Background:

In C++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. For example:

```
int max ( int x, int y)
{
    return x > y ? x: y;
}

char max (char x, char y)
{
    return x > y ? x: y;
}

double max (double x, double y)
{
    return x > y ? x: y;
}

float max (float x, float y)
{
    return x > y ? x: y;
}
```

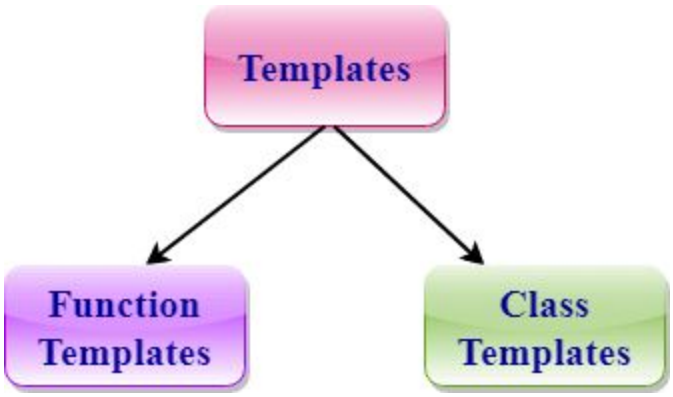
Here we can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided by using template provided by C++.

Introduction:

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:

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 data passed as a parameter.

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.

Syntax:

```
template < class T_type>
 return_type function_name(parameter_list)
{
   // body of function.
}
```

Example:

```
template<class T>
T add(T a , T b)
{
   //defintion
}
```

Example: Finding sum of two numbers with different data types.

```
#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;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is :"<<add(m,n);
  return 0;
}
```

Output:

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

Example of function template to find maximum values

```cpp
#include <iostream>
using namespace std;

template <typename T>
T myMax(T x, T y)
{
   return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```

Output:

7

7

g

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

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

Example:

```
#include <iostream>
using namespace std;
template<class X,class Y>
void fun(X a,Y b)
{
    cout << "Value of a is : " <<a<< endl;
    cout << "Value of b is : " <<b<< endl;
}

int main()
{
   fun(15,12.3);

   return 0;
}
```

Output:

Value of a is : 15

Value of b is : 12.3

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

template<class T_type>
class class_name
{
 .
 .
}

Now, we create an instance of a class

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.

Example: Program to add two numbers

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

};

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

Output:

Addition of num1 and num2 : 11

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.
}

Example:

```
#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()
       {
           cout << "Values of a and b are : " << a<<" ,"<<b<<endl;
       }
 };

 int main()
{
       A<int,float> d(5,6.5);
       d.display();
       return 0;
}
```

Output:

Values of a and b are : 5,6.5

Advantages of C++ Class Templates:

- One C++ Class Template can handle different types of parameters.
- Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.