Unit 12 Class Templates

Structure:

12.1 Introduction

In the previous unit you have studied the methods to open and close files along with the methods to check whether a file opening was successful or not. You have also studied the I/O status flags, binary files and some very useful functions in C++. In this unit you are going to study in detail about the templates in C++. Template in C++ is newly added concept that enables us to define generic classes and functions and hence provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for variety of suitable data types and data structures. We can use templates to create family of class and functions. For example, a class template for an array will

enable us to create arrays of various data types. Similarly, a template for a function, let us say add() can be defined, that will enable us to create various versions of function add() to multiply int, float and double type values.

Objectives:

After studying this unit you should be able to:

- define templates, and the types of templates available in C++.
- describe how templates are dealt with by the C++ compiler.
- describe the scenarios in which templates are useful.
- discuss in detail the in-depths of template programming.
- discuss C++ programming using templates.

12.2 Class Templates

A template can be considered a kind of macro. When we define an object of a specific type for actual use, the definition of template for that class is substituted with the required data type. Since the template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

Consider a vector class defined as follows:

```
class vector
  {
    int *v;
   int size;
  public:
  vector(int m)
    {
        v= new int[size = m];
        for(int i=0; i<size; i++)
            v[i]=0;
     }
   vector (int *a)                  //create a vector from an array
     {
        for( int i=0; i<size ; i++)
         v[i]= a[i];
```

```
    }
 int operator*(vector &y)          //scalar product
    {
      int sum=0;
      for( int i=0; i<size; i++)
            sum= sum + this-> v[i] * y . v[i];
       return sum;
    }
};
```

The vector class can store an array of int numbers and perform the scalar product of two int vectors as shown below:

```
int main()
 {
    int x[3] = {1, 2, 3};
    int y[3] = {4, 5, 6};
    vector v1(3);                  //creates a null vector of three integers
    vector v2(3);
    v1 = x;                        //creates v1 from the array x
    v2= y;
   int R = v1 * v2;
   cout << "R = " << R;
   return 0;
 }
```

Now let's assume that we want to define a vector that can store an array of float values. This can be done by replacing the appropriate int declarations with float in the vector class. This means that the complete class needs to be redefined.

This helps us define a vector class with the data type as a parameter and then use this class to create a vector of any data type instead of defining a new class every time. This can be achieved by using the templates.

12.2.1 Implementing a Class Template

As it has already been mentioned, the templates allow us to define generic classes. It is a simple process to create a generic class using a template with anonymous type. The general format of class template is as follows:

```
class classname
 {
    //class member specification with anonymous type T wherever
     appropriate.
  …….
};
```

For example, the definition of vector class is given below:

```
template<class T>
class vector
{
        T* v;     //type T vector
        int size;
 public:
   vector(int m)
    {
        v= new T [ size = m];
        for (int i=0; i<size; i++)
          v[i]=0;
    }
   vector (T *a)
    {
        for(int i=0; i<size; i++)
        v[i] = a[i];
   }
 T operator* (vector &y)
 {
   T sum = 0;
    for (int i =0; i<size ; i++)
        sum = sum + this -> v[i] * y. v[i];
     return sum;
   }
```

The class template definition is similar to an ordinary class definition except the prefix template<class T> and the use of type T. This prefix tells the compiler that a template is to be declared and T is used as a type name in declarations. Thus vector has become a parameterized class with type T as its parameter. T can be substituted with any data type including the user-

defined types. Now it is possible to create vectors holding different data types.

For example

vector <int> v1(10); //10 element int vector
vector <float> v2(25); // 25 element float vector

A template class is a class created from a class template. The object of a template class is defined by using the following syntax:

classname<type> objectname(arglist);

This is a process of creating a specific class from a class template and it is called *instantiation.* The error analysis will be performed by the compiler only when instantiation takes place. Hence, it is advised to create and debug an ordinary class before converting it into a template.

