26.6 — Partial template specialization for pointers

learncpp.com/cpp-tutorial/partial-template-specialization-for-pointers/

In previous lesson 26.4 -- Class template specialization, we took a look at a simple templated `Storage` class, along with a specialization for type `double`:

```
#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';
    }
};

template<>
void Storage<double>::print() // fully specialized for type double
{
    std::cout << std::scientific << m_value << '\n';
}

int main()
{
    // Define some storage units
    Storage i { 5 };
    Storage d { 6.7 }; // will cause Storage<double> to be implicitly instantiated

    // Print out some values
    i.print(); // calls Storage<int>::print (instantiated from Storage<T>)
    d.print(); // calls Storage<double>::print (called from explicit specialization
of Storage<double>::print())
}
```

However, as simple as this class is, it has a hidden flaw: it compiles but malfunctions when `T` is a pointer type. For example:

```
int main()
{
    double d { 1.2 };
    double *ptr { &d };

    Storage s { ptr };
    s.print();

    return 0;
}
```

On the authors machine, this produced the result:

```
0x7ffe164e0f50
```

What happened? Because `ptr` is a `double*`, `s` has type `Storage<double*>`, which means `m_value` has type `double*`. When the constructor is invoked, `m_value` receives a copy of the address that `ptr` is holding, and it is this address that is printed when the `print()` member function is called.

So how do we fix this?

One option would be to add a full specialization for type `double*`:

```
#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';
    }
};

template<>
void Storage<double*>::print() // fully specialized for type double*
{
    if (m_value)
        std::cout << std::scientific << *m_value << '\n';
}

template<>
void Storage<double>::print() // fully specialized for type double (for comparison,
not used)
{
    std::cout << std::scientific << m_value << '\n';
}

int main()
{
    double d { 1.2 };
    double *ptr { &d };

    Storage s { ptr };
    s.print(); // calls Storage<double*>::print()

    return 0;
}
```

This now prints the correct result:

```
1.200000e+00
```

But this only solves the problem when `T` is of type `double*`. What about when `T` is `int*`, or `char*`, or any other pointer type?

We really don’t want to have to create a full specialization for every pointer type. And in fact, it’s not even possible, because the user can always pass in a pointer to a program-defined type.

Partial template specialization for pointers

You might think to to try creating a template function overloaded on type `T*`:

```
// doesn't work
template<typename T>
void Storage<T*>::print()
{
    if (m_value)
        std::cout << std::scientific << *m_value << '\n';
}
```

Such a function is a partially specialized template function because it’s restricting what type `T` can be (to a pointer type), but `T` is still a type template parameter.

Unfortunately, this doesn’t work for a simple reason: as of the time of writing (C++23) functions cannot be partially specialized. As we noted in lesson 26.5 -- Partial template specialization, only classes can be partially specialized.

So let’s partially specialize the `Storage` class instead:

```
#include <iostream>

template <typename T>
class Storage // This is our primary template class (same as previous)
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

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

template <typename T> // we still have a type template parameter
class Storage<T*> // This is partially specialized for T*
{
private:
    T* m_value {};
public:
    Storage(T* value)
      : m_value { value }
    {
    }

    void print();
};

template <typename T>
void Storage<T*>::print() // This is a non-specialized function of partially
specialized class Storage<T*>
{
    if (m_value)
        std::cout << std::scientific << *m_value << '\n';
}

int main()
{
    double d { 1.2 };
    double *ptr { &d };

    Storage s { ptr }; // instantiates Storage<double*> from partially specialized
class
    s.print(); // calls Storage<double*>::print()

    return 0;
}
```

We've defined `Storage<T*>::print()` outside the class just to show how it's done, and to show that the definition is identical to the partially specialized function `Storage<T*>::print()` that did not work above. However, now that `Storage<T*>` is a partially specialized class, `Storage<T*>::print()` is no longer partially specialized -- it is a non-specialized function, which is why it is allowed.

