Template

- A **template** is a simple yet very powerful tool in C++.
- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- A template is a blueprint or formula for creating a generic class or a function.
- Using C++ templates, you can create a group of classes or functions that can handle various forms of data.
- When there is a need to duplicate the same code across many types, we use templates.
- The **simple idea is to pass data type as a parameter** so that we don't need to write the same code for different data types.
- For example, a software company may need to add() for different data types. Rather than writing and maintaining multiple codes, we can write one add() and pass data type as a parameter.
- There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.
- You can use templates to define functions as well as classes.

1. **Function Template**
2. **Class Template**

Function Template

When a function uses the concept of Template, then the function is known as generic function.

- ✓ The general **Syntax** of a template function definition is shown here –

```
template <class type>
return-type func-name(parameter list)
  {
      // body of function
   }
```

Example:

```
template <class T>             // Function Template
void show(T a, T b)
{
        // Body of Function Template
}
```

```
template<class T>       // Function Template
void show(T a, T b)
{
  cout<<"A= "<<a<<endl;
  cout<<"A= "<<b<<endl;
}
void main()
{
int p=10,q=20;
char m='a', n='b';
float s=10.50, f=12.56;
clrscr();

show(p,q);
show(m,n);
show(s,f);
getch();
}
```

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>

class class_name
{
  --------
  --------
}
```

Example::::

```
template<class T>
class A
{
  T no1=50;
  -------
  --------
}
```

- ✓ **T** 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 T can be used inside the class body.

Now, we create an **Object of a class**

class_name<type> ob;

Example:

A<int> **ob1;** **// Object**

Example of Template Class

```
template <class T>

class show
{
      T a, b;
   public:
      show(T x, T y)
      {
          a=x;
          b=y;
      }

      void show()
      {
         cout<<"A=" <<a<< endl;
         cout<<"B=" <<b<< endl;
      }
};
```

```
void main()
{
        show <int> ob1 (10,20);

        clrscr();

        ob1.show();

        getch();

}
```

Using Class Template with Multiple Parameters in C++

```
template <class T1, class T2>

class Test
{
        T1 a;
        T2 b;
    public:
        Test(T1 x, T2 y)
        {
            a=x;
            b=y;
        }

        void show()
        {
          cout<<"A=" <<a<< endl;
          cout<<"B=" <<b<< endl;
        }
};
```

```
void main()
{
Test <int,char> ob1 (10,20);

Test <float,int> ob2 (5.2,7);

clrscr();

ob1.show();

ob2.show();

getch();

}
```

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 built-in or derived data types as template arguments.

Difference between function overloading and templates in C++?

Function overloading	Function Template
This is used when multiple functions do similar operations.	This is used when functions do identical operations.
Function overloading can take varying numbers of arguments.	Templates cannot take varying numbers of arguments.

What is a template and what are its advantages?

Using C++ templates, you can create a group of classes or functions that can handle various forms of data.

When there is a need to duplicate the same code across many types, we use templates.

Several advantages of templates are as follows:

- ✓ They increase the efficiency of the program by reducing the developing-time when used in combination with STL.
- ✓ They permit type generalization.
- ✓ They reduce the quantity of repetitive code you must type.
- ✓ They assist in writing type-safe code.
- ✓ They aid in creating extremely powerful libraries
- ✓ Templates are type-safe.
- ✓ They are generally considered as an improvement over macros for these purposes.
- ✓ Templates avoid some common errors found in code that make heavy use of function-like macros.
- ✓ Both templates and macros are expanded at compile time.
- ✓ They are a good way of making generalizations for APIs.

Disadvantages of Using Templates in C++

- Many compilers do not support nesting of templates.
- When templates are used, all codes exposed.
- Some compilers have poor support of templates.
- Approx all compilers produce unhelpful, confusing error messages when errors are detected in the template code.
- It can make it challenging to develop the template.

How many templates are there in CPP?

- ✓ As of the latest version, CPP14, there are three main templates, namely function, class , and variable templates.

Nested class templates

- ✓ Templates can be defined within classes or class templates, in which case they're referred to as member templates.
- ✓ Member templates that are classes are referred to as nested class templates.
- ✓ Member templates that are functions are discussed in Member Function Templates.
- ✓ Nested class templates are declared as class templates inside the scope of the outer class.
- ✓ They can be defined inside or outside of the enclosing class.

Example

```
#include <iostream>

using namespace std;

template <class T>
class X
{
   template <class U> class Y
   {
      U* u;
   public:
      Y();
      U& Value();
      void print();
      ~Y();
   };

   Y<int> y;
public:
   X(T t) { y.Value() = t; }
   void print() { y.print(); }
};
```

```
template <class T>
template <class U>

X<T>::Y<U>::Y()
{
   cout << "X<T>::Y<U>::Y()" <<
endl;
   u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
   return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
   cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
   cout << "X<T>::Y<U>::~Y()" <<
endl;
   delete u;
}
```

```cpp
int main()
{
   X<int>* xi = new X<int>(10);
   X<char>* xc = new X<char>('c');
   xi->print();
   xc->print();
   delete xi;
   delete xc;
}
```

```
Output:

X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()
X<T>::Y<U>::~Y()
```