

Object Oriented Programming (CS1143)

Week 12

Department of Computer Science

Capital University of Science and Technology (CUST)

Outline

- Function Templates
- Class Templates

Template Programming

- In this chapter we concentrate on generalization, which means to write a general program that can be used in several special cases.
- C++ calls this process template programming.
- We first discuss how to write general functions (called function templates) and then we discuss general classes (called class templates).

Outline

- Function Templates
- Class Templates

Function Template

- When programming in any language, we sometimes need to apply the same code to different data types.
- For example, we may have a function that finds the smaller of two integer data types, another function that finds the smaller of two floating-point data types, and so on.
- We can write a program to solve a problem with a generic data type.
- We can then apply the program to the specific data type we need.
- This is known as generic programming or template programming.
- Using function templates, actions can be defined when we write a program, and the data types can be defined when we compile the program.

Using a family of functions

- If we do not use template and generic programming, we must define a family of functions.
- For example, assume that we need to find the smaller between two characters, two integers, and two floating-point numbers.
- Since the types of the data are different, without templates we would need to write three functions

```
1  #include <iostream>
2  using namespace std;
3
4  // Function to find the smaller between two characters
5  char smaller (char first , char second )
6  {
7      if ( first < second )
8      {
9          return first ;
10     }
11     return second ;
12 }
13
14 // Function to find the smaller between two integers
15 int smaller (int first , int second )
16 {
17     if ( first < second )
18     {
19         return first ;
20     }
21     return second ;
22 }
```

```

24 // Function to find the smaller between two doubles
25 double smaller (double first , double second )
26 {
27     if ( first < second )
28     {
29         return first ;
30     }
31     return second ;
32 }
33
34 int main ()
35 {
36     cout << "Smaller of 'a' and 'B': " << smaller ('a', 'B') << endl;
37     cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
38     cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
39     return 0;
40 }

```

Smaller of 'a' and 'B': B

Smaller of 12 and 15: 12

Smaller of 44.2 and 33.1: 33.1

Using Function Template

- To create a template function, we can use a placeholder for each generic type.
- The template header contains the keyword `template` followed by a list of symbolic types inside two angle brackets.
- The template function header follows the rules for function headers except that the types are symbolic as declared in the template header.

```
template <typename T, typename U, ..., typename Z>  
T functionName (U first, ... Z last)  
{  
    ...  
}
```

Example

```
1  #include <iostream>
2  using namespace std;
3
4  // Definition of a template function
5  template <typename T>
6  T smaller (T first , T second )
7  {
8      if ( first < second )
9      {
10         return first ;
11     }
12     return second ;
13 }
14
15 int main ( )
16 {
17     cout << "Smaller of a and B: " << smaller ('a', 'B') << endl;
18     cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
19     cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
20     return 0;
21 }
```

Smaller of 'a' and 'B': B

Smaller of 12 and 15: 12

Smaller of 44.2 and 33.1: 33.1

Description

- We have only one generic type, but the type is used three times: twice as parameters and once as the return type.
- In this program the type of the parameters and the returned value are the same.
- We have saved code by writing only one template function instead of three overloaded functions.

Another Example

```
1  #include <iostream>
2  using namespace std;
3
4  // Definition of template function
5  template <typename T>
6  void exchange (T& first, T& second)
7  {
8      T temp = first;
9      first = second;
10     second = temp;
11 }
```

```

13 int main ()
14 {
15     // Swapping two int types
16     int integer1 = 5;
17     int integer2 = 70;
18     exchange (integer1, integer2);
19     cout << "After swapping 5 and 70: ";
20     cout << integer1 << " " << integer2 << endl;
21
22     // Swapping two double types
23     double double1 = 101.5;
24     double double2 = 402.7;
25     exchange (double1, double2);
26     cout << "After swapping 101.5 and 402.7: ";
27     cout << double1 << " " << double2 << endl;
28     return 0;
29 }

