Modern C++ - Templates

16. Templates

Independent concepts should be independently represented and should be combined only when needed. Otherwise unrelated concepts are bundled together or unnecessarry dependencies are created.

In modern programming languages *generics* provide a simple way to represent a wide range of general concepts parameterized by type(s) and a simple ways to combine them. In C++ generics are implemented as *templates*.

The standard library – which requires a greater degree of generality, flexibility and efficiency – is a good example of template collection.

Alternative implementation ideas

Suppose we want to implement a **max** function to select the greater value for many different types.

```
 1 int max( int a, int b)
 2 {
 3   if ( a > b )
 4     return a;
 5   else
 6     return b;
 7 }
 8 double max( double a, double b)
 9 {
10   if ( a > b )
11     return a;
12   else
13     return b;
14 }
15 ... and so on ...
```

How can we minimize the *copy-paste* effect? Just overloading many functions different only in the type of the parameter is not maintainable.

Preprocessor macro

One natural attempt is to use the preprocessor macro.

```
#define MAX(a,b)    ((a) > (b) ? (a) : (b))
```

The parentheses are necessary because of the possible context of the macro expansion. Still, macros have side effects:

```
x = MAX(a++,b);
```

After the expansion become:

```
x = ((a++) > (b) ? (a++) : (b))
```

so when **a++ > b** the increment of **a** will be executed twice.

Even when we avoid these issues, the macro is processed by the preprocessor, therefore it cannot communicate with the type system.

Templates

Ada designers described generic program units as: *a restricted form of context-sensitive macro facility*, and the designers of C++ claims: *templates are clever kind of macros that obeys the scope, naming, and type rules of C++*.

```
1 template <typename T>  // same as: <class T>
2 T max( T a, T b)
3 {
4   if ( a > b )
5     return a;
6   else
7     return b;
8 }
```

The template parameter can be a type, a const expression, or a pointer to an external object or function.

Instantiation, parameter deduction

Template is not a function, template is a pattern to generate a function The concrete function is the *specialization* created by automatic *instantiation*.

```
1 int i, j = 3, k = 4;
2 double x, y = 5.1, z = 3.14;
3
4 i = max( j, k);  // instantiates int max(int,int) specialization
5 x = max( y, z);  // instantiates double max(double,double) specialization
```

The compiler *deduces* the parameter types, then creates a specific version. The compiler *instantiates* the *specialization* with concrete types. Parameter deduction happens during compilation time.

Templates must be placed into header files, since the compiler must read their source.

Type exquivalence:

Templates are the same type only if all type parameters are equivalent.

```
1 C<char,10>      c1;
2 C<int, 10>      c2; // different than c1
3 C<int, 25-15>   c3; // same type as of c2
```

Explicit specialization

```
1 int i = 3, j = 4, k;
2 double  x = 3.14, y = 4.14, z;
3 const int ci = 6;
4
5 k = max( i, j);     // -> max(int, int)
6 z = max( x, y);     // -> max(double, double)
7 k = max( i, ci);    // -> max(int, int)
8
9 z = max( i, x);     // -> error
```

In the last case the parameter deduction failes, since there is contradiction between the type of **i** and **x**. This causes a compile time error.

To fix the problem, we try introduce an other version of **max** with more parameters.

```
1 template <class R, class T, class S>
2 R max( T a, S b)
3 {
4   if ( a > b )
5      return a;
```

```
 6   else
 7      return b;
 8 }
 9 z = max( i, x);     // error, no parameter deduction on return value
10                     // impossible to deduce R
```

Unfortunately, this does not work, since there is no parameter deduction on the return type, therefore **R** is not deducible. The information on **R** should be presented explicitly:

```
1 z = max<double>( i, x);  // ok, return type is double
2 z = max<int>( i, x);     // ok, return type is int
3 z = max<int,double,int>( i, x);   // ok, R, T, S is given explicitly
```

User specialization

In the earlier examples, the different versions of max used the same algorithm, only the parameters were different. There are cases, when the algorithm should be different for specific types.

```
 1 const char *s1 = "Hello";
 2 const char *s2 = "world";
 3
 4 template <>    // user specialization
 5 const char *max<const char *>( const char *s1, const char *s2)
 6 {
 7   return  strcmp( s1, s2) < 0;
 8 }
 9
10 std::cout << max( s1, s2) << std::endl;   // uses user specialization
```

The earlier templates would match for max(const char *, const char *), but the usage of **operator<** would be misleading to compare pointers. Therefore we define a *user specialization*, that matches for this case.

Template overloading

We can use all the templates introduced earlier *together*. The compiler will apply the *most specialized* version to instantiate.

```
1 template <class T>
2 T max( T a, T b)
```

```
 3 {
 4   if ( a > b )
 5     return a;
 6   else
 7     return b;
 8 }
 9 template <class R, class T, class S>
10 R max( T a, S b)
11 {
12   if ( a > b )
13     return a;
14   else
15     return b;
16 }
17 template <>    // user specialization
18 const char *max<const char *>( const char *s1, const char *s2)
19 {
20   return  strcmp( s1, s2) < 0;
21 }
22
23 i = max(3,4);                           // line 1
24 x = max<double>(3.14, 4);               // line 9
25 const char *m = max( "hello", "world");  // line 17
```

Class templates

Class templates define patterns for classes.

```
 1 template <typename T>
 2 class Matrix
 3 {
 4 public:
 5   Matrix( int cols, int rows) : _t ( new T[cols * rows] ) {}
 6   Matrix( const Matrix& rhs);
 7   Matrix& operator=(const Matrix& rhs);
 8   ~Matrix();
 9   // ...
10 private:
11   T* _t;
```

```
12   // ..
13 };
```

All of the memberfunctions of class templates are template functions. Therefore we have to put both the class template declaration and all memberfunction implementations into header files.

