# 10.x — Chapter 10 summary and quiz

ALEX JANUARY 5, 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.

```
int main()
{
    int a { 5 }; // 1a
    int b { 'a' }; // 1b
    int c { 5.4 }; // 1c
    int d { true }; // 1d
    int e { static_cast<int>(5.4) }; // 1e

    double f { 5.0f }; // 1f
    double g { 5 }; // 1g

    // Extra credit section
    long h { 5 }; // 1h

    float i { f }; // 1i (uses previously defined variable f)
    float j { 5.0 }; // 1j

}
```

1a) Show Solution (javascript:void(0))

1b) Show Solution (javascript:void(0))

1c) Show Solution (javascript:void(0))

1d) Show Solution (javascript:void(0))

1e) Show Solution (javascript:void(0))

1f) Show Solution (javascript:void(0))

1g) Show Solution (javascript:void(0))

1h) Show Solution (javascript:void(0))

1i) Show Solution (javascript:void(0))

1j) Show Solution (javascript:void(0))

### Question #2

2a) Upgrade the following program using type aliases:

```
#include <iostream>

namespace constants
{
    inline constexpr double pi { 3.14159 };
}

double convertToRadians(double degrees)
{
    return degrees * constants::pi / 180;
}

int main()
{
    std::cout << "Enter a number of degrees: ";
    double degrees{};
    std::cin >> degrees;

    double radians { convertToRadians(degrees) };
    std::cout << degrees << " degrees is " << radians << " radians.\n";

    return 0;
}
```

Show Solution (javascript:void(0))

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

```
radians = degrees;
```

Show Solution (javascript:void(0))

Next lesson
11.1 Introduction to function overloading

Back to table of contents

Previous lesson
10.9 Type deduction for functions