```

After swapping 5 and 70: 70 5

After swapping 101.5 and 402.7: 402.7 101.5

Instantiation

- Using template functions postpones the creation of non-template function definitions until compilation time.
- This means that when a program involving a function template is compiled, the compiler creates as many versions of the function as needed by function calls.
- This process is referred to as template instantiation, but it should not be confused with instantiation of an object from a type.

```
template <typename T>  
T smaller (T first, T second)  
{  
...  
}
```


Before Compilation

```
char smaller (char first, char second)  
{  
int smaller (int first, int second)  
{  
double smaller (double first, double second)  
{  
...  
}}
```

After compilation

Passing Arrays to Functions (W1, Slide 31)

```
1  #include <iostream>
2  using namespace std;
3
4  void printArray1 (int p[], int size)
5  {
6      for (int i = 0; i < size ; i++)
7      {
8          cout<< p[i]<<" ";
9      }
10     cout<<endl;
11 }
12
13 void printArray2 (int* p, int size)
14 {
15     for (int i = 0; i < size ; i++)
16     {
17         cout<< *(p+i)<<" ";
18     }
19     cout<<endl;
20 }
21
22
23 int main ()
24 {
25     int arr [5] = {1, 2, 3, 4, 5};
26     printArray1(arr, 5);
27     printArray2(arr, 5);
28
29     return 0;
30 }
```

 D:\Work Umair\4_CUST (1-9-22)\

```
1 2 3 4 5
1 2 3 4 5
```


Using Template Function

```
1  #include <iostream>
2  using namespace std;
3
4  // Definition of the print template function
5  template <typename T>
6  void print (T array[], int N)
7  {
8      for (int i = 0; i < N ; i++)
9      {
10         cout << array [i] << " ";
11     }
12     cout << endl;
13 }
14
15 int main ()
16 {
17     // Creation of two arrays
18     int arr1 [4] = {7, 3, 5, 1};
19     double arr2 [3] = {7.5, 6.1, 4.6};
20     // Calling template function
21     print (arr1, 4);
22     print (arr2, 3);
23     return 0;
24 }
```

Another Way to pass array to a function

```
1  #include <iostream>
2  using namespace std;
3
4  void printArray (int p[5])
5  {
6      for (int i = 0; i < 5 ; i++)
7      {
8          cout<< p[i]<<" ";
9      }
10     cout<<endl;
11 }
12
13 int main ()
14 {
15     int arr [5] = {1, 2, 3, 4, 5};
16     printArray(arr);
17
18     return 0;
19 }
```

Non-type Template Parameter

- Assume we want a function that prints the elements of any array regardless of the type of the element and the size of the array.
- We know that the type of each element can vary from one array to another, but the size of an array is always an integer (or unsigned integer).
- In the following program we have two template parameters, T and N.
- The parameter T can be any type; the parameter N is a non-type (the type is predefined as integer).

```

1  #include <iostream>
2  using namespace std;
3
4  // Definition of the print template function
5  template <typename T, int N>
6  void print (T (&array) [N])
7  {
8      for (int i = 0; i < N ; i++)
9      {
10         cout << array [i] << " ";
11     }
12     cout << endl;
13 }
14
15 int main ()
16 {
17     // Creation of two arrays
18     int arr1 [4] = {7, 3, 5, 1};
19     double arr2 [3] = {7.5, 6.1, 4.6};
20     // Calling template function
21     print (arr1);
22     print (arr2);
23     return 0;
24 }

```

```

7 3 5 1
7.5 6.1 4.6

```

Explicit Type Determination

If we try to call the function template to find the smaller between an integer and a floating-point value such as the following, we get an error message.

```
cout << smaller (23, 67.2); // Errors! Two different types for the same template type T.
```

In other words, we are giving the compiler an integral value of 23 for the first argument of type T and a floating-point value of 67.2 for the second argument of type T. The type T is one template type, and the values for it must always be the same as each other. The error with the previous case can be avoided if we define the explicit type conversion during the call. This is done by defining the type inside the angle brackets as shown below:

```
cout << smaller <double> (23, 67.2); // 23 will be changed to 23.0
```

We are telling the compiler that we want to use the version of the *smaller* programs in which the value of T is of type *double*. The compiler then creates that version and converts 23 to 23.0 before finding the *smaller*.

