C++ Templates

What is a template in C++ programming?

A template is believed to escalate the potential of C++ several fold by giving it the ability to define data types as parameters making it useful to reduce repetitions of the same declaration of classes for different data types. Declaring classes for every other data type(which if counted is way too much) in the very first place violates the DRY(Don't Repeat Yourself) rule of programming and on the other doesn't completely utilise the potential of C++.

It is very analogous to when we said classes are the templates for objects, here templates itself are the templates of the classes. That is, what classes are for objects, templates are for classes.

Why templates?

DRY Rule:

To understand the reason behind using templates, we will have to understand the effort behind declaring classes for different data types. Suppose we want to have a vector for each of the three(can be more) data types, int, float and char. Then we'll obviously write the whole thing again and again making it awfully difficult. This is where the saviour comes, the templates. It helps parametrizing the data type and declaring it once in the source code suffice. Very similar to what we do in functions. It is because of this, also called, 'parameterized classes'.

Generic Programming:

It is called generic, because it is sufficient to declare a template once, it becomes general and it works all along for all the data types.

```
#include <iostream>

using namespace std;


template <class T>

class vector {

   T *arr;

   int size;

   public:

      vector(T* arr)[

         //code

      ]

      //and many other methods


};
```

```
int main() {
   vector<int> myVec1();
   vector<float> myVec2();
   return 0;
}
```

Writing our First C++ Template

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

class vector
{
   public:
      int *arr;
      int size;
      vector(int m)
      {
         size = m;
         arr = new int[size];
      }
   int dotProduct(vector &v){
      int d=0;
      for (int i = 0; i < size; i++)
      {
         d+=this->arr[i]*v.arr[i];
      }
      return d;
   }
};

int main()
```

```
{
   vector v1(3); //vector 1
   v1.arr[0] = 4;
   v1.arr[1] = 3;
   v1.arr[2] = 1;
   vector v2(3); //vector 2
   v2.arr[0]=1;
   v2.arr[1]=0;
   v2.arr[2]=1;
   int a = v1.dotProduct(v2);
   cout<<a<<endl;
   return 0;
}
```

Understanding the changes, we made in the above program to generalise it for all data types:

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

template <class T>
class vector
{
   public:
      T *arr;
      int size;
      vector(int m)
      {
         size = m;
         arr = new T[size];
      }
   T dotProduct(vector &v){
      T d=0;
      for (int i = 0; i < size; i++)
```

```
        {
            d+=this->arr[i]*v.arr[i];
        }
        return d;
    }
};

int main()
{
    vector<float> v1(3); //vector 1 with a float data type
    v1.arr[0] = 1.4;
    v1.arr[1] = 3.3;
    v1.arr[2] = 0.1;
    vector<float> v2(3); //vector 2 with a float data type
    v2.arr[0]=0.1;
    v2.arr[1]=1.90;
    v2.arr[2]=4.1;
    float a = v1.dotProduct(v2);
    cout<<a<<endl;
    return 0;
}
```

Templates with Multiple Parameters

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

/*
template<class T1, class T2>
class nameOfClass{
    //body
}
*/
```

```
int main(){
   //body of main
}
```

Syntax of a template with multiple parameter

```
class myClass{
   public:
      int data1;
      char data2;
   void display(){
      cout<<this->data1<<" "<<this->data2;
   }
};
```

Constructing a class

```
template<class T1, class T2>
class myClass{
   public:
      T1 data1;
      T2 data2;
      myClass(T1 a,T2 b){
         data1 = a;
         data2 = b;
      }
   void display(){
      cout<<this->data1<<" "<<this->data2;
   }
};
```

Constructing a template with two parameters.

```
int main()
{
   myClass<int, char> obj(1, 'c');
   obj.display();
}
```

Output:

```
1   c
PS D:\MyData\Business\code playground\C++ course>
```

```
int main()
{
   myClass<int, float> obj(1,1.8 );
   obj.display();
}
```

Output:

```
1   1.8
PS D:\MyData\Business\code playground\C++ course>
```

Class Templates with Default Parameters

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

template <class T1=int, class T2=float, class T3=char>
class Harry{
   public:
      T1 a;
      T2 b;
      T3 c;
      Harry(T1 x, T2 y, T3 z) {
         a = x;
         b = y;
         c = z;
      }
```

```
    void display(){
        cout<<"The value of a is "<<a<<endl;
        cout<<"The value of b is "<<b<<endl;
        cout<<"The value of c is "<<c<<endl;
    }
};
int main()
{
   Harry<> h(4, 6.4, 'c');
   h.display();
   cout << endl;
   Harry<float, char, char> g(1.6, 'o', 'c');
   g.display();
   return 0;
}
```

Output:

```
The value of a is 4
The value of b is 6.4
The value of c is c

The value of a is 1.6
The value of b is o
The value of c is c
PS D:\MyData\Business\code playground\C++ course>
```

C++ Function Templates & Function Templates with Parameters

Suppose we want to have a function which calculates the average of two integers.

```
#include<iostream>
using namespace std;
 float funcAverage(int a, int b){
   float avg= (a+b)/2.0;
   return avg;
```

```
}
int main(){
   float a;
   a = funcAverage(5,2);
   printf("The average of these numbers is %f",a);
   return 0;
}
```

Output:

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

Refer to the snippet below.

```
template<class T1, class T2>
float funcAverage(T1 a, T2 b){
   float avg= (a+b)/2.0;
   return avg;
}
int main(){
   float a;
   a = funcAverage(5,2);
   printf("The average of these numbers is %f",a);
   return 0;
}
```

Output:

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
int main(){
   float a;
   a = funcAverage(5,2.8);
   printf("The average of these numbers is %f",a);
   return 0;
}
```

Output:

The average of these numbers is 3.900000

PS D:\MyData\Business\code playground\C++ course>

Member Function Templates & Overloading Template Functions in C++

```
template <class T>
class Harry
{
public:
   T data;
   Harry(T a)
   {
      data = a;
   }
   void display();
};

template <class T>
void Harry<T> :: display(){
   cout<<data;
}
int main()
{
   Harry<int> h(5.7);
   cout << h.data << endl;
   h.display();
   return 0;
}
```

Output:

5

5

PS D:\MyData\Business\code playground\C++ course>

overloading of a function template.

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


void func(int a){
   cout<<"I am first func() "<<a<<endl;
}


template<class T>
void func(T a){
   cout<<"I am templatised func() "<<a<<endl;
}
int main()
{
   func(4);  //Exact match takes the highest priority
   return 0;
}
```

Output:

```
I am first func() 4
PS D:\MyData\Business\code playground\C++ course>
```

If we hadn't created the first function with int data type, the call would have gone to the templatised func only because a template function is an exact match for every kind of data type.