Template member functions have special syntax:

```
1 template <typename T>
2 Matrix<T>::Matrix( const Matrix& rhs) {  ... }
3 ^          ^             ^
4 |          |             |
5 class   constr name   inside Matrix namespace this is Matrix<T>
```

Since class template parameters regularly cannot be deduced from constructor parameters, objects from template classes must be explicitly specialized:

```
1 #include "matrix.h"
2
3 int main()
4 {
5   matrix<int>  m(10,20);
6
7   m.at(2,3) = 1;
8   cout << m(2,3) << endl;
9 }
```

Class template specialization

We can specialize class templates writing a completely new version for a specific template parameter. The specialization does not need to be follow the interface of the generic version (although this is not a good design decision).

```
1 template <>
2 class Matrix<bool>
3 {
4 public:
5   // possibly different interface
6   // ...
7 private:
```

```
 8   // possibly completely different implementations
 9   // ...
10 };
11
12 void f()
13 {
14   Matrix<int>     mi(10,20);  // use generic version
15   Matrix<double>  md(20,40);  // use generic version
16   Matrix<bool>    mb(20,40);  // use bool specialization
17   // ...
18 }
```

The complete example

Here we provide the template version of the vector class we implemented in Chapter 15.

```
 1 // vector.h
 2 #ifndef VECTOR_H
 3 #define VECTOR_H
 4
 5 #include <stdexcept>
 6
 7 template <typename T>
 8 class Vector
 9 {
10 public:
11   Vector();     // constructor
12
13   Vector(const Vector& rhs);            // copy constructor
14   Vector& operator=(const Vector& rhs); // assignment operator
15
16   ~Vector();    // destructor
17
18   int     size() const;    // actual size
19
20   T& operator[](int i);        // unchecked access
21   T  operator[](int i) const;  // unchecked access, const member
22
```

```
23   void    push_back(T d);       // append to end
24   void    pop_back();           // remove from end;
25 private:
26   int     _size;        // actual number of elements
27   int     _capacity;    // buffer size
28   T*      _ptr;         // pointer to buffer
29
30   void copy(const Vector& rhs);   // private helper function
31   void release();                 // private helper function
32   void grow();                    // reallocate buffer
33 };
34
35 // the implementation is in the same header file
36 template <typename T>
37 Vector<T>::Vector()
38 {
39   _capacity = 4;
40   _size = 0;
41   _ptr = new T[_capacity];
42 }
43 template <typename T>
44 Vector<T>::Vector(const Vector& rhs)
45 {
46   copy(rhs);
47 }
48 template <typename T>
49 Vector<T>& Vector<T>::operator=(const Vector& rhs)
50 {
51   if ( this != &rhs )  // avoid x = x
52   {
53     release();
54     copy(rhs);
55   }
56   return *this;  // for x = y = z
57 }
58 template <typename T>
59 Vector<T>::~Vector()
60 {
61   release();
```

```
 62 }
 63 template <typename T>
 64 void Vector<T>::copy(const Vector& rhs)
 65 {
 66   _capacity = rhs._capacity;
 67   _size = rhs._size;
 68   _ptr = new T[_capacity];
 69 
 70   for (int i = 0; i < _size; ++i)
 71     _ptr[i] = rhs._ptr[i];
 72 }
 73 template <typename T>
 74 void Vector<T>::release()
 75 {
 76   delete [] _ptr;
 77 }
 78 template <typename T>
 79 void Vector<T>::grow()
 80 {
 81   T *_oldptr = _ptr;
 82   _capacity = 2 * _capacity;
 83   _ptr = new T[_capacity];
 84 
 85   for ( int i = 0; i < _size; ++i)
 86     _ptr[i] = _oldptr[i];
 87 
 88   delete [] _oldptr;
 89 }
 90 template <typename T>
 91 int Vector<T>::size() const
 92 {
 93   return _size;
 94 }
 95 template <typename T>
 96 T& Vector<T>::operator[](int i)
 97 {
 98   return _ptr[i];
 99 }
100 template <typename T>
```

```
101 T Vector<T>::operator[](int i) const
102 {
103   return _ptr[i];
104 }
105 template <typename T>
106 void Vector<T>::push_back(T d)
107 {
108   if ( _size == _capacity )
109     grow();
110 
111   _ptr[_size] = d;
112   ++_size;
113 }
114 template <typename T>
115 void Vector<T>::pop_back()
116 {
117   if ( 0 == _size )
118     throw std::out_of_range("vector empty");
119 
120   --_size;
121 }
122 #endif /* VECTOR_H */
```

```
 1 #include <iostream>
 2 #include <string>
 3 #include "vector.h"
 4 
 5 template <typename T>
 6 void print(const vec<T>& v, char *name)
 7 {
 8   std::cout << s << " = [ ";
 9   for (int i = 0; i < v.size(); ++i )
10     std::cout << v[i] << " ";
11   std::cout << "]" << std::endl;
12 }
13 int main()
14 {
15   Vector<int> x;  // declare and fill x
16   for (int i = 0; i < 10; ++i )
17     x.push_back(i);
```

```
18
19   Vector<double> y;  // declare and fill y
20   for (int i = 0; i < 15; ++i )
21     y.push_back(i+20.5);
22
23   std::string s;
24   vector<std::string> z; // declare and fill z
25   for (char ch = 'a'; ch < 'z'; ++ch )
26   {
27     s += ch;
28     z.push_back(s);
29   }
30   print(x,"x");
31   print(y,"y");
32   print(z,"z");
33 }
```