Overloading Function Templates

- We can also overload a function template to have several functions with the same name but different signatures.
- Normally, the template type is the same, but the number of parameters is different.
- In the following program, we overload the smaller template function to accept two or three parameters (we call it smallest because it uses more than two arguments).
- We have defined the second function in terms of the first one. This is why the second function is shorter.

```

1  #include <iostream>
2  using namespace std;
3
4  // Template function with two parameters
5  template <typename T>
6  T smallest (const T& first, const T& second)
7  {
8      if (first < second)
9      {
10         return first;
11     }
12     return second;
13 }
14
15 // Template function with three parameters
16 template <typename T>
17 T smallest (const T& first, const T& second, const T& third)
18 {
19     return smallest (smallest (first, second), third);
20 }

```

```

22 int main ( )
23 {
24     // Calling the overloaded version with three integers
25     cout << "Smallest of 17, 12, and 27 is ";
26     cout << smallest (17, 12, 27) << endl;
27
28     // Calling the overloaded version with three doubles
29     cout << "Smallest of 8.5, 4.1, and 19.75 is ";
30     cout << smallest (8.5, 4.1, 19.75) << endl;
31     return 0;
32 }

```

```

Smallest of 17, 12, and 27 is 12
Smallest of 8.5, 4.1, and 19.75 is 4.1

```


Outline

- Function Templates
- Class Templates

Class Template

- We know that a class is a combination of data members and member functions.
- We may also have a class with data types and another class with the same functionality but with different data types.
- In these cases, we can use a class template.

Example

```

1  #include <iostream>
2  using namespace std;
3
4  template <typename T>
5  class Fun
6  {
7      private:
8          T data;
9
10     public:
11         Fun (T data );
12         ~Fun ();
13         T get () const;
14         void set (T data);
15 };

```

```

17 // Constructor
18 template <typename T>
19 Fun <T> :: Fun (T d)
20 : data (d)
21 {
22 }
23
24 // Destructor
25 template <typename T>
26 Fun <T> :: ~Fun ()
27 {
28 }
29
30 // Accessor Function
31 template <typename T>
32 T Fun <T> :: get () const
33 {
34     return data;
35 }
36
37 // Mutator Function
38 template <typename T>
39 void Fun <T> :: set (T d)
40 {
41     data = d;
42 }

```

```

44 int main ( )
45 {
46     // Instantiation of four classes each with different data type
47     Fun <int> Fun1 (23);
48     Fun <double> Fun2 (12.7);
49     Fun <char> Fun3 ('A');
50     Fun <string> Fun4 ("Hello");
51
52     // Displaying the data values for each class
53     cout << "Fun1: " << Fun1.get() << endl;
54     cout << "Fun2: " << Fun2.get() << endl;
55     cout << "Fun3: " << Fun3.get() << endl;
56     cout << "Fun4: " << Fun4.get() << endl;
57
58     // Setting the data values in two classes
59     Fun1.set(47);
60     Fun3.set ('B');
61
62     // Displaying values for newly set data
63     cout << "Fun1 after set: " << Fun1.get() << endl;
64     cout << "Fun3 after set: " << Fun3.get() << endl;
65     return 0;
66 }

```

D:\Work Umair\4_CUST (1-9-22)\1_Teaching\

```

Fun1: 23
Fun2: 12.7
Fun3: A
Fun4: Hello
Fun1 after set: 47
Fun3 after set: B

```

This is all for Week 12