CS170#08

Function Templates

Vadim Surov

Outline

- Introduction
- Automatic Type Deduction
- Class Objects In Template Functions
- Multiple Template Parameters
- Explicit Template Specialization
- Overloaded Template Functions
- Explicit Instantiations

Introduction

- Overloading of the cube function:

```
int cube(int n) {
  return n * n * n;
}
long cube(long n) {
  return n * n * n;
}
float cube(float n) {
  return n * n * n;
}
double cube(double n) {
  return n * n * n;
}
```

Introduction

- This is convenient for users since they can use the cube function for 4 different data types

- Not so convenient for the programmer to maintain 4 and even more different versions of the same function

- We want to write one function and apply it to all possible different types

Introduction

- A **function template** (Ada, 1983) is a type-generic way of describing a function
- This kind of programming is referred to as **generic programming**
- Templates are also called parameterized types because the type is passed as a parameter to the function

Introduction

- Our new cube function as template:

```
template <typename T> T cube(T v) {
  return v * v * v;
}
```

- The keyword **template** indicates that the function is a template function

- We put the "type parameter" name in angle brackets < > with the **typename** keyword

Introduction

- The **typename** keyword can be replaced with the **class** keyword (but this is less clear)
- The rest of the function is the same, except that we replace the type (**int**, **float**, etc.) with the type parameter
- The type parameter must be a legal identifier
  - T is often used

Introduction

- Alternative indentations:

```
template <typename T> T cube(T v) {
  return v * v * v;
}

template <typename T>
T cube(T v) {
  return v * v * v;
}
```

Introduction

- Template declarations do *not* generate any code
  - Similar to **class** and **struct** declarations
  - Should usually belong in header files

- Code is only generated when the function is used (ex: called in main function)

- This automatic code generation is called ***template instantiation***

Introduction

- If we have this statement in our program:

```
int i = cube(2);
```

- Then code similar to this is generated:

```
int cube (int v) {
  return v * v * v;
}
```

Introduction

- A template is a way to describe to the compiler how to generate functions

- The compiler generates these functions based on the template
  - The generated program is no smaller than writing these functions yourself
  - This generation is done implicitly, so it is called ***implicit instantiation***
    - You cannot see the result of instantiation!

Automatic Type Deduction

- The compiler can usually deduce the type of the function arguments (i.e., the type of T)
- However, we can "force" a particular instantiation using
  - Explicit function call, or
  - Typecasting the argument

Automatic Type Deduction

- We modify our cube function to show the type information:

```
#include <typeinfo> // for use with typeid

template<typename T> T cube(T v){
  std::cout << "<" << typeid(v).name()
            << ">" << std::endl;
  return v * v * v;
}
```

- Learn more about **typeid** in CS225!

Automatic Type Deduction

```
// Compiler deduction Microsoft/Borland GNU
cube(2); // <int> <i>
cube(2.0f); // <float> <f>
cube(2.0); // <double> <d>
cube('A'); // <char> <c>

// Explicit call
cube<double>(2); // <double> <d>
cube<int>(2.1); // <int> <i> (warning)

// Explicit typecasting
cube((double) 2); // <double> <d>
cube((int) 2.1); // <int> <i> (no warning)
```

Class Objects In Template Functions

- User-defined class objects can also be used as type parameters

- Does the following compile?

```
StopWatch sw1(4);
StopWatch sw2;
sw2 = cube(sw1);
std::cout << sw2 << std::endl;
```

Class Objects In Template Functions

- This is what the compiler generates:

```
StopWatch cube(StopWatch v){
  return v * v * v;
}
```

- This will not compile if there is no overloaded `operator*` for StopWatch

- We need to define this:

```
StopWatch StopWatch::operator*(
            const StopWatch& rhs) {
  return StopWatch(seconds * rhs.seconds);
}
```

Class Objects In Template Functions

- Now this will compile:

```
StopWatch sw1(4);
        // Create a StopWatch set to 4 seconds
StopWatch sw2;
        // Create a StopWatch set to 0 seconds
sw2 = cube(sw1);
        // cube sw1, assign it to sw2
std::cout << sw2 << std::endl;
        // 00:01:04 (4*4*4 = 64 sec)
```

Multiple Template Parameters

- Let's try to create a generic max function:

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

- Using the function:

```
int i = Max(2, 5); // i = 5
double d = Max(2.2, 5.5); // d = 5.5
double e = Max(2.2, 5);
```

Multiple Template Parameters

- To mix types, we need to specify both parameters:

```
template<typename T1, typename T2>
T1 Max(T1 a, T2 b) {
  return a > b ? a : b;
}
```

- Using the function:

```
double d = Max(2.2, 5); // d = 5.0
double e = Max(2, 5.5);
```

Multiple Template Parameters

- We have to add a third type:

```
template<typename T1, typename T2, typename T3>
T1 Max(T2 a, T3 b) {
  return a > b? a : b;
}
```

- How do we use this function?

```
double d = Max(2.2, 5); // Error
d = Max(2, 5.5); // Error
```

- The compiler cannot deduce the return type
  - User must specify it

Multiple Template Parameters

- Ways of using the function:

```
d = Max<double, int, double>(2, 5.5);
d = Max<double, double, int>(2.2, 5);
d = Max<double>(2, 5.5);
// Possible warning
d = Max<double, int, int>(2, 5.5);
// Possible warning
d = Max<int, int, double>(2, 5.5);
```

