Chapter 16 Templates and Generic Programming

2008/4/21

Wontae Choi@ROPAS

Coverage

- 16.1 Template Definition
- 16.2 Instantiation
- 16.3 Template Compilation Model
- 16.4 Class Template Members
- 16.6 Template Specialization
- 16.7 Overloading and Function Templates

Chapter 16.1
Template Definition

Motivation

```
int compare(const string a, const string b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

Motivation

```
int compare(const string a, const string b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

```
int compare(const int & a, const int & b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

Motivation

```
int compare(const string a, const string b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

Very much similar

```
int compare(const int & a, const int & b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

Parametric Polymorphism

```
int compare(const [ ] a, const [ ] b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

- instantiate program code by replace [ ] with proper type

Template Definition

```
template <typename T> int compare(const T a, const T b);

template <typename T>
int compare(const T a, const T b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

- template <template parameters> definition_statement
- template <template parameters> declaration_statement

Using Template : Function

```
template <typename T>
int compare(const T a, const T b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

```
cout << compare(1,2);                                   // T = int
cout << compare(String("asd"),String("efg"));           // T = String
```

Using Template : Class

```
template <typename T>
class Queue{
    Queue();
    T & front();
    void push(const T &);
    void pop();
};
```

```
Queue<int> qi;
Queue<vector<int>> qv;
```

Template Parameter Scope

- The name of a template parameter can be used after i has been declared as a template parameter
- and until the end of template declaration of definition.

Template Parameter Restriction

```
template <class T> T calc(const T & a, const T & b){
   typedef double T;//ERROR : re-declaration of template parameter
}
```

```
template <class V, class V> ....
// ERROR : reuse of template parameter name V
```

```
template <class V, U> ....
// ERROR : must precede U by either typename or class
```

Typename, Class?

- Class is old style. Recommend Typename
- Typename can replace Class

Nontype Template Parameter

- Nontype parameters are replaced by values
- The type of that value is specified in the template parameter list.
- expression that evaluate to same value makes same instance
- expression must be *compile time constant*

```
template <class T, int SIZE>
void array_init(T (&parm)[SIZE]){
   for(int i =0; i != SIZE ; ++i){
      parm[i]=0;
   }
}
```

Type-Independent Code

- avoid to use type dependent method
- the type parameters to the template are *const references*

16.2 Instantiation

Class Instantiation

```
Queue<int> q1;
Queue<vector<int>> q2;
```

Function Instantiation

```
compare(1,2);

compare(String(“asdasd”),String(“zxxzczcx”));
```

Function Instantiation

```
compare(1,2);

compare(String(“asdasd”),String(“zxxzczcx”));
```

- Template Argument Deduction

Template Argument Deduction

```
compare((int) 1 , (int) 2);

compare((short) 1 , (short) 2);
```

- different template instance

Template Argument Deduction

```
template <typename T> compare(T &a, T &b);
Foo a;
const Foo b;

compare(a, a);
compare(b, b);
```

- different template instance

Template Argument Deduction

```
template <typename T> compare(T a, T b);
Foo a;
const Foo b;

compare(a, a);
compare(b, b);
```

- same template instance (using copy constructor)

Template Argument Deduction

```
template <typename T> compare(T a, T b);
Foo a[10];
Foo b[12];

compare(a, a);              //compare<T>(T * a, T * a);
compare(b, b);              //compare<T>(T * b, T * b);
```

- same template instance

Template Argument Deduction

```
template <typename T> compare(T & a, T & b);
Foo a[10];
Foo b[12];

compare(a, a);              //compare<T>(Foo &[10], Foo &[10]);
compare(b, b);              //compare<T>(Foo &[12], Foo &[12]);
```

- different template instance

Template Argument Deduction

```
class Foo{
   void func(int(*) (string & a, string & b));
   void func(int(*) (int & a, int & b));
}

func(compare);              //error : which instantiation to use??
```

Explicit Template Argument

```
class Foo{
   void func(int(*) (string & a, string & b));
   void func(int(*) (int & a, int & b));
}

func(compare<int>);              //Happy now
```

Explicit Template Argument

```
template <class T, class U> ??? sum(T, U)

                                        //what to return?
```

```
template <class R, class T, class U> R sum(T, U)

sum<int>(1, 0.2);                       //Happy now
```

16.3 Template Compilation Model

Inclusion Compilation Model

- the compiler must see the definition for any template that is used

Seperate Compilation Model

- the compiler keeps track of the associated template definition.
- However, we must tell the compiler to remember a given template definition. USE “export” keyword
- NO implementation.....

Compare

```
//header.h
template <class T>
int compare(T & a, T & b);

#include “contents.cc”

//content.cc
template<class T>
int compare(T & a, T & b){
......
}
```

VS

```
//header.h
template <class T>
int compare(T & a, T & b);


//contents.cc
export template<class T>
int compare(T & a, T & b){
......
}
```

16.4 Class Template Members

Definition

```
template <typename T>
class Foo{
   void bar();                          //declaration
}
```

```
template <typename T>
void Foo<T>::bar(){
.....
}
```

Friend

- friend class Foo1;;
  - Friend of All class template instance.
- friend template<typename T> class Foo2;
  - All instances are friend of all class instance.
- friend class Foo2<int>;
  - only specific instances are friend.

Member Template

- template as member of class template

```
template <class T>
class Foo{
   template <class TT> class BAR;
};
```

```
template <class T>
template <class TT>
class BAR{
   .......
};
```

- member template obey normal access control

16.6 Template Specialization

Specialization?

- It is not always possible to write a single template that is best suited for every possible template argument

```
template <typename T>
int compare(const T & a, const T & b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```

- If template argument is *char * .....*

Specialization!

- Declaration

```
template<>
int compare<const char *>(const char * &, const char & *);
```

```
template<>
int compare(const char * &, const char & *);
```

- Definition

```
template<>
int compare<const char *>(const char * &, const char & *){
    ....
}
```

Specialization....

- It is possible to define “Class Template Specialization”
- You can implement it totally independent way
- DON’T DO THAT!

Partial Specialization;;

- Original class template declaration

```
template <class T1, class T2>
class some_template{
    ...
}
```

- Partial Specialization

```
template <class T1>
class some_template<T1, int>{
    ...
}
```

Partial Specialization matching

```
some_template<int, string> foo;
                                        // uses template

some_template<string, int> bar;
                                        // uses template partial specialization
```

16.7 Overloading and Function Template

How to Resolve Overloaded name?

- 1. Exactly one function exist : chose it
- 2. Try to find in matching template
  - from template with least argument
- 3. Try to match non-template function
  - using “implicit type conversion”