templates

part 3

template specializations (from C++ Primer)

template specializations

it’s not always possible to write a single template that is best suited for every possible template argument with which the template might be instantiated

- general definition might not compile, or do the wrong thing for a type
- might be more efficient to take advantage of some specific knowledge of type

in such cases we can define a **specialized** version of the template

function template specialization example

```
// first version; can compare any two types
template <typename T> int compare(const T&, const T&);
// second version to handle string literals
template <size_t N, size_t M> int compare(const char (&)[N], const char (&)[M]);

const char *p1 = "hi", *p2 = "mom"; // character pointers
compare(p1, p2); // calls the first template
compare("hi", "mom"); // string literals, calls the second template with two nontype parameters
```

we can write a specialization to handle character pointers:

```
// special version of compare to handle pointers to character arrays
template <>
int compare(const char* const &p1, const char* const &p2) // what is “T” here?
{
    return strcmp(p1, p2);
}
```

function parameter types must be comply with the template we are specializing

some notes on function template specializations

when specializing, we are basically taking over from the compiler by supplying a definition of a specific instantiation of the template

this is different from overloading the function name

basic function matching rules:

- if there is only one nontemplate function in the set of equally good matches, the nontemplate function is called
- if there are no nontemplate functions in the set, but there are multiple function templates, and one of these templates is more specialized than any of the others, the more specialized function template is called

some notes on function template specializations

to specialize a template, a declaration for the original template must be in scope

a declaration for a specialization must be in scope before any code uses that instantiation of the template

unlike with ordinary classes and functions, missing declarations of specializations are hard to find, because the compiler will usually just generate code using the original template and not complain

best practices:

- templates and their specializations should be declared in the same header
- declarations for all the templates with a given name should appear first, followed by any specializations of those templates

full class template specialization example: `std::hash`

```
// open the std namespace so we can specialize std::hash
namespace std {
template <>                    // we're defining a specialization with
struct hash<Sales_data>        // the template parameter of Sales_data
{
    // the type used to hash an unordered container must define these types
    typedef size_t result_type;
    typedef Sales_data argument_type; // by default, this type needs ==
    size_t operator()(const Sales_data& s) const;
    // our class uses synthesized copy control and default constructor
};

size_t hash<Sales_data>::operator()(const Sales_data& s) const
{
   return hash()(s.bookNo) ^ hash()(s.units_sold) ^ hash()(s.revenue);
}
} // close the std namespace; note: no semicolon after the close curly

// uses hash<Sales_data>
std::unordered_multiset<Sales_data> SDset;
```

more class template specialization options

a class template specialization does not have to supply an argument for every parameter, allowing for **partial specializations** which are themselves templates:

```
template <typename T, typename U> class Foo {};       // original class template with parameters T & U
template <> class Foo<int, std::string> {};           // full specialization
template <typename T> class Foo<T, std::string> {};   // partial specialization where U is a string
template <typename T> class Foo<T, T*> {};            // partial specialization where U is pointer to T
```

can specialize member functions without specializing the whole class template:

```
template <typename T> struct Foo {
    Foo(const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // other members of Foo };
template<>        // we're specializing a template
void Foo<int>::Bar()    // we're specializing the Bar member of Foo<int>
{ // do whatever specialized processing that applies to ints }
```

variable templates
(new in C++14)

variable templates

in addition to functions and classes, variables can be templated as of C++14

usual template rules apply for things like specialization (example from Wikipedia):

```
template <typename T> constexpr T pi = T(3.1415926535897932384626433833);
template <> constexpr const char* pi<const char*> = “pi”; // specialization
template <> std::string pi<std::string> = “pi”;
```

note that template parameters must be provided

```
std::cout << pi<const char*> << " is " << pi<int> << std::endl;
std::cout << pi<double> << std::endl;
std::cout << pi<int> << std::endl;
```

variable template example: Fibonacci numbers

Fibonacci sequence: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

can be computed recursively:

```
int fib_recursive(int n)
{
    if (n < 2) return n;  // base cases
    return fib_recursive(n - 1) + fib_recursive(n - 2);  // recursive case
}
```

alternatively we can just iterate

but what if we want to compute a specific F_n at compile time, rather than run time?

variable template example: Fibonacci numbers

we can use a variable template to compute specific F_n at compile time

```
template <int N>
constexpr int fib = fib<N - 1> + fib<N - 2>;
```

will this work as written? what is missing?

more Fibonacci template fun: https://youtu.be/hErD6WGqPlA

variable template example: Fibonacci numbers

we can use a variable template to compute specific F_n at compile time

```
template <int N>
constexpr int fib = fib<N - 1> + fib<N - 2>;
```

will this work as written? what is missing?

```
template <>                    // base case specialization 0
constexpr int fib<0> = 0;

template <>                    // base case specialization 1
constexpr int fib<1> = 1;
```

more Fibonacci template fun: https://youtu.be/hErD6WGqPlA