The following program shows the use of a vector class template for performing the scalar product of int type vectors. #include<iostream>

```
using namespace std;
const size = 3;
template  <class T>
class vector
  {
        T* v;              //type T vector
     public:
        vector()
        {
            v = new T[size];
            for (int i=0; i<size; i++)
                v[i] = 0;
        }
        vector (T* a)
        {
           for (int i=0; i<size; i++)
                v [i] = a[i];
        }
        T operator*(vector &y)
          {
```

```
                T sum =0;
                for (int i=0; i<size; i++)
                  sum = sum + this -> v[i] * y.v[i];
                  return sum;
            }
        };

int main()
        {
                int x[3] = {1, 2, 3};
                int y[3] = {4, 5, 6};
                vector <int> v1;
                vector <int> v2;
                v1 = x;
                v2 = y;
                int R = v1 * v2;
                cout << "R =" << R << "\n";
                return 0;
        }
```

The output of the above program will be:

R = 32

The following program shows the use of a vector class template for performing the scalar product of float type vectors.

```
#include<iostream>
using namespace std;
const size = 3;
template  <class T>
class vector
  {
        T* v;               //type T vector
      public:
        vector()
         {
             v = new T[size];
              for (int i=0; i<size; i++)
```

```
            v[i] = 0;
        }
        vector (T* a)
        {
            for (int i=0; i<size; i++)
                v [i] = a[i];
        }
        T operator*(vector &y)
          {
                T sum =0;
                for (int i=0; i<size; i++)
                  sum = sum + this -> v[i] * y.v[i];
                  return sum;
           }
    };
int  main()
        {
                float x[3] = {1.1, 2.2, 3.3};
                int y[3] = {4.4, 5.5, 6.6};
                vector <float> v1;
                vector <float> v2;
                v1 = x;
                v2 = y;
                int R = v1 * v2;
                cout << "R =" << R << "\n";
                return 0;
        }
```

The output of the above program will be:

R = 38.720001

12.2.2 Class Templates with Multiple Parameters

It is possible to use more than one generic data type in a class template. They are declared as comma-separated list within the template specification as shown below:

```
template< class T1, class T2,…..>
class classname
```

```
 {
    ……
   …….
  body of the class
 ……
};
```

The program below shows the use of template class with two generic data types.

```
 #include<iostream>
 using namespace std;
 template< class T1, class T2>
class demo
  {
          T1 a;
          T2 b;
}
public:
 Test(T1 x, T2 y)
     {
            a=x;
            b=y;
     }
void show()
  {
   cout << a << " and " << b << "\n";
  }
};
int main()
    {
       demo <float, int> d1 (1.11, 567);
       demo <int, char> d2 (200, 'A');
       d1.show();
       d2.show();
       return 0;
}
```

The output of the above program will be:
1.11 and 456
200 and A

Self Assessment Questions

1. _____ enables us to define generic classes and functions and hence provides support for generic programming.
2. The class template definition is similar to an ordinary class definition except the prefix _____.

12.3 Function Templates

In C++, we can create functions that use variable types. These function templates serve as an outline or pattern for a group of functions that differ in the types of parameters they use. A group of functions that are generated from the same template is often called a family of functions. In a function template, at least one parameter is generic, or parameterized, it means that one parameter can stand for any type number of C++ types. You can write a single function template definition. Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

12.3.1 Implementing Function Templates

The template definition should have the following information:

- the keyword template
- a left angle bracket (<)
- a list of generic types, separated with comas if more than one type is needed
- a right angle bracket (>)

Each generic type in the list of generic types has two parts:

- the keyword class
- an identifier that represents the generic type

The general format of a function template is as follows:

```
template < class T>
retruntype function name (arguments of type T)
 {
   //body of function with type T wherever appropriate
}
```

The function template is similar to class template. We must use the template parameter T as and when necessary in the function body and in its argument list.

Function templates are implemented like regular functions, except they are prefixed with the keyword template. Here is a sample with a function template.

```
#include <iostream>
using namespace std;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
 {
   return a > b ? a : b;
}
```

Using the keyword class in the template definition does not necessarily mean that T stands for a programmer-created class type, but it may. Despite the keyword-class, T can represent a simple scalar type such as int. T is simply a placeholder for the type that will be used at each location in the function definition where T appears.

