Function templates

Michael R. Nowak
Texas A&M University

Motivation

- C++ requires us to declare variables and functions using specific types
- However, a lot of code looks the same for different types

Function templates

- In C++, function templates allow generic behavior to be encapsulated inside a function and then called for different types
  - The representation of such functions is almost identical to the functions that we've talked about to this point, with the exception the types of the parameters are left open as a template parameters
  - For instance, to parameterize the definition of a function that returns the minimum valued object of two objects, we would write:

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

Defining a function template

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

- We use the keyword `template`, followed by the type parameters that we'd like to announce inside angled brackets
- The keyword `typename` introduces a type parameter; here, the type parameter is identified by `T`
  - `T` represents an arbitrary type that is determined by the caller when the caller calls the function
  - Any type can be used as long as it has the operations used in the template defined; here `T` must support `operator>`

Using a function template

- We pass the data type as an argument to the function to initialize the type parameter T
    - Instead of writing multiple versions of the function `min`, as we did with function overloading, we can write min() and pass data type as a parameter and have the compiler generate the code for us
- A template parameter is a special kind of parameter that can be used to pass a type as argument:
    - Template parameters allow to pass also types to a function
    - We can use these parameters as if they were any other regular type
- The format for declaring function templates with type parameters is: `template<typename identifier> function_declaration`

Using a function template

- We can explicitly invoke min with template argument passed in <i>, an instance of the template is created by the compiler, with the template parameter T being replaced by type i

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

```
min<int>(7, 8)
```

```
int min (int a, int b)
{
    return (b > a) ? a : b;
}
```

- This process of replacing template parameters by concrete types is called instantiation
- To trigger the instantiation process, we call min<i>(a, b), where i is the argument to initialize the type parameter T

Code generated automatically by compiler

```
template<typename T> T min (T a, T b)
{
    return (b > a) ? a : b;
}
```

`min<int>(7, 8)`

```
int min (int a, int b)
{
    return (b > a) ? a : b;
}
```

`min<double>(7.0, 8.0)`

```
double min (double a, double b)
{
    return (b > a) ? a : b;
}
```

`min<char>('a', 'b')`

```
char min (char a, char b)
{
    return (b > a) ? a : b;
}
```

(slide intentionally left blank)

Template argument deduction

- When we call a function template for some arguments, the compiler can infer the type parameter T based-on the arguments
- For instance, if we pass two objects of the same type to our min function, the compiler will conclude that T is of that type

```
min(2,4) // T is deduced as an int
min(2.2, 4.4) // T is deduced as a double
min('a', 's') // T is deduced as a char
min("a", "s") // T is deduced as a char*
```

  - However, if we passed two objects of different type to our min function, the compiler would be unable to deduce what type T is

```
min(2, 2.4) // ERROR: T cannot be deduced as both an int and a double
min(2.4, 2) // ERROR: T cannot be deduced as both a double and an int
min(2, 'a') // ERROR: T cannot be deduced as both an int and a char
```

Template argument deduction

```
min(2, 2.4) // ERROR: T cannot be deduced as both an int and a double
min(2.4, 2) // ERROR: T cannot be deduced as both a double and an int
min(2, 'a') // ERROR: T cannot be deduced as both an int and a char
```

- We can handle these errors by either:
  1. Casting the arguments so that they are of the same type:
     ```
     min(2, static_cast<int>(2.4))
     ```
  2. Explicitly stating what type T should be, thus preventing the compiler from attempting to deduce the type of T:
     ```
     min<double>(2, 2.4)
     ```
  3. Specifying in our function template definition that the parameters may be of different types and then letting the compiler figure out the return type:
     ```
     template<typename T1, typename T2> auto min (T1 a, T2 b)
     {
         return (b > a) ? a : b;
     }
     ```

Templates and separate compilation

- For each template instantiation, the compiler generates specific code for that instantiation
  - If you have N different kinds of instantiations for class/function, you will have N different copies of code

- Recall that C++ uses separate compilation to compile multiple translation units; i.e., compiler operates on a single translation unit at a time
  - When we #include a header file, we bring the contents of that file into our source file
  - The implementation details are in the cpp file, which our source file doesn't have access to until we link things together

Templates and separate compilation

- Templates must be fully defined in each translation unit
  - There are many different ways to approach this problem
  - ***For this class*, you will write templated class/function implementation details in the header file**

Parameterizing a function : before

Max.h

```
#ifndef MAX_H
#define MAX_H

int const& max(int const& a,
               int const& b);

#endif
```

Max.cpp

```
#include "Max.h"

int const& max(int const& a, int const& b) {
    return (a < b) ? b : a;
}
```

Parameterizing a function : after

Max.h

```
#ifndef MAX_H
#define MAX_H

template<typename T>
T const& max(T const& a, T const& b)
{
    return (a < b) ? b : a;
}

#endif
```