

10.x — Chapter 10 summary and quiz

👤 [ALEX](#)¹ ⌚ **SEPTEMBER 3, 2024**

Great job making it this far. The standard conversion rules are pretty complex -- don't worry if you don't understand every nuance.

Chapter Review

The process of converting a value from one data type to another data type is called a **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.

The C++ language defines a number of built-in conversions between its fundamental types (as well as a few conversions for more advanced types) called **standard conversions**. These include numeric promotions, numeric conversions, and arithmetic conversions.

A **numeric promotion** is the conversion of certain smaller numeric types to certain larger numeric types (typically `int` or `double`), so that the CPU can operate on data that matches the natural data size for the processor. Numeric promotions include both integral promotions and floating-point promotions. Numeric promotions are **value-preserving**, meaning there is no loss of value or precision. Not all widening conversions are promotions.

A **numeric conversion** is a type conversion between fundamental types that isn't a numeric promotion. A **narrowing conversion** is a numeric conversion that may result in the loss of value or precision.

In C++, certain binary operators require that their operands be of the same type. If operands of different types are provided, one or both of the operands will be implicitly converted to matching types using a set of rules called the **usual arithmetic conversions**.

Explicit type conversion is performed when the programmer explicitly requests conversion via a cast. A **cast** represents a request by the programmer to do an explicit type conversion. C++ supports 5 types of casts: `C-style casts`, `static casts`, `const casts`, `dynamic casts`, and `reinterpret casts`. Generally you should avoid `C-style casts`, `const casts`, and `reinterpret casts`. `static_cast` is used to convert a value from one type to a value of another type, and is by far the most used cast in C++.

Typedefs and **type aliases** allow the programmer to create an alias for a data type. These aliases are not new types, and act identically to the aliased type. Typedefs and type aliases do not provide any kind of type safety, and care needs to be taken to not assume the alias is different than the type it is aliasing.

The **auto** keyword has a number of uses. First, auto can be used to do **type deduction** (also called **type inference**), which will deduce a variable's type from its initializer. Type deduction drops const and references, so be sure to add those back if you want them.

Auto can also be used as a function return type to have the compiler infer the function's return type from the function's return statements, though this should be avoided for normal functions. Auto is used as part of the **trailing return syntax**.

Quiz time

Question #1

What type of conversion happens in each of the following cases? Valid answers are: No conversion needed, numeric promotion, numeric conversion, won't compile due to narrowing conversion. Assume `int` and `long` are both 4 bytes.

```
1 | int main()
2 | {
3 |     int a { 5 }; // 1a
4 |     int b { 'a' }; // 1b
5 |     int c { 5.4 }; // 1c
6 |     int d { true }; // 1d
7 |     int e { static_cast<int>(5.4) }; // 1e
8 |
9 |     double f { 5.0f }; // 1f
10 |    double g { 5 }; // 1g
11 |
12 |    // Extra credit section
13 |    long h { 5 }; // 1h
14 |
15 |    float i { f }; // 1i (uses previously defined variable f)
16 |    float j { 5.0 }; // 1j
17 |
18 | }
```

1a) [Show Solution](#) (javascript:void(0))²

1b) [Show Solution](#) (javascript:void(0))²

1c) [Show Solution](#) (javascript:void(0))²

1d) [Show Solution](#) (javascript:void(0))²

1e) [Show Solution](#) (javascript:void(0))²

1f) [Show Solution](#) (javascript:void(0))²

1g) [Show Solution](#) (javascript:void(0))²

1h) [Show Solution](#) (javascript:void(0))²

1i) [Show Solution](#) (javascript:void(0))²

1j) [Show Solution](#) (javascript:void(0))²

Question #2

2a) Update the following program to use type aliases for degrees and radians values:

```

1  #include <iostream>
2
3  namespace constants
4  {
5      constexpr double pi { 3.14159 };
6  }
7
8  double convertToRadians(double degrees)
9  {
10     return degrees * constants::pi / 180;
11 }
12
13 int main()
14 {
15     std::cout << "Enter a number of degrees: ";
16     double degrees{};
17     std::cin >> degrees;
18
19     double radians { convertToRadians(degrees) };
20     std::cout << degrees << " degrees is " << radians << " radians.\n";
21
22     return 0;
23 }

```

[Show Solution \(javascript:void\(0\)\)²](#)

2b) Given the definitions for `degrees` and `radians` in the previous quiz solution, explain why the following statement will or won't compile:

```
1 | radians = degrees;
```

[Show Solution \(javascript:void\(0\)\)²](#)



Next lesson

11.1 [Introduction to function overloading](#)

3



Back to table of contents

4



Previous lesson

10.9 [Type deduction for functions](#)

5

6



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/>⁹ are connected to your provided email address.

153 COMMENTS

Newest ▼



Mr. Stiven

 May 20, 2025 4:40 am PDT

Just nitpicking, the solution removes spaces between identifier and opening curly brace in lines 5 and 22 (which differs from the original code, but OTOH makes it consistent with the no-space definition of `degrees` variable in line 19).