12.3.2 Using Template Functions

As mentioned earlier you can use the function templates as regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly, the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b) {
   return a > b ? a: b;
}
void main() {
```

```
        cout << "max(10, 15) = " << max(10, 15) << endl;
        cout << "max('k', 's') = " << max('k', 's') << endl;
        cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl;
    }
```

Output:

max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2

12.3.3 Function templates with multiple parameters

Multiple parameters are supported by function templates. We can write a function that compares three parameters and returns the largest of the three. For example, in the program code shown below the function template named findLargest(), returns the largest amongst the three parameters.

```
template <class T>

T findLargest(T  x, T  y, T  z)
 {
      T max;
      if( x > y)
            max= x;
      else
            max = y;
     if ( z> max)
            max = z;
      return max;
}
```

The three parameters are passed in the findLargest() function in the above code. A temporary variable max is declared within this function. The data type of max variable is the same as that of the function parameters. i.e., if the three integers are passed to the function, then max is also an integer; if three doubles are passed, then max is also of type double. If the first parameter x passed to the findLargest() is larger than the second parameter y, then x s assigned to max otherwise y is assigned to max. Then if the third parameter z is larger than max, then z is assigned to max. And finally value of max is returned. The variables x, y, z and max may be of any type for which the greater than (>)operator and the assignment (+) operator have

been defined, but x, y, z and max must be of the same type because they are all defined to be the same type named T.

12.3.4 Overloading Function Templates

Like ordinary functions it is possible to overload function templates. In this section, we are going to discuss overloading when the templates are involved. That is, you can have many function definitions with the same function name so that when that name is used in function call, the C++ compiler decides which of the functions to be called.

Let us see the following program to understand the overloading of function templates,

```
//maximum of two int values
int const& max (int const& a, int const& b)
 {
   return a < b ? b : a;
 }
//maximum of two values of any type
template <typename>
T const& max (T const& a, T const& b)
  {
    return a < b ? b:a;
 }
//maximum of three values of any type
template<typename T>

T const& max (T const& a, T const& b, T const& c)
  {
   return :: max (:: max (a, b), c);
 }

int main()
 {
   max(4, 5, 6);              //calls the template for three arguments
  max(8.1, 9,.1);             //calls max<double> (by argument detection)
  max ('c', 'd');             // calls max<char> (by argument detection)
  max(7,42)                   //calls the nontemplate for two ints
  max< > (7, 42);             //class max<int> (by argument deduction
```

```
  max<double>(7, 42);      // calls max<double> (no argument detection)
   max ('c', 55.5);        //calls the non-template of two ints
}
```

As you can see in this example, a non-template function can coexist with a template function that has the same name and can be instantiated with the same type. All other factors being equal, the overload resolution process normally prefers this non-template over one generated from the template. The call of the following function is under this rule.

```
max(7, 42)          //both int values match the non-template function
```

However, If the template can generate a better match, that template is selected.

This can be illustrated by the following function calls:

```
max (8.1, 9.1)          // calls the max<double> (by argument detection)
max ('c', 'd')          //class the max<char> (by argument detection)
```

It is also possible to specify explicitly an empty template argument list. This

syntax indicates that only templates may resolve a call, but all the template parameters should be deduced from the call arguments:

```
max<>(7,42)          //calls max <int> (by argument detection).
```

In the function call max ('c', 55.5) both the arguments will be converted into int by automatic type conversion. Because the automatic type conversion is not considered for templates but is considered for ordinary functions.

Self Assessment Questions

3. The _____ serves as an outline or pattern for a group of functions that differs in the types of parameters they use.
4. Like ordinary functions it is possible to overload function templates. (True/ False).

12.4 Template Instantiation

When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.

- When a class is generated from a class template, it is called a generated class.

- When a function is generated from a function template, it is called a generated function.
- When a static data member is generated from a static data member template, it is called a generated static data member.

The compiler generates a class, function or static data members from a template when it sees an implicit instantiation or an explicit instantiation of the template.

1. Consider the following sample. This is an example of implicit instantiation of a class template.

