11.x — Chapter 11 summary and quiz

Nice work. Function templates may seem pretty complex, but they are a very powerful way to make your code work with objects of different types. We'll see a lot more template stuff in future chapters, so hold on to your hat.

Chapter Review

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 **type template parameters** 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**.

A **non-type template parameter** is a template parameter with a fixed type that serves as a placeholder for a constexpr value passed in as a template argument.

Quiz time

quiz 1

1a) 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

1b) Why won’t the following compile?

```
#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

1c) Why won't the following compile?

```
#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 #2

> Step #1

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

```
#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

> Step #2

Write a function template named `mult()` that allows the user to multiply one value of any type (first parameter) and an integer (second parameter). The function should return the same type as the first parameter. The following program should run:

```
#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

> Step #3

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

Question #3

What is the output of this program and why?

```
#include <iostream>

template <typename T>
int count(T) // This is the same as int count(T x), except we're not giving the
parameter a name since we don't use the parameter
{
    static int c { 0 };
    return ++c;
}

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

    return 0;
}
```

Show Solution

Question #4

What is the output of this program?

```
#include <iostream>

int foo(int n)
{
    return n + 10;
}

template <typename T>
int foo(T n)
{
    return n;
}

int main()
{
    std::cout << foo(1) << '\n'; // #1

    short s { 2 };
    std::cout << foo(s) << '\n'; // #2

    std::cout << foo<int>(4) << '\n'; // #3

    std::cout << foo<int>(s) << '\n'; // #4

    std::cout << foo<>(6) << '\n'; // #5

    return 0;
}
```

Show Solution