Chapter 9 : Templates

Introduction to Template

A template is a simple and yet very powerful tool in C++. 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 sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: 'template' and 'typename'. The second keyword can always be replaced by keyword 'class'.

How do templates work?

Templates are expanded at compiler time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

Class Templates

Class Templates Like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, Binary Tree, Stack, Queue, Array, etc.
Following is a simple example of template Array class.

```
#include <iostream>

using namespace std;

template <typename T>

class Array {
```

```
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};
template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
```

```
 for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}
template <typename T>
void Array<T>::print() {
for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}
 int main() {
```

```
    int arr[5] = {1, 2, 3, 4, 5};
Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Output :

1 2 3 4 5

Function Templates

Function Templates We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().
Know more on Generics in C++

For Example :-

```
#include <iostream>

using namespace std;

// One function works for all data types.  This would work
```

```
// even for user defined types if operator '>' is overloaded

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
```

Overloading of Function Templates with Multiple Parameters

The name of the function templates are the same but called with different arguments is known as function template overloading.

If the function template is with the ordinary template, the name of the function remains the same but the number of parameters differs.

When a function template is overloaded with a non-template function, the function name remains the same but the function's arguments are unlike.

Below is the program to illustrate overloading of template function using an explicit function:

// Write a C++ program to illustrate overloading of template function using an explicit function.

```
#include <bits/stdc++.h>

using namespace std;

template <class T> // Template declaration

void display(T t1)  // Template overloading of function

{

   cout << "Displaying Template :  " << t1 << "\n";  }

void display(int t1) // Template overloading of function

{
```

```
cout << "Explicitly display: "
        << t1 << "\n";  }
int main()  // Driver Code
{      // Function Call with a different arguments
    display(200);
    display(12.40);
    display('G');
    return 0;  }
```