C++ Templates

Venkatesh

WDC

September 18, 2017

Outline

1. Introduction
2. Function Templates
   - Examples
   - Function Template Overloading
3. Class Templates
   - Examples
4. Type Checking
5. Source Code Organization
6. References

Introduction

Templates are introduced for the purpose of design, implementation, and use of the standard library

C++ STL

string, ostream, regex, complex, list, map, thread

Templates

- A template is a "pattern" that the compiler uses to generate a family of classes or functions
- Templates does not imply any run-time mechanisms
- The composition offered by templates is *type-safe*

Three kinds of templates

- Function templates,
- Class templates and,
- Variable templates (since C++14)

Function Templates

Function Templates Syntax

template <**class** identifier> function_declaration;
template <**typename** identifier> function_declaration;

Both expressions have the same meaning and behaviour

A type parameter need not be a class

make_pair function template

```
template<typename T1, typename T2>
pair<T1,T2> make_pair(T1 a, T2 b)
{
    return {a,b};
}
auto x = make_pair(1,2); // x is a pair<int,int>
auto y = make_pair(string("New York"),7.7); // y is a
    pair<string,double>
```

Template Instantiation

Template Instantiation

The process of generating a class or a function from a template

- Function template arguments are deduced from the function arguments
- If a template argument cannot be deduced from the function arguments, we must specify it explicitly
  - e.g: static_cast , dynamic_cast
- Class template parameters are never deduced

max function template

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

max(3, 7)
max(3.0, 7.0)
max(3, 7.0) // error
max<double>(3, 7.0)
max(double(3), 7.0)
```

Template Specialization

```
template<typename T>
inline std::string stringify(const T& x){
  std::ostringstream out;
  out << x;
  return out.str();
}

template<>
inline std::string stringify<bool>(const bool& x) {
  std::ostringstream out;
  out << std::boolalpha << x; // bools as "true" or "false"
  return out.str();
}
```

Function Template Overloading

- Several function templates with the same name is possible
- Combination of function templates and ordinary functions with the same name is possible

Overload resolution is used to find the right function or function template to invoke.

Overload Resolution Rules

1. A non-templated overload is preferred to templates
2. Only non-template and primary template overloads participate in overload resolution
3. Specializations are not overloads and are not considered
4. Overload resolution selects the best-matching function template
   - Now, specializations are examined to see if one is a better match

Overload resolution Examples

non-template vs overload templates

template <**class** T> **void** foo(T);
void foo(**int**);

foo(10); *//calls void foo(int)*
foo(10u); *//calls void foo(T) with $T = unsigned$*

Overload resolution Examples

```
template< class T > void f(T);    // #1: template overload
template< class T > void f(T*);   // #2: template overload
void   f(double);    // #3: nontemplate overload
template<> void f(int);  // #4: specialization of #1

f('a');          // calls #1
f(new int(1)); // calls #2
f(1.0);          // calls #3
f(1);            // calls #4
```

Argument Substitution Failure

Substitution Failure Is Not An Error

```
template<typename Iter>
typename Iter::value_type mean(Iter first, Iter last); // #1

template<typename T> T mean(T*,T*); // #2

void f(vector<int>& v, int* p, int n)
{
    auto x = mean(v.begin(),v.end()); // OK: call #1
    auto y = mean(p,p+n); // OK: call #2
}
```

Class Templates

- Specification for generating classes based on parameters
- Class generated from a class-template is a ordinary class

Decrease of generated code

Code for a member function of a class-template is only generated if that member is used

Class Templates Declaration

- Members declaration is same as non-template class
- A template member need not be defined within the template class itself
- Members of a template class are themselves templates
    - parameterized by the template class parameters

Class Templates Examples

```
template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second) // defined in-class
      {a=first; b=second;}
    T getmax ();
};
```

Class Templates Examples

```
template <class T>
T mypair<T>::getmax () { // defined outside-class
  T retval = a > b ? a : b;
  return retval;
}

mypair<int> myobject (100, 75);
myobject.getmax() // generated on-demand
```

Non-type parameters for templates

Templates can also have regular typed parameters.

```
template <class T, int N>
class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}
```

Non-type parameters for templates

```
template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

mysequence <int,5> myints;
mysequence <double,5> mydoubles;
myints.setmember (0,100);
mydoubles.setmember (3,3.1416);
```

Class Templates Overload

It is not possible to overload a class template name.

template<**typename** T>
class String { /* ... */ };

class String { /* ... */ }; // *error : double definition*

Type Checking

- Type checking is done on the code generated by template instantiation
- Mismatch between what the programmer sees and what the compiler type checks can be a major problem
- Errors that relate to the use of template parameters cannot be detected until the template is used.

Type Equivalence

Aliases do not introduce new types.

vector<**unsigned char**> s3;
using Uchar = **unsigned char**;
vector<Uchar> s4;

Both s3, s4 are instances of same-class

Type Equivalence

- Types generated from a single template by different arguments are different types
  - Generated types from related arguments are not automatically related
- For example, assume that a Circle is a kind of Shape :

  Shape *p {**new** Circle(100)}; *// Circle* converts to Shape**
  vector<Shape> *q{**new** vector<Circle>{}};
  // error : no vector<Circle> to vector<Shape>* conversion*

Template Aliases

```
template<typename T, typename Allocator = allocator<T>>
    vector;

using Cvec = vector<char>; // both arguments are bound
Cvec vc = {'a', 'b', 'c'}; // vc is a vector<char,allocator<char>>
```

In the standard library std::string is

```
using string = std::basic_string<char>
```

Variable templates : since C++14

Syntax

template < parameter−list > variable−declaration

- A variable template defines a family of variables or static data members
- The declared variable name becomes a template name

Examples

```
template<class T>
constexpr T pi = T(3.1415926535897932385L);  // variable
    template

template<class T>
T circular_area(T r) // function template
{
  return pi<T> * r * r; // pi<T> is a variable template
    instantiation
}
```

Variadic templates (Since C++11)

- Variadic templates takes variable number of arguments
- Both function templates and class templates can be variadic

Syntax

template<**typename**... Values> **class** tuple; *// takes zero or more arguments*

template<**typename** First, **typename**... Rest> **class** tuple; *// takes one or more arguments*

Source Code Organization

- Place the declaration and definition of the templates in the header file
- Why can't I separate the definition of my templates into a .cpp file ?
  - Compiler has to see both the template definition (not just declaration) and the specific types for generating the code
  - Rely on your Compiler to optimize compile times and eliminate object code duplication

Drawbacks to the use of templates

- Many compilers lack clear instructions when they detect a template definition error.
- It can be difficult to debug code that is developed using templates

References

The C++ Programming Language [4th Edition] - Bjarne Stroustrup

Thank You