss of precision results

A loss precision is not considered by narrowing conversion? I think because of the narrowing conversion that loss precision is occurred

0 Reply

Mike T.
December 27, 2024 11:14 am PST

"Thus, a compiler that has signed/unsigned warnings enabled will issue a warning for this case."

```
void print(unsigned int u) // note: unsigned
{
    std::cout << u << '\n';
}

int main()
{
    std::cout << "Enter an integral value: ";
    int n{};
    std::cin >> n; // enter 5 or -5
    print(n);      // conversion to unsigned may or may not preserve value
}
```

I do not get any warnings, and I have all my settings in Visual Studio configured as you stated in Lesson 0.11, including running the test code to confirm.

Why no warnings for this particular code?

0 Reply

Alex Author
Reply to Mike T. [10] January 3, 2025 5:16 pm PST

I get a warning for that code on VS:(13): warning C4365: 'argument': conversion from 'int' to 'unsigned int', signed/unsigned mismatch

I suspect some setting still isn't set right (maybe you configured only the release build instead of release and debug?)

0 Reply

r.kh
October 19, 2024 3:34 pm PDT

According to exception clauses for narrowing conversion, isn't a variable considered a narrowing conversion if initialized with a value not marked as constexpr, even if it's exactly stored in the destination type?

0 Reply

Alex Author
Reply to r.kh [11] October 20, 2024 4:02 pm PDT

I'm not sure I follow what you're asking.

0 Reply

rkh
October 13, 2024 8:25 am PDT

```
initializing a narrower or lesser ranked floating point type with a constexpr value is
allowed as long as the value is in range of the destination type, even if the
```

`destination type doesn’t have enough precision to precisely store the value!`

can we apply this context for integral type?

0 Reply

Alex Author

Reply to rkh [12] October 17, 2024 5:59 pm PDT

Yes, minus the part about precision since integral values are always exact.

0 Reply

rkh

October 12, 2024 8:58 am PDT

hello Alex

#include <iostream>

int main()
{

constexpr int n2 { -5 }; // note: constexpr
unsigned int u2 = n2 ;

return 0;
}

in this snippet, a compilation error does not occur, so is the exception clause appropriate for a brace initialization case?

0 Reply

Alex Author

Reply to rkh [13] October 16, 2024 11:21 am PDT

I get a compilation error on both GCC and Clang. What compiler and language standard are you using?

0 Reply

r.kh

Reply to Alex [14] October 18, 2024 4:49 am PDT

I didn't get any compilation error and warning
C++ language on Linux environment with GCC compiler is used
I compiled and got executable file as follows:

```
g++ -o exam.exe example.cpp
```

0 Reply

Alex Author

Reply to r.kh [15] October 20, 2024 3:21 pm PDT

Oh I see, it's because you used copy initialization, which doesn't prevent narrowing conversions.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/numeric-conversions/
3. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/
4. https://www.learncpp.com/
5. https://www.learncpp.com/narrowing-conversions-list-initialization-and-constexpr-initializers/
6. https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/
7. https://www.learncpp.com/cpp-tutorial/constexpr-if-statements/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-608131
10. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-605786
11. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-603353
12. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-603077
13. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-603001
14. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-603195
15. https://www.learncpp.com/cpp-tutorial/narrowing-conversions-list-initialization-and-constexpr-initializers/#comment-603287
16. https://g.ezoic.net/privacy/learncpp.com