8.x — Chapter 8 summary and quiz

ALEX SEPTEMBER 29, 2021

You made it! The topics in this chapter (particularly type aliases, overloaded functions, and function templates) appear everywhere in the C++ standard library. We've got one more chapter to go (introducing compound types), and then we'll be ready to dig into some of the most useful pieces of the standard library!

Quick 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 smaller numeric types to 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.

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.

Function overloading allows us to create multiple functions with the same name, so long as each identically named function has different set of parameter types (or the functions can be otherwise differentiated). Such a function is called an overloaded function (or

overload for short). Return types are not considered for differentiation.

When resolving overloaded functions, if an exact match isn't found, the compiler will favor overloaded functions that can be matched via numeric promotions over those that require numeric conversions. When a function call is made to function that has been overloaded, the compiler will try to match the function call to the appropriate overload based on the arguments used in the function call. This is called overload resolution.

An ambiguous match occurs when the compiler finds two or more functions that can match a function call to an overloaded function and can't determine which one is best.

A default argument is a default value provided for a function parameter. Parameters with default arguments must always be the rightmost parameters, and they are not used to differentiate functions when resolving overloaded functions.

Function templates allow us to create a function-like definition that serves as a pattern for creating related functions. In a function template, we use template types as placeholders for any types we want to be specified later. The syntax that tells the compiler we're defining a template and declares the template types is called a template parameter declaration.

The process of creating functions (with specific types) from function templates (with template types) is called function template instantiation (or instantiation) for short). When this process happens due to a function call, it's called implicit instantiation. An instantiated function is called a function instance (or instance for short, or sometimes a template function).

Template argument deduction allows the compiler to deduce the actual type that should be used to instantiate a function from the arguments of the function call. Template argument deduction does not do type conversion.

Template types are sometimes called generic types, and programming using templates is sometimes called generic programming.

In C++20, when the auto keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an abbreviated function template.

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

1b) Show Solution

1c) Show Solution

1d) Show Solution

1e) Show Solution

1f) Show Solution

1g) Show Solution

1h) Show Solution

1i) Show Solution

1j) Show Solution

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

2b) Building on quiz 2a, explain why the following statement will or won't compile:

```
radians =
degrees;
```

Show Solution

Question #3

3a) What is the output of this program and why?

```
#include <iostream>

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

void print(double x)
{
    std::cout << "double " << x <<
'\n';
}

int main()
{
    short s { 5 };
    print(s);

    return 0;
}
```

Show Solution

3b) Why won't the following compile?

```cpp
#include <iostream>

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

void print(int x=0)
{
    std::cout << "int " << x << '\n';
}

void print(double x)
{
    std::cout << "double " << x << '\n';
}

int main()
{
    print(5.0f);
    print();

    return 0;
}
```

Show Solution

3c) Why won't the following compile?

```cpp
#include <iostream>

void print(long x)
{
    std::cout << "long " << x << '\n';
}

void print(double x)
{
    std::cout << "double " << x << '\n';
}

int main()
{
    print(5);

    return 0;
}
```

Show Solution

Question #4

What is the output of this program and why?

```cpp
#include <iostream>

template <typename T>
int count(T x)
{
    static int c { 0 };
    return ++c;
}

int main()
{
    std::cout << count(1);
    std::cout << count(1);
    std::cout << count(2.3);
    std::cout <<
count<double>(1);

    return 0;
}
```

Show Solution

Question #5

5a) Write a function template named `add` that allows the users to add 2 values of the same type. The following program should run:

```cpp
#include <iostream>

// write your add function template
here

int main()
{
 std::cout << add(2, 3) << '\n';
 std::cout << add(1.2, 3.4) << '\n';

 return 0;
}
```

and produce the following output:

```
5
4.6
```

Show Solution

5b) Write a function template named `mult` that allows the user to multiply one value of any type and an integer. The following program should run:

```cpp
#include <iostream>

// write your mult function template
here

int main()
{
 std::cout << mult(2, 3) << '\n';
 std::cout << mult(1.2, 3) << '\n';

 return 0;
}
```

and produce the following output:

```
6
3.6
```

Show Solution

5c) Write a function template named `sub` that allows the user to subtract two values of different types. The following program should

run:

```
#include <iostream>

// write your sub function template
here

int main()
{
 std::cout << sub(3, 2) << '\n';
 std::cout << sub(3.5, 2) << '\n';
 std::cout << sub(4, 1.5) << '\n';

 return 0;
}
```

and produce the following output:

```
1
1.5
2.5
```

Show Solution

Next lesson

9.1 Using a language reference

Back to table of contents

Previous lesson

8.15 Function templates with multiple template types

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

DP N N FOUT

Newest