Explicit Template Specialization

- Sometimes we may need to handle "special cases" of our template functions

- Example

```
template<typename T>
bool equal(T a, T b) {
  std::cout << a << " and " << b << " are ";
  if (a == b)
    std::cout << "equal." << std::endl;
  else
    std::cout << "not equal." << std::endl;
  return a == b;
}
```

Explicit Template Specialization

- Example usage:

```
int a = 5, b = 5, c = 8;

equal(a, b); // 5 and 5 are equal
equal(a, c); // 5 and 8 are not equal
equal(a + 3, c); // 8 and 8 are equal
equal(a, 5.0); // Error! Conflicting
               types for parameter 'T'
```

Explicit Template Specialization

- What about these?

```
const char s1[] = "One";
const char s2[] = "One";
const char s3[] = "Two";

equal(s1, s2); // One and One are not equal
equal(s1, s3); // One and Two are not equal
equal(s1, "One"); // One and One are not equal
equal(s1, s1); // One and One are equal
equal("One", "One"); // Undefined behaviour
```

Explicit Template Specialization

- This is what's actually happening:

```
bool equal(const char* a, const char* b)
{
  // Comparing pointers
  if (a == b)
  // Etc.
}
```

- We need a "specialized" version of our template function
- This is called *explicit template specialization*

Explicit Template Specialization

- Specialization of equal for C-style strings:

```
template<>
bool equal<const char*>(const char* a,
                        const char* b) {
  std::cout << a << " and " << b << " are ";
  bool same = !strcmp(a, b);
  if (same)
    std::cout << "equal." << std::endl;
  else
    std::cout << "not equal." << std::endl;
  return same;
}
```

Explicit Template Specialization

- The syntax is similar to normal templates, except that there are empty angle brackets <> after the **template** keyword
- The specialization must be *after* the template declaration
  - However, you can use a prototype for your template function

Explicit Template Specialization

- Now this works as expected:

```
const char s1[] = "One";
const char s2[] = "One";
const char s3[] = "Two";

equal(s1, s2);        // One and One are equal
equal(s1, s3);        // One and Two are not equal
equal(s1, "One");     // One and One are equal
equal(s1, s1);        // One and One are equal
equal("One", "One"); // One and One are equal
```

Explicit Template Specialization

- Actually, the second angle brackets after the function name is optional. So this:

```
template<>
bool equal <const char*>
   (const char* a, const char* b)
```

  can be changed to this:

```
template<>
bool equal
   (const char* a, const char* b)
```

Explicit Template Specialization

- The compiler's order of preference when choosing functions (from best to worst):
  - Regular functions
  - Explicit specializations
  - Template generated
- However, you can always force the compiler to use a function by stating it explicitly

Explicit Template Specialization

- Example:

```
// Template function
template<typename T> T cube(T value) {
  return value * value * value;
}
// Explicit specialization cube<int>
template<> int cube<int>(int value) {
  return value * value * value;
}
// Regular function
int cube(int value) {
  return value * value * value;
}
```

Explicit Template Specialization

- Using the functions:

```
cube(5);
cube(5.5);
cube<int>(5);
cube<double>(5.5);
cube('A');
```

Explicit Template Specialization

- Note that you cannot create an explicit specialization after an implicit instantiation is created:

```
void foo(void) {
  cube(5); // Implicit instantiation
}
// Explicit specialization cube<int> -
error!
template <>
int cube<int>(int v) {
  return v * v * v;
}
```

Overloaded Template Functions

- Suppose we create a template swap function:

```
template<typename T>
void swap(T& a, T& b) {
   T temp = a;
   a = b;
   b = temp;
}
```

- Example usage:

```
int i = 10, j = 20;
swap(i, j); // i = 20, j = 10
```

Overloaded Template Functions

- What if we try to swap arrays?

```
int a1[] = {1, 3, 5, 7, 9};
int a2[] = {2, 4, 6, 8, 10};
swap(a1, a2);
```

- This is what happens:

```
void swap(int a[5], int b[5]) {
   int temp[5] = a;
   a = b;
   b = temp;
}
```

- Error assignment of arrays

Overloaded Template Functions

- We can overload the function to deal with arrays:

```
template <typename T>
void swap(T* a, T* b, int size) {
  T temp;
  for (int i = 0; i < size; i++) {
    temp[i] = a[i];
    a[i] = b[i];
    b[i] = temp[i];
  }
}
```

Overloaded Template Functions

- Using the function:

```
int a1[] = {1, 3, 5, 7, 9};
int a2[] = {2, 4, 6, 8, 10};
int size = sizeof(a1) / sizeof(*a);
swap(a1, a2, size);
```

Explicit Instantiations

- So far we have been using template functions via *implicit instantiations*
  - The compiler generates the functions when it encounters the function call
  - This is the point of templates
- However, it is also possible to have *explicit instantiations*
  - Generating the function definition without a function call

Explicit Instantiations

- Example:

```
template <typename T>
T cube(T v){
  return v * v * v;
}
// Explicit instantiation of cube<int>
template int cube(int v);
```

- The angle brackets are omitted
- This is mainly useful for creating library files for distribution because uninstantiated template functions are not compiled into object files

Summary

- Function templates are used to create generic functions that can take any data type
  - They tell the compiler how to generate a function
  - Code is only generated when called
- Template functions can be called implicitly using automatic type deduction (implicit instantiation)
- They can also be called explicitly by specifying the types in angle brackets (explicit call)

Summary

- Class objects can be used in template functions like any other data type
- Multiple parameters can be used
- If a function acts differently for a data type, it can be defined using an explicit specialization
- A template function can also be overloaded to accept different numbers/types of parameters
- Use explicit instantiation to produce a function from a template without a function call