```
template <class T>
class Z
  {
   public:
      Z() {};
      ~Z() {};
      void f(){};
      void g(){};
};
int main() {
   Z<int> zi;   //implicit instantiation generates class Z<int>
   Z<float> zf; //implicit instantiation generates class Z<float>
   return 0;
}
```

2. Consider the following sample. This sample uses the template class members Z<T>::f() and Z<T>::g().

```
template <class T>
class Z {
   public:
      Z() {};
      ~Z() {};
      void f(){};
      void g(){};
};
int main() {
   Z<int> zi; //implicit instantiation generates class Z<int>
```

```
    zi.f();    //and generates function Z<int>::f()
    Z<float> zf; //implicit instantiation generates class Z<float>
    zf.g();      //and generates function Z<float>::g()
    return 0;
}
```

This time in addition to the generating classes Z<int> and Z<float>, with constructors and destructors, the compiler also generates definitions for Z<int>::f() and Z<float>::g(). The compiler does not generate definitions for functions, nonvirtual member functions, class or member class that does not require instantiation. In this example, the compiler did not generate any definitions for Z<int>::g() and Z<float>::f(), since they were not required.

3. Consider the following sample. This is an example of explicit instantiation of a class template.

```
template <class T>
class Z {
public:
    Z() {};
    ~Z() {};
    void f(){};
    void g(){};
};
int main() {
    template class Z<int>; //explicit instantiation of class Z<int>
    template class Z<float>; //explicit instantiation of class Z<float>
    return 0;
}
```

4. Consider the following sample. Will the compiler generate any classes in this case? The answer is NO.

```
template <class T>
class Z {
    public:
        Z() {};
        ~Z() {};
        void f(){};
```

```
        void g(){};
};
int main() {
    Z<int>* p_zi; //instantiation of class Z<int> not required
    Z<float>* p_zf; //instantiation of class Z<float> not required
    return 0;
}
```

This time the compiler does not generate any definitions! There is no need for any definitions. It is similar to declaring a pointer to an undefined class or structure.

5. Consider the following sample. This is an example of implicit instantiation of a function template.

```
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
    return a > b ? a : b;
}
void main() {
    int I;
    I = max(10, 15); //implicit instantiation of max(int, int)
    char c;
    c = max('k', 's'); //implicit instantiation of max(char, char)
}
```

In this case the compiler generates functions max(int, int) and max(char, char). The compiler generates definitions using the template function max.

6. Consider the following sample. This is an example of explicit instantiation of a function template.

```
template <class T>
void Test(T r_t) {
}
int main() {
    //explicit instantiation of Test(int)
    template void Test<int>(int);
```

```
    return 0;
}
```

In this case the compiler would generate function Test(int). The compiler generates the definition using the template function Test.

7. If an instantiation of a class template is required, and the template declared but not defined, the program is ill-formed.

```
template <class T> class X ;
int main() {
    X<int> xi;
    return 0;
}
```

8. Instantiating virtual member functions of a class template that does not require instantiation is implementation defined. For example, in the following sample, virtual function X<T>::Test() is not required, VC5.0 generates a definition for X<T>::Test.

```
template <class T>
class X {
    public:
        virtual void Test() {}
};
int main() {
    X<int> xi; //implicit instantiation of X<int>
    return 0;
}
```

In this case the compiler generates a definition for X<int>::Test, even if it is not required.

Self Assessment Questions

5. A class generated from a class template is called _____.
6. Instantiating virtual member functions of a class template that does not require instantiation is implementation defined. (True/False)

12.5 Class Template Specialization

In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template

specialization. The following example defines a template class specialization for template class stream.

