Introduction to Standard C++

Lecture 04: Template Functions and Template Classes

Massimiliano Culpo[1]

[1]CINECA - SuperComputing Applications and Innovation Department

07.04.2014

Table of contents - Language rules

Table of contents - Language rules

Template declarations (§14)

A template defines a family of classes or functions:

```
template <typename T>
T const& max (T const& a, T const& b) {
    // if a < b then use b else use a
    return a < b ? b : a;
}
```

1. A template declaration shall:
   - declare or define a function or a class
   - define a member function, a member class or a static data member
   - define a member template of a class or class template
2. A template declaration is also a definition if its declaration defines a function or a class

Template parameters (§14.1)

```
template<const X& x, int i> void f() {
  i++; // error: can't assign or modify
  &x; // OK
  &i; // error: can't take the address
  int& ri = i; // error
  const int& cri = i; // OK
}
```

1. There are three kinds of allowed template parameters:
   - non-type template parameters
   - type template parameters
   - template template parameters
2. A non-type template parameter shall be either:
   - an integral or enumeration type
   - a pointer/reference to object or function

Template parameters (§14.1)

```
template<class K, class V> class Map {
  std::vector<K> key;
  std::vector<V> value;
};

template<class K, class V,
template<class T> class C > class Map {
  C<K> key;
  C<V> value;
};
```

1. The keyword **class** names a type parameter
2. Template classes may also be used as template parameters

Template parameters (§14.1)

A default argument may be specified for any kind of template parameter:

```
template<class T1, class T2 = int> class A;

template <class T = float> struct B {};
template <template <class TT = float> class T>
struct A {
  inline void f();
  inline void g();
};
template <template <class TT> class T>
void A<T>::f() { T<> t; } // error
template <template <class TT = char> class T>
void A<T>::g() { T<> t; } // OK - T<char>
```

Table of contents - Language rules

Function templates (§14.5.6)

A function template defines an unbounded set of related functions:

```
// sort.hxx
template<class T> void sort( T& input );
```

The mere use of a function template can trigger an instantiation:

```
#include "sort.hxx"
std::vector< double > vec;
sort(vec);
```

A function template can be overloaded:

1. with other function templates
2. with normal functions

Overloading of function templates

```
/* Maximum of two ints */
int const& max (int const& a, int const& b);
/* Maximum of two values of any type */
template <typename T>
T const& max (T const& a, T const& b);
/* Maximum of three values of any type */
template <typename T>
T const& max (T const& a, T const& b, T const& c);

int main() {
  ::max(7, 42, 68);      // template for three arguments
  ::max(7.0, 42.0);      // max<double>
  ::max('a', 'b');       // max<char>
  ::max(7, 42);          // non-template for two ints
  ::max<>(7, 42);        // max<int>
  ::max<double>(7, 42); // max<double>
  ::max('a', 42.7);      // non-template for two ints
}
```

Table of contents - Language rules

Class templates (§14.5.1)

A class template defines an unbounded set of related types:

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

A member function may be defined outside the class in which it is declared:

```
template<class T> T& Array<T>::operator[](int i) {
  if (i<0 || sz<=i) error("Array: range error");
  return v[i];
}
```

Class templates (§14.5.1)

Similar rules apply to class member:

```
template<class T> struct A { class B; };
A<int>::B* b1;
template<class T> class A<T>::B { };
A<int>::B b2;
```

For a member function of a class template:

```
Array<int> v1(20);
Array<dcomplex> v2(30);
// Array<int>::operator[]()
v1[3] = 7;
// Array<dcomplex>::operator[]()
v2[3] = dcomplex(7,8);
```

the template arguments are determined by the type of the object for which the member function is called.

Member template (§14.5.2)

A template can be declared within a class or class template:

```
template<class T> struct string {

  template<class T2>
  int compare(const T2&);

  template<class T2>
  string(const string<T2>& s)
  { /* ... */ }
};

template<class T> template<class T2>
int string<T>::compare(const T2& s)
{ /* ... */ }
```

Member template (§14.5.2)

```
template <class T> struct AA {
  template <class C>
  virtual void g(C); // error
};
class B {
  virtual void f(int);
};
class D : public B {
  template <class T> void f(T); // no override
  void f(int i) { f<>(i); } // overriding
};
```

1. A local class shall not have member templates
2. A member function template:
   - shall not be virtual
   - does not override a virtual function

Class template partial specialization (§14.5.5)

```
template<class T1, class T2, int I>
class A { }; // Primary class template

template<class T, int I>
class A<T, T*, I> { }; // #1

template<class T1, class T2, int I>
class A<T1*, T2, I> { }; // #2
```

1. A class template may be partially specialized
2. Each class template partial specialization is a distinct template
3. Definitions shall be provided for its members

Table of contents - Language rules

The **typename** keyword (§14.6)

The keyword **typename** clarifies that an identifier is a type:

```
template <class T>
class A {
   typename T::SubType * ptr;
   /* ... */
};
```

Without **typename**, the expression:

```
T::SubType * ptr;
```

would be a multiplication of a static member of class **T** with **ptr**.

In general, **typename** has to be used whenever a name that depends on a template parameter is a type.

The **.template** construct (§14.6)

Consider the following example:

```
template <class T> class C {
public:
    template<class U> stringify();
}

template<int N>
void print(C<N> const& bs) {
  cout << bs.template stringify<char>();
}
```

Without that extra use of **template**, the compiler can't deduce the meaning of the $<$ token.

Dependent names (§14.6.2)

If a base class depends on a template parameter, the base class scope is not examined during unqualified name lookup:

```
typedef double A;

template<class T> class B {
  typedef int A;
};

template<class T> struct X : B<T> {
  A a; // a has type double
};
```

The consequence is that using a name **x** by itself is not always equivalent to **this->x**.

Q: Can you explain the following behavior?

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

int main() {
 string s;
 ::max("apple","peach");    // OK
 ::max("apple","tomato");   // ERROR
 ::max("apple",s);          // ERROR
}
```

Explicit instantiation (§14.8.1)

An empty template argument list can be used to indicate that a given use refers to a specialization of a function template:

```
template <class T> int f(T); // #1
int f(int); // #2
int k = f(1); // uses #2
int l = f<>(1); // uses #1
```

Implicit conversions will be performed if the parameter type contains no parameters that participate in template argument deduction:

```
template<class T> void f(T);
class Complex {
  Complex(double);
};
void g() {
  f<Complex>(1); // OK
}
```

Table of contents - Language rules

Summary: stages of a name resolution

When a series of tokens is to be resolved:

```
foo(12,45);
```

the C++ machinery goes through a number of codified stages:

1. (compile-time) Name lookup (§3.4)
2. (compile-time) Template argument deduction (§14.8.2)
3. (compile-time) Overload resolution (§13)
4. (compile-time) Access control (§11)
5. (run-time) Virtual function resolution (§10.3-§10.4)

It is important to know the subtleties of each of these stages to avoid common pitfalls.

Table of contents - Appendices


Abrahams, D., and Gurtovoy, A.
C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).
Addison-Wesley Professional, 2004.


Alexandrescu, A.
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.


Vandevoorde, D., and Josuttis, N. M.
C++ Templates: The Complete Guide, 1 ed.
Addison-Wesley Professional, Nov. 2002.