It's worth noting that our type template parameter is defined as `T`, not `T*`. This means that `T` will be deduced as the non-pointer type, so we have to use `T*` anywhere we want a pointer to `T`. It's also worth a reminder that the partial specialization `Storage<T*>` needs to be defined after the primary template class `Storage<T>`.

Ownership and lifetime issues

The above partially specialized class `Storage<T*>` has another potential issue. Because `m_value` is a `T*`, it is a pointer to the object that is passed in. If that object is then destroyed, our `Storage<T*>` will be left dangling.

The core problem is that our implementation of `Storage<T>` has copy semantics (meaning it makes a copy of its initializer) but `Storage<T*>` has reference semantics (meaning it's a reference to its initializer). This inconsistency is a recipe for bugs.

There are a few different ways we can deal with such issues (in order of increasing complexity):

1. Make it clear really clear that `Storage<T*>` is a view class (with reference semantics), so it's on the caller to ensure the object being pointed to stays valid for as long as the `Storage<T*>` does. Unfortunately, because this partially specialized class must have the same name as a the primary template class, we can't give it a name like `StorageView`. So we're restricted to using comments or other things that might be missed. Not a great option.
2. Prevent use of `Storage<T*>` altogether. We probably don't need `Storage<T*>` to exist, as the caller can always dereference the pointer at the point of instantiation to use `Storage<T>` and make a copy of the value (which is semantically appropriate for a storage class).

However, while you can delete an overloaded function, C++ (as of C++23) won't let you delete a class. The obvious solution is to partially specialize `Storage<T*>` and then do something to make it not compile (e.g. `static_assert`) when the template is instantiated, this approach has one major downside: `std::nullptr_t` is not a pointer type, so `Storage<std::nullptr_t>` won't match `Storage<T*>`!

A better solution is to avoid partial specialization altogether, and use a `static_assert` on our primary template to ensure `T` is a type that we're okay with. Here's an example of that approach:

```
#include <iostream>
#include <type_traits> // for std::is_pointer_v and std::is_null_pointer_v

template <typename T>
class Storage
{
    // Make sure T isn't a pointer or a std::nullptr_t
    static_assert(!std::is_pointer_v<T> && !std::is_null_pointer_v<T>, "Storage<T*>
and Storage<nullptr> disallowed");

private:
    T m_value {};

public:
    Storage(T value)
      : m_value { value }
    {
    }

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

int main()
{
    double d { 1.2 };

    Storage s1 { d }; // ok
    s1.print();

    Storage s2 { &d }; // static_assert because T is a pointer
    s2.print();

    Storage s3 { nullptr }; // static_assert because T is a nullptr
    s3.print();

    return 0;
}
```

3. Have `Storage<T*>` make a copy of the object on the heap. If you do all the heap memory management yourself, this requires overloading the constructor, copy constructor, copy assignment, and destructor. An easier alternative is to just use `std::unique_ptr` (which we cover in lesson 22.5 -- std::unique_ptr):

```
#include <iostream>
#include <type_traits> // for std::is_pointer_v and std::is_null_pointer_v
#include <memory>

template <typename T>
class Storage
{
    // Make sure T isn't a pointer or a std::nullptr_t
    static_assert(!std::is_pointer_v<T> && !std::is_null_pointer_v<T>, "Storage<T*>
and Storage<nullptr> disallowed");

private:
    T m_value {};

public:
    Storage(T value)
      : m_value { value }
    {
    }

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

template <typename T>
class Storage<T*>
{
private:
    std::unique_ptr<T> m_value {}; // use std::unique_ptr to automatically deallocate
when Storage is destroyed

public:
    Storage(T* value)
      : m_value { std::make_unique<T>(value ? *value : 0) } // or throw exception
when !value
    {
    }

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

int main()
{
    double d { 1.2 };

    Storage s1 { d }; // ok
```

```
    s1.print();

    Storage s2 { &d }; // ok, copies d on heap
    s2.print();

    return 0;
}
```

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.