26.3 — Function template specialization

When instantiating a function template for a given type, the compiler stencils out a copy of the templated function and replaces the template type parameters with the actual types used in the variable declaration. This means a particular function will have the same implementation details for each instanced type (just using different types). While most of the time this is exactly what you want, occasionally there are cases where it is useful to implement a templated function slightly differently for a specific data type.

Using a non-template function

Consider the following example:

```
#include <iostream>

template <typename T>
void print(const T& t)
{
    std::cout << t << '\n';
}

int main()
{
    print(5);
    print(6.7);

    return 0;
}
```

This prints:

```
5
6.7
```

Now, let’s say we want double values (and only double values) to output in scientific notation.

One way to get different behavior for a given type is to define a non-template function:

```
#include <iostream>

template <typename T>
void print(const T& t)
{
    std::cout << t << '\n';
}

void print(double d)
{
    std::cout << std::scientific << d << '\n';
}

int main()
{
    print(5);
    print(6.7);

    return 0;
}
```

When the compiler goes to resolve `print(6.7)`, it will see that `print(double)` has already been defined by us, and use that instead of instantiating a version from `print(const T&)`.

This produces the result:

```
5
6.700000e+000
```

One nice thing about defining functions this way is that the non-template function doesn’t need to have the same signature as the function template. Note that `print(const T&)` uses pass by const reference, whereas `print(double)` uses pass by value.

Generally, prefer to define a non-template function if that option is available.

Function template specialization

Another way to achieve a similar result is to use **function template specialization** to create a explicit specialization of the `print()` function template for type double:

```
#include <iostream>

template <typename T>
void print(const T& t)
{
    std::cout << t << '\n';
}

// A specialized function template
template<> // template parameter declaration containing no template parameters
void print<double>(const double& d)
{
    std::cout << std::scientific << d << '\n';
}

int main()
{
    print(5);
    print(6.7);

    return 0;
}
```

Let’s take a closer look at our explicit function template specialization:

```
template<> // template parameter declaration containing no template parameters
void print<double>(const double& d)
```

First, we need a template parameter declaration to tell the compiler that we’re defining a function template. However, in this case, we don’t actually need any template parameters, so we don’t put anything between the angled brackets. On the next line, `print<double>` tells the compiler that we’re specializing the `print` template function for type `double`. The rest of the function is the same as the non-template version.

This prints the same result as above.

A function template specialization must have the same signature as the function template (e.g. one can’t use pass by const reference and the other pass by value). Also note that if a matching non-template function and a template function specialization both exist, the non-template function will take precedence. Explicit specializations are not implicitly inline, so if you define one in a header file, make sure you `inline` it to avoid ODR violations.

Just like normal functions, function template specializations can be deleted (using `= delete`) if you want any function calls that resolve to the specialization to produce a compilation error.

In general, you should avoid function template specialization.

Function template specialization for member functions?

Now consider the following class template:

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

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

int main()
{
    // Define some storage units
    Storage i { 5 };
    Storage d { 6.7 };

    // Print out some values
    i.print();
    d.print();
}
```

This prints:

```
5
6.7
```

Let’s say we again want to create a version of the `print()` function that prints doubles in scientific notation. However, this time `print()` is a member function, so we can’t define a non-member function. So how might we do this?

Although it may seem like we need to use function template specialization here, that’s the wrong tool. Note that `i.print()` calls `Storage<int>::print()` and `d.print()` calls `Storage<double>::print()`. Therefore, if we want to change the behavior of this function when `T` is a double, we need to specialize `Storage<double>::print()`, which is a class template specialization, not a function template specialization!

So how might we do that? We’ll cover class template specialization in the next lesson.