

26.6 — Partial template specialization for pointers

👤 ALEX 🕒 SEPTEMBER 11, 2023

In previous lesson [26.3 -- Function template specialization](https://www.learncpp.com/cpp-tutorial/function-template-specialization/) (<https://www.learncpp.com/cpp-tutorial/function-template-specialization/>), we took a look at a simple templated Storage class:

```
#include <iostream>

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

    ~Storage()
    {
    }

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

We showed that this class had problems when template parameter T was of type `char*` because of the shallow copy/pointer assignment that takes place in the constructor. In that lesson, we used full template specialization to create a specialized version of the Storage constructor for type `char*` that allocated memory and created an actual deep copy of m_value. For reference, here's the fully specialized `char*` Storage constructor and destructor:

```
// You need to include the Storage<T> class from the example above here

template <>
Storage<char*>::Storage(char* value)
{
    // 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];
}

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

While that worked great for `Storage<char*>`, what about other pointer types (such as `int*`)? It's fairly easy to see that if `T` is any pointer type, then we run into the problem of the constructor doing a pointer assignment instead of making an actual deep copy of the element being pointed to.

Because full template specialization forces us to fully resolve templated types, in order to fix this issue we'd have to define a new specialized constructor (and destructor) for each and every pointer type we wanted to use `Storage` with! This leads to lots of duplicate code, which as you well know by now is something we want to avoid as much as possible.

Fortunately, partial template specialization offers us a convenient solution. In this case, we'll use class partial template specialization to define a special version of the `Storage` class that works for pointer values. This class is considered partially specialized because we're telling the compiler that it's only for use with pointer types, even though we haven't specified the underlying type exactly.

...



```
#include <iostream>

// You need to include the Storage<T> class from the example above here

template <typename T>
class Storage<T*> // this is a partial-specialization of Storage that works with pointer types
{
private:
    T* m_value;
public:
    Storage(T* value) // for pointer type T
        : m_value { new T { *value } } // this copies a single value, not an array
    {
    }

    ~Storage()
    {
        delete m_value; // so we use scalar delete here, not array delete
    }

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

And an example of this working:

```
int main()
{
    // Declare a non-pointer Storage to show it works
    Storage<int> myint { 5 };
    myint.print();

    // Declare a pointer Storage to show it works
    int x { 7 };
    Storage<int*> myintptr { &x };

    // Let's show that myintptr is separate from x.
    // If we change x, myintptr should not change
    x = 9;
    myintptr.print();

    return 0;
}
```

This prints the value:

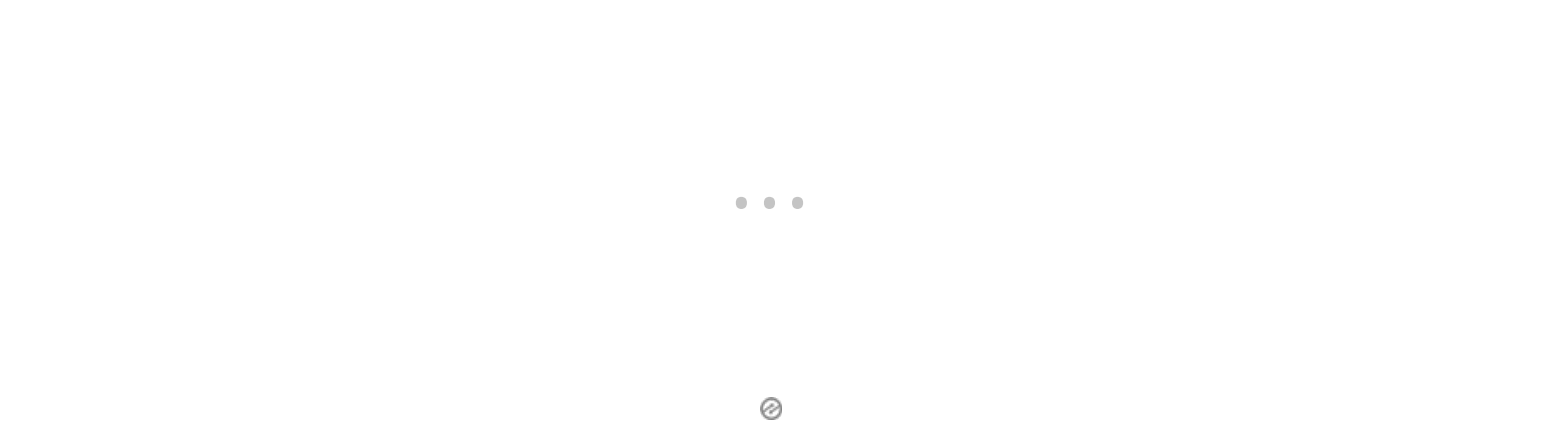
5
7

When `myintptr` is defined with an `int*` template parameter, the compiler sees that we have defined a partially specialized template class that works with any pointer type, and instantiates a version of `Storage` using that template. The constructor of that class makes a deep copy of parameter `x`. Later, when we change `x` to 9, the `myintptr.m_value` is not affected because it's pointing at its own separate copy of the value.

If the partial template specialization class did not exist, `myintptr` would have used the normal (non-partially-specialized) version of the template. The constructor of that class does a shallow copy pointer assignment, which means that `myintptr.m_value` and `x` would be referencing the same address. Then when we changed the value of `x` to 9, we would have changed `myintptr's` value too.

It's worth noting that because this partially specialized `Storage` class only allocates a single value, for C-style strings, only the first character will be copied. If we want to store entire C-style strings, we can do a full specialization for type `char*`. A fully specialized class or function will take precedence over a partially specialized version.

To fully specialize `Storage` for type `char*`, we have two options. We can either fully specialize the `Storage` class for `char*`, or we can fully specialize each of the individual functions we'll need (the constructor, destructor, and print function) for type `char*`. In this case, the `Storage` class has 3 member functions, and we'll need to specialize them all -- so we might as well just specialize the whole class.



Here's an example program that uses both partial specialization for pointers, and full specialization for `char*`:

```
#include <iostream>
#include <cstring>

// Our Storage class for non-pointers
template <typename T>
class Storage
{
private:
    T m_value;
public:
    Storage(T value)
        : m_value{ value }
    {
    }

    ~Storage()
    {
    }

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

// Partial-specialization of Storage class for pointers
template <typename T>
class Storage<T*>
{
private:
    T* m_value;
public:
    Storage(T* value)
        : m_value{ new T { *value } } // this copies a single value, not an array
    {
    }

    ~Storage()
    {
        delete m_value;
    }

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

// Full specialization of Storage class for char*
```

```

// Full specialization of Storage class for char
template <>
class Storage<char*>
{
private:
    char* m_value;
public:
    // Full specialization of constructor for type char*
    Storage(char* value)
    {
        // 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];
    }

    // Full specialization of destructor for type char*
    ~Storage()
    {
        delete[] m_value;
    }

    // Full specialization of print function for type char*
    // Without this, printing a Storage<char*> would call Storage<T*>::print(), which only prints the first char
    void print() const
    {
        std::cout << m_value;
    }
};

int main()
{
    // Declare a non-pointer Storage to show it works
    Storage<int> myint{ 5 };
    myint.print();

    // Declare a pointer Storage to show it works
    int x{ 7 };
    Storage<int*> myintptr{ &x };

    // If myintptr did a pointer assignment on x,
    // then changing x will change myintptr too
    x = 9;
    myintptr.print();

    // Dynamically allocate a temporary string
    char* name{ new char[40] { "Alex" } };

    // Store the name
    Storage<char*> myname{ name };

    // Delete the temporary string
    delete[] name;

    // Print out our name to prove we made a copy
    myname.print();
}

```

This works as we expect:

```

5
7
Alex

```

If we had decided to fully specialize the individual functions instead, they would look like this:

```

// Full specialization of constructor for type char*
template <>
Storage<char*>::Storage(char* value)
{
    // 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];
}

// Full specialization of destructor for type char*
template<>
Storage<char*>::~~Storage()
{
    delete[] m_value;
}

// Full specialization of print function for type char*
// Without this, printing a Storage<char*> would call Storage<T*>::print(), which only prints the first char
template<>
void Storage<char*>::print() const
{
    std::cout << m_value;
}

```

You can replace the definition of `Storage<char*>` in the prior example with these individual functions, and the example will compile and run the same.

Using partial template class specialization to create separate pointer and non-pointer implementations of a class is extremely useful when you want a class to handle both differently, but in a way that's completely transparent to the end-user.

...



[Next lesson](#)

26.x [Chapter 26 summary and quiz](#)



[Back to table of contents](#)



[Previous lesson](#)

26.5 [Partial template specialization](#)

Leave a comment...

 Name*


 Email* 

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!🔗

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

114 COMMENTS

Newest ▾

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info: ☐

OKAY

✕