```
#include <iostream>
using namespace std;
template <class T>
class stream
{
    public:
        void f()
   { cout << "stream<T>::f()"<< endl;}
};
template <>
class stream<char>
{
    public:
        void f() { cout << "stream<char>::f()"<< endl; }
};
int main()
{
    stream<int> si;
    stream<char> sc;
    si.f();
    sc.f();
    return 0;
}
```

Output:

```
stream<T>::f()
stream<char>::f()
```

In the above example, stream<char> is used as the definition of streams of chars; other streams will be handled by the template class generated from the class template.

12.5.1 Template Class Partial Specialization

You may want to generate a specialization of the class for just one parameter, for example

```
//base template class
```

```
template<typename T1, typename T2>
class X {
};
//partial specialization
template<typename T1>
class X<T1, int> {
};
int main() {
   // generates an instantiation from the base template
   X<char, char> xcc ;
   //generates an instantiation from the partial specialization
   X<char, int> xii ;
   return 0 ;
}
```

A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list.

12.6 Template Function Specialization

As already mentioned in sub-unit 12.5, in some cases it is possible to override the template-generated code by providing special definitions for specific types using template specialization. The following example demonstrates a situation where overriding the template generated code would be necessary:

```
#include <iostream>
using namespace std;
//max returns the maximum of the two elements of type T,
//where T is a class or data type for which operator> is defined.
template <class T>
T max(T a, T b)
{
   return a > b ? a : b;
}
int main() {
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
```

```
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
    cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin",
"Jasmine") << endl ;
    return 0 ;
}
```

Output:

max(10, 15) = 15

max('k', 's') = s

max(10.1, 15.2) = 15.2

max("Aladdin", "Jasmine") = Aladdin

Not quite the expected results! Why did that happen? The function call max("Aladdin", "Jasmine") causes the compiler to generate code for max(char*, char*), which compares the addresses of the strings! One can use template specializations to correct special cases like these or to provide more efficient implementations for certain types. The above example can be rewritten with specialization as follows:

```
#include <iostream>
#include <cstring>
using namespace std;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b) {
   return a > b ? a : b ;
}
// Specialization of max for char*
template <>
char* max(char* a, char* b) {
   return strcmp(a, b) > 0 ? a : b ;
}
int main() {
   cout << "max(10, 15) = " << max(10, 15) << endl ;
   cout << "max('k', 's') = " << max('k', 's') << endl ;
   cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
```

```
   cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine")
<< endl ;
   return 0 ;
}
```

Output:

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
max("Aladdin", "Jasmine") = Jasmine
```

Self Assessment Questions

7. _____ is to override the template-generated code by providing special definitions for specific types.

12.7 Template Parameters

1. C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced by actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>. For example,

   ```
   template <class T>
   class Stack{
   };
   ```

   Here T is a template parameter, also referred to as type-parameter.

2. In C++ it is allowed to specify a default template parameter, so the definition could now look like:

   ```
   template <class T = float, int elements = 100> Stack { ....};
   ```

   Then a declaration such as

   ```
   Stack<> mostRecentSalesFigures;
   ```

   would instantiate (at compile time) a 100 element Stack template class named mostRecentSalesFigures of float values; this template class would be of type Stack<float, 100>.

   Note, C++ also allows non-type template parameters. In this case, template class Stack has an int as a non-type parameter. Once a default parameter is declared all subsequent parameters must have defaults.

3. Default arguments cannot be specified in a declaration or a definition of a specialization. For example,

```
template <class T, int size>
class Stack {
};
//defined as a non-template class
template <class T, int size = 10>
class Stack<int, 10> {
};
int main() {
   Stack<float,10> si;
            return 0;
}
```

4. A type-parameter defines its identifier to be a type-name in the scope of the template declaration, and cannot be re-declared within its scope (including nested scopes). For example,

```
template <class T, int size>
class Stack {
    int T;
    void f()
    {
        char T; //error type-parameter re-defined.
    }
};
class A {};
int main() {
    Stack<A,10> si;
    return 0;
}
```

