19.3 — Function template specialization

ALEX OCTOBER 8, 2021

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 different for a specific data type.

Template specialization is one way to accomplish this.

Let's take a look at a very simple template class:

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

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

The above code will work fine for many data types:

```
int main()
{
    // Define some storage units
    Storage<int> nValue { 5 };
    Storage<double> dValue { 6.7 };

    // Print out some values
    nValue.print();
    dValue.print();
}
```

This prints:

```
5
6.7
```

Now, let's say we want double values (and only double values) to output in scientific notation. To do so, we can use a function template specialization (sometimes called a full or explicit function template specialization) to create a specialized version of the print() function for type double. This is extremely simple: simply define the specialized function (if the function is a member function, do so outside of the class definition), replacing the template type with the specific type you wish to redefine the function for. Here is our specialized print() function for doubles:

```cpp
template <>
void Storage<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}
```

When the compiler goes to instantiate Storage<double>::print(), it will see we've already explicitly defined that function, and it will use the one we've defined instead of stenciling out a version from the generic templated class.

The template <> tells the compiler that this is a template function, but that there are no template parameters (since in this case, we're explicitly specifying all of the types). Some compilers may allow you to omit this, but it's correct to include it.

As a result, when we rerun the above program, it will print:

```
5
6.700000e+000
```

Another example

Now let's take a look at another example where template specialization can be useful. Consider what happens if we try to use our templated Storage class with datatype const char*:

```cpp
#include <iostream>
#include <string>

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()
{
    // Dynamically allocate a temporary string
    std::string s;

    // Ask user for their name
    std::cout << "Enter your name: ";
    std::cin >> s;

    // Store the name
    Storage<char*> storage(s.data());

    storage.print(); // Prints our name

    s.clear(); // clear the std::string

    storage.print(); // Prints nothing
}
```

As it turns out, instead of printing the name, the second storage.print() prints nothing! What's going on here?

When Storage is instantiated for type char*, the constructor for Storage<char*> looks like this:

```cpp
template <>
Storage<char*>::Storage(char* value)
      : m_value { value }
{
}
```

In other words, this just does a pointer assignment (shallow copy)! As a result, m_value ends up pointing at the same memory location as string. When we delete string in main(), we end up deleting the value that m_value was pointing at! And thus, we get garbage when trying to print that value.

Fortunately, we can fix this problem using template specialization. Instead of doing a pointer copy, we'd really like our constructor to make a copy of the input string. So let's write a specialized constructor for datatype char* that does exactly that:

```
template <>
Storage<char*>::Storage(char* value)
{
    if (!value)
        return;

    // Figure out how long the string in value is
    int length { 0 };
    while (value[length] != '\0')
        ++length;
    ++length; // +1 to account for null terminator

    // Allocate memory to hold the value string
    m_value = new char[length];

    // Copy the actual value string into the m_value memory we just allocated
    for (int count=0; count < length; ++count)
        m_value[count] = value[count];
}
```

Now when we allocate a variable of type Storage<char*>, this constructor will get used instead of the default one. As a result, m_value will receive its own copy of string. Consequently, when we delete string, m_value will be unaffected.

However, this class now has a memory leak for type char*, because m_value will not be deleted when a Storagevariable goes out of scope. As you might have guessed, this can also be solved by specializing a Storage<char*> destructor:

```
template <>
Storage<char*>::~Storage()
{
    delete[] m_value;
}
```

Now when variables of type Storage<char*> go out of scope, the memory allocated in the specialized constructor will be deleted in the specialized destructor.

Although the above examples have all used member functions, you can also specialize non-member template functions in the same way.

Next lesson
19.4 Class template specialization

Back to table of contents

Previous lesson
19.2 Template non-type parameters

Newest

X

X