

## 13.8 — Partial template specialization for pointers

BY ALEX ON DECEMBER 5TH, 2016 | LAST MODIFIED BY ALEX ON MARCH 24TH, 2018

In previous lesson [13.5 -- Function template specialization](#), we took a look at a simple templated Storage class:

```
1  #include <iostream>
2
3  template <class T>
4  class Storage
5  {
6  private:
7      T m_value;
8  public:
9      Storage(T value)
10     {
11         m_value = value;
12     }
13
14     ~Storage()
15     {
16     }
17
18     void print()
19     {
20         std::cout << m_value << '\n';
21     }
22 };
```

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:

```
1  // You need to include the Storage<T> class from the example above here
2
3  template <>
4  Storage<char*>::Storage(char* value)
5  {
6      // Figure out how long the string in value is
7      int length=0;
8      while (value[length] != '\0')
9          ++length;
10     ++length; // +1 to account for null terminator
11
12     // Allocate memory to hold the value string
13     m_value = new char[length];
14
15     // Copy the actual value string into the m_value memory we just allocated
16     for (int count=0; count < length; ++count)
17         m_value[count] = value[count];
18 }
19
20 template<>
21 Storage<char*>::~~Storage()
22 {
23     delete[] m_value;
24 }
```

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.

```
1  #include <iostream>
2
3  // You need to include the Storage<T> class from the example above here
4
5  template <typename T>
6  class Storage<T*> // this is a partial-specialization of Storage that works with pointer types
7  {
8  private:
9      T* m_value;
10 public:
11     Storage(T* value) // for pointer type T
12     {
13         // For pointers, we'll do a deep copy
14         m_value = new T(*value); // this copies a single value, not an array
15     }
16
17     ~Storage()
18     {
19         delete m_value; // so we use scalar delete here, not array delete
20     }
21
22     void print()
23     {
24         std::cout << *m_value << '\n';
25     }
26 };
```

And an example of this working:

```
1  int main()
2  {
3      // Declare a non-pointer Storage to show it works
4      Storage<int> myint(5);
5      myint.print();
6
7      // Declare a pointer Storage to show it works
8      int x = 7;
9      Storage<int*> myintptr(&x);
10
11     // If myintptr did a pointer assignment on x,
12     // then changing x will change myintptr too
13     x = 9;
14     myintptr.print();
15
16     return 0;
17 }
```

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 the desire is to copy entire strings, a specialization of the constructor (and destructor) for type `char*` can be fully specialized. The fully specialized version will take precedence over the partially specialized version. Here's an example program that uses both partial specialization for pointers, and full specialization for `char*`:

```
1  #include <iostream>
2  #include <cstring>
3
4  // Our Storage class for non-pointers
5  template <class T>
6  class Storage
7  {
8  private:
9      T m_value;
10 public:
11     Storage(T value)
12     {
13         m_value = value;
14     }
15
16     ~Storage()
17     {
18     }
19
20     void print()
21     {
22         std::cout << m_value << '\n';
23     }
24 };
25
26 // Partial-specialization of Storage class for pointers
27 template <class T>
28 class Storage<T*>
29 {
30 private:
31     T* m_value;
32 public:
33     Storage(T* value)
34     {
35         m_value = new T(*value);
36     }
37
38     ~Storage()
39     {
40         delete m_value;
41     }
42
43     void print()
44     {
45         std::cout << *m_value << '\n';
46     }
47 };
48
49 // Full specialization of constructor for type char*
50 template <>
51 Storage<char*>::Storage(char* value)
52 {
53     // Figure out how long the string in value is
54     int length = 0;
55     while (value[length] != '\0')
56         ++length;
57     ++length; // +1 to account for null terminator
```

```

58
59     // Allocate memory to hold the value string
60     m_value = new char[length];
61
62     // Copy the actual value string into the m_value memory we just allocated
63     for (int count = 0; count < length; ++count)
64         m_value[count] = value[count];
65 }
66
67 // Full specialization of destructor for type char*
68 template<>
69 Storage<char*>::~~Storage()
70 {
71     delete[] m_value;
72 }
73
74 // Full specialization of print function for type char*
75 // Without this, printing a Storage<char*> would call Storage<T*>::print(), which only prints the first
76 // element
77 template<>
78 void Storage<char*>::print()
79 {
80     std::cout << m_value;
81 }
82
83 int main()
84 {
85     // Declare a non-pointer Storage to show it works
86     Storage<int> myint(5);
87     myint.print();
88
89     // Declare a pointer Storage to show it works
90     int x = 7;
91     Storage<int*> myintptr(&x);
92
93     // If myintptr did a pointer assignment on x,
94     // then changing x will change myintptr too
95     x = 9;
96     myintptr.print();
97
98     // Dynamically allocate a temporary string
99     char *name = new char[40]{ "Alex" }; // requires C++14
100
101     // If your compiler isn't C++14 compatible, comment out the above line and uncomment these
102     // char *name = new char[40];
103     // strcpy(name, "Alex");
104
105     // Store the name
106     Storage<char*> myname(name);
107
108     // Delete the temporary string
109     delete[] name;
110
111     // Print out our name
112     myname.print();
113 }

```

This works as we expect:

```

5
7
Alex

```

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.