5. The value of a non-type-parameter cannot be assigned to or have its value changed. For example,

```
template <class T, int size>
class Stack {
```

```
    void f()
    {
        size++ ; //error change of template argument value
    }
};
int main() {
    Stack<double,10> si ;
    return 0 ;
}
```

6. A template-parameter that could be interpreted as either a parameter-declaration or a type-parameter, is taken as a type-parameter. For example,

```
class T {};
int i;
template <class T, T i>
void f(T t) {
    T t1 = i; //template arguments T and i
    ::T t2 = ::i; //globals T and i
}
int main() {
    f('s');
    return 0 ;
}
```

Self Assessment Questions

8. C++ allows you to specify a default template parameter. (True/False)
9. _____ cannot be specified in a declaration or a definition of a specialization.

12.8 Static Members and Variables

1. Each template class or function generated from a template has its own copies of any static variables or members.
2. Each instantiation of a function template has its own copy of any static variables defined within the scope of the function. For example,

```
template <class T>
class X {
```

```
    public:
        static T s;
};
int main() {
    X<int> xi ;
    X<char*> xc ;
}
```

Here X<int> has a static data member s of type int and X<char*> has a static data member s of type char*.

3. Static members are defined as follows:

```
#include <iostream>
using namespace std;
template <class T>
class X {
    public:
        static T s;
};
template <class T> T X<T>::s = 0 ;
template <> int X<int>::s = 3 ;
template <> char* X<char*>::s = "Hello" ;
int main() {
    X<int> xi ;
    cout << "xi.s = " << xi.s << endl ;
    X<char*> xc ;
    cout << "xc.s = " << xc.s << endl ;
    return 0 ;
}
```

Program Output:

```
xi.s = 10
xc.s = Hello
```

4. Each instantiation of a function template has its own copy of the static variable. For example,

```
#include <iostream>
using namespace std;
template <class T>
void f(T t) {
   static T s  = 0;
   s = t;
   cout << "s = " << s << endl;
}
int main() {
   f(10);
   f("Hello");
   return 0;
}
```

Program Output:

s = 10
s = Hello

Here f<int>(int) has a static variable s of type int, and f<char*>(char*) has a static variable s of type char*.

12.9 Templates and Friends

We have already studied that functions and classes can be declared as friends of non-template classes. With class templates friendship can be established between a class template and a global function, a member function of another class (possibly a template class), or even an entire class (possible template class). Throughout this section we assume that we have defined a class template for a class named x with a single type parameter T, as shown below:

template <typename T> class X

The table 12.1 lists the results of declaring different kinds of friends of a class.

Table 12.1: Different kinds of friends of a class

Class Template	**friend declaration in class template X**	**Results of giving friendship**
template class <T> class X	friend void f1();	makes f1() a friend of all instantiations of template X. For example, f1() is a friend of X<int>, X<A>, and X<Y>.
template class <T> class X	friend void f2(X<T>&);	For a particular type T for example, float, makes f2(X<float>&) a friend of class X<float> only. f2(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend A::f4(); // A is a user defined class with a member function f4();	makes A::f4() a friend of all instantiations of template X. For example, A::f4() is a friend of X<int>, X<A>, and X<Y>.
template class <T> class X	friend C<T>::f5(X<T>&); // C is a class template with a member function f5	For a particular type T for example, float, makes C<float>::f5(X<float>&) a friend of class X<float> only. C<float>::f5(x<float>&) cannot be a friend of class X<A>.
template class <T> class X	friend class Y;	makes every member function of class Y a friend of every template class produced from the class template X.
template class <T> class X	friend class Z<T>;	when a template class is instantiated with a particular type T, such as a float, all members of class Z<float> become friends of template class X<float>.

12.10 Templates and Multiple-file Projects

From the compiler's point of view, templates are not considered as normal functions or classes. They are compiled on demand. It means that the code of a template function is not compiled until an instantiation with specific template arguments is required. The moment when a need for an instantiation arises, a function is generated by the compiler which is specifically for those arguments from the template

When the project expands, the code of program is distributed in many source code files. In such situations the interface and implementations are separated. For example in library functions the interface contains the declaration of prototypes of all the functions that can be called. These functions are declared in a header file having .h extension and the definitions of functions i.e. implementation is contained in a different file with C++ code.

Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that the interface cannot be separated in different header file. And it is required to include both the interface and implementation in any file using the templates.

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.

Self Assessment Questions

10. Each template class or function generated from a template has its own copies of any static variables or members. (True/False)
11. Templates are compiled when required. (True/False)
12. With _____, friendship can be established between a class template and a global function, a member function of another class (possibly a template class), or even an entire class.

12.11 Summary

- Templates are a fairly new addition to the C++ language, and were only recently standardized on. They are also one of the more useful features of C++. They allow you to create classes that are more dynamic in terms of the types of data they can handle.
- A class template is a class that is implemented with one or more type parameters left open. The class template definition is similar to an ordinary class definition except the prefix template<class T> and the use of type T.
- This is a process of creating a specific class from a class template and it is called as instantiation.
- It is possible to use more than one generic data type in a class template
- In C++ we can create functions that use variable types. These function templates serve as an outline or pattern for a group of functions that differ in the types of parameters they use.
- Function templates are implemented like regular functions, except they are prefixed with the keyword template.
- Multiple parameters are supported by function templates.
- Like ordinary functions it is possible to overload function templates.
- When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.
- In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization.
- Each template class or function generated from a template has its own copies of any static variables or members.
- With class templates, friendship can be established between a class template and a global function, a member function of another class or even an entire class.
- Compiling templates when required forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration

12.12 Terminal Questions

1. Explain with the help of an example the method to implement a class template.
2. Explain the implementation of function template with multiple parameters and procedure to overload function templates.
3. Discuss template function specialization.
4. Write a note on template parameters.
5. How static members are declared within a template class? Explain with an example.
6. Describe the various types of template friendships.

12.13 Answers

Self Assessment Question

1. Templates
2. template<class T>
3. Function templates
4. True
5. Generated class
6. False
7. Template specialization
8. True
9. Default arguments
10. True
11. True
12. Class templates

Terminal Questions

1. It is a simple process to create a generic class using a template with anonymous type. The general format of class template is as follows:

```
class classname
 {
    //class member specification with anonymous type T wherever
     appropriate.
  …….
};
```

For more details refer section 12.2.1

2. Multiple parameters are supported by function templates. We can write a function that compares three parameters and returns the largest of the three. For example, in the program code shown below the function template named findLargest(), returns the largest amongst the three parameters.

   Like ordinary functions, it is possible to overload function templates.

   That means you can have many function definitions with the same function name so that when that name is used in function call, it is decided by the C++ compiler that which one of the functions to be called.

   For more details refer sections 12.3.3 and 12.3.4.

3. In some cases it is possible to override the template-generated code by providing special definitions for specific types. This is called template specialization.
   For more details refer section 12.6
4. C++ templates allow one to implement a generic Queue<T> template that has a type parameter T. T can be replaced with actual types, for example, Queue<Customers>, and C++ will generate the class Queue<Customers>.
   For example,

```
template <class T>
 class Stack{
 };
```

   For more details refer section 12.7.

5. Each template class or function generated from a template has its own copies of any static variables or members. Each instantiation of a function template has its own copy of any static variables defined within the scope of the function.
   For more details refer section 12.8
6. Friendship can be established between a class template and a global function, a member function of another class, or even an entire class.
   For more details refer section 12.9.

References:

- Object Oriented Programming with C++ - Sixth Edition, by E Balagurusamy. Tata McGraw-Hill Education.
- Object Oriented Programming In C++, 4/E by Robert Lafore. Pearson Education India.
- C++ Templates: The Complete Guide, By David Vandevoorde, Nicolai M. Josuttis. Addison Wesley Professional.
- C++ for Programmers, By Paul Deitel, Harvey M. Deitel. Pearson Education.