With that in mind, is there a reason to prefer any of these styles? I mean, are there points in favor or against putting a space between an identifier and a curly brace for variable initialization?

```
1 | Radians radians {...};  
2 | Radians radians{...};
```

PS. Just completed reading the first 10 chapters. This is guide is amazing!

 0  Reply



Copernicus

 May 7, 2025 4:52 am PDT

Question #2b

It will compile, because Degrees and Radians are just another name/identifier for type double.

 0  Reply



Copernicus

 May 7, 2025 4:49 am PDT

Question #2a

```

1  #include <iostream>
2
3  using Degrees = double;
4  using Radians = double;
5
6  namespace constants
7  {
8      constexpr double pi{ 3.14159 };
9  }
10
11 double convertToRadians(Degrees degrees)
12 {
13     return degrees * constants::pi / 180;
14 }
15
16 int main()
17 {
18     std::cout << "Enter a number of degrees: ";
19     Degrees degrees{};
20     std::cin >> degrees;
21
22     Radians radians{ convertToRadians(degrees) };
23     std::cout << degrees << " degrees is " << radians << " radians.\n";
24
25     return 0;
26 }

```

👍 0 ➡ Reply



Archid

🕒 May 2, 2025 6:52 am PDT

Is the convention to use plurals for typedefs?

The fundamental types are singular (double, int, etc.), but for your typedef you use plurals (Degrees, Radians)

👍 0 ➡ Reply



Hari

🕒 April 10, 2025 5:49 am PDT

I doubted myself at the trick question, and got it wrong (even though I knew the answer T.T)

👍 1 ➡ Reply



smart

🕒 March 4, 2025 9:18 am PST

Congratulations to everyone who has read the 10 chapters. (It took me 2-3 months)

👍 11 ➡ Reply



Nikolai

🕒 February 20, 2025 1:13 am PST

In the solution of the last problem shouldn't pi be defined with the Radians type and not double?



0

➡ Reply



Howie

🗨 Reply to [Nikolai](#)¹⁰ 🕒 March 15, 2025 9:15 am PDT

I tried it and the compiler gave me the error: 'pi' is not a member of 'constants'.

I think it's because `constexpr double` does not have the same type with `double`, since `constexpr` implicitly indicates a variable is a `const` type. This info is given in the Chapter 10.9.

Also note that `constexpr` is not a type keyword even though it implicitly defines a `const` variable. So `using Demo = constexpr double` is an invalid statement. Although `using Demo = const double` can compile correctly, you don't need it for `pi` in this case.



0

➡ Reply



Alex

Author

🗨 Reply to [Nikolai](#)¹⁰ 🕒 February 26, 2025 10:24 pm PST

I don't think so, as pi is just a ratio, not a value in radians.



0

➡ Reply



charlottenberggenie

🕒 February 18, 2025 3:53 am PST

Shouldn't 1g be numeric promotion? `int` to `double` is always safe and value preserving since double is a wider type.



1

➡ Reply



charlottenberggenie

🗨 Reply to [charlottenberggenie](#)¹¹ 🕒 February 18, 2025 4:56 am PST

Never mind, if it's not integral or floating point promotion then it isn't considered a promotion even if the conversion is widening...



0

➡ Reply



ice-shroom

🕒 February 5, 2025 9:00 am PST

Good afternoon, thanks for the chapter. I wanted to ask, do the memory addresses of variables change in RAM if we explicitly or implicitly convert them from one type to another? If so, how does this work for arithmetic operations and types with the `auto` keyword?

👍 0 ➡ Reply

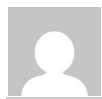


Alex Author

🗨 Reply to [ice-shroom](#)¹² ⌚ February 8, 2025 8:39 pm PST

The memory address of an instantiated object/variable doesn't change, and neither does the object's type. A conversion always produces a new (temporary) object to hold the converted value.

👍 0 ➡ Reply



rasp

⌚ January 15, 2025 3:22 am PST

Hi Alex, wanted to ask

```
1 | using Angle = double;
```

or

```
1 | using Degree = double;
2 | using Radian = double;
```

which would be an overall better approach in the quiz question.

thanks!

✎ Last edited 5 months ago by rasp

👍 0 ➡ Reply



BiscuitRE-L

🗨 Reply to [rasp](#)¹³ ⌚ February 18, 2025 7:33 am PST

what i went was with,

```
1 | using Degree = double;
2 | using Rad = double;
```

👍 0 ➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>

2. [javascript:void\(0\)](javascript:void(0))
3. <https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/>
4. <https://www.learncpp.com/>
5. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/>
6. <https://www.learncpp.com/chapter-10-summary-and-quiz/>
7. <https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/>
8. <https://www.learncpp.com/cpp-tutorial/function-template-instantiation/>
9. <https://gravatar.com/>
10. <https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/#comment-607920>
11. <https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/#comment-607854>
12. <https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/#comment-607407>
13. <https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/#comment-606579>
14. <https://g.ezoic.net/privacy/learncpp.com>