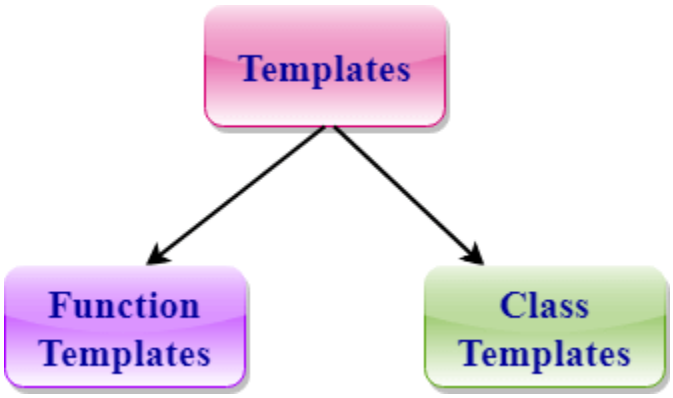
C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.

- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
    // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
{
    T result = a+b;
    return result;

}
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
```

```
15.  cout<<"Addition of i and j is :"<<add(i,j);
16.  cout<<'\n';
17.  cout<<"Addition of m and n is :"<<add(m,n);
18.  return 0;
19. }
```

Output:

```
Addition of i and j is :5
Addition of m and n is :3.5
```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
1.  template<class T1, class T2,.....>
2. return_type function_name (arguments of type T1, T2....)
3. {
4.     // body of function.
5. }
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. template<class X,class Y> void fun(X a,Y b)
4. {
5.     std::cout << "Value of a is : " <<a<< std::endl;
6.     std::cout << "Value of b is : " <<b<< std::endl;
7. }
```

```
8.
9.  int main()
10. {
11.    fun(15,12.3);
12.
13.    return 0;
14. }
```

Output:

```
Value of a is : 15
Value of b is : 12.3
```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
1.  #include <iostream>
2.  using namespace std;
3.  template<class X> void fun(X a)
4.  {
5.      std::cout << "Value of a is : " <<a<< std::endl;
6.  }
7.  template<class X,class Y> void fun(X b ,Y c)
8.  {
9.      std::cout << "Value of b is : " <<b<< std::endl;
10.     std::cout << "Value of c is : " <<c<< std::endl;
11. }
12. int main()
13. {
14.    fun(10);
15.    fun(20,30.5);
```

```
16.    return 0;
17. }
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:

```
1.  #include <iostream>
2.  using namespace std;
3.  void fun(double a)
4.  {
5.      cout<<"value of a is : "<<a<<'\n';
6.  }
7.
8.  void fun(int b)
9.  {
10.     if(b%2==0)
11.     {
12.         cout<<"Number is even";
13.     }
14.     else
15.     {
16.         cout<<"Number is odd";
17.     }
18.
19. }
```

```
20.
21. int main()
22. {
23.    fun(4.6);
24.    fun(6);
25.    return 0;
26. }
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
1. template<class Ttype>
2. class class_name
3. {
4.   .
5.   .
6. }
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

1. class_name<type> ob;

   where class_name: It is the name of the class.

   type: It is the type of the data that the class is operating on.

   ob: It is the name of the object.

   Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

   Output:

```
Addition of num1 and num2 : 11
```

   In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
   // Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
       T1 a;
       T2 b;
       public:
      A(T1 x,T2 y)
     {
        a = x;
        b = y;
     }
        void display()
       {
             std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
    };

    int main()
   {
```

```
22.         A<int,float> d(5,6.5);
23.         d.display();
24.         return 0;
25.     }
```

Output:

```
Values of a and b are : 5,6.5
```

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. **Let' s see the following example:**

```
1. template<class T, int size>
2. class array
3. {
4.         T arr[size];          // automatic array initialization.
5. };
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
1. array<int, 15> t1;                 // array of 15 integers.
2. array<float, 10> t2;               // array of 10 floats.
3. array<char, 4> t3;                 // array of 4 chars.
```

Let's see a simple example of nontype template arguments.

```
1. #include <iostream>
2. using namespace std;
3. template<class T, int size>
4. class A
5. {
```

```
6.      public:
7.      T arr[size];
8.      void insert()
9.      {
10.        int i =1;
11.        for (int j=0;j<size;j++)
12.        {
13.            arr[j] = i;
14.            i++;
15.        }
16.     }
17.
18.     void display()
19.     {
20.        for(int i=0;i<size;i++)
21.        {
22.            std::cout << arr[i] << " ";
23.        }
24.     }
25. };
26. int main()
27. {
28.     A<int,10> t1;
29.     t1.insert();
30.     t1.display();
31.     return 0;
32. }
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.