Data Structures and Object Oriented Programming using C++

Ahsan Ijaz

Templates

- Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

Function Overloading using cout

```
1  void PrintTwice(int data)
2  {
3      cout << "Twice is: " << data * 2 << endl;
4  }
5
6  void PrintTwice(double data)
7  {
8      cout << "Twice is: " << data * 2 << endl;
9  }
```

Function Overloading using printf

```
1  void PrintTwice(int data)
2  {
3      printf("Twice is: %d", data * 2 );
4  }
5
6  void PrintTwice(double data)
7  {
8      printf("Twice is: %lf", data * 2 );
9  }
```

Templates

- Class type **ostream** (the type of cout object) has multiple overloads for operator $<<$ for all basic data-types. Therefore, same/similar code works for both int and double, and no change is required for our **PrintTwice**.

Types of Templates

- Function Templates
- Class Templates

Without template, you would need to replicate same code all over again and again, for all required data-types.

Function Template Example

```
1  template<class TYPE>
2  void PrintTwice(TYPE data)
3  {
4      cout<<"Twice: " << data * 2 << endl;
5  }
```

template<class TYPE>
tells the compiler that this is a function-template. The actual meaning of TYPE would be deduced by compiler depending on the argument passed to this function. Here, the name, TYPE is known as template type parameter.

Compiler Instantiate Appropriate Function

- On calling **PrintTwice(124)**:
  - TYPE would be replaced by compiler as int.
- If call is made to **PrintTwice(4.5547)**:
  - TYPE would be replaced by compiler as double automatically.

Without template, you would need to replicate same code all over again and again, for all required data-types.

Compiler does the overloading

It means, in your program, if you call PrintTwice function with int and double parameter types, two instances of this function would be generated by compiler:

```
1 void PrintTwice(int data) { ... }
2
3 void PrintTwice(double data) { ... }
```

Return type Example

```
1  template<typename TYPE>
2  TYPE Twice(TYPE data)
3  {
4      return data * 2;
5  }
```

Return type Example

When we call:

```
1  cout << Twice(10);
2  cout << Twice(3.14);
```

Following set of functions would be generated:

```
1  int      Twice(int data) {..}
2  double   Twice(double data) {..}
```

Addition Example

When we call:

```
1  template<class T>
2  T Add(T n1, T n2)
3  {
4      T result;
5      result = n1 + n2;
6
7      return result;
8  }
```

Multiple Types with Function Templates

When we call:

```
1  template<class T1, class T2>
2  void PrintNumbers(const T1& t1Data, const T2& t2Dat
3  {
4          cout << "First value:" << t1Data;
5          cout << "Second value:" << t2Data;
6  }
```

Multiple Types with Function Templates

On calling these functions:

```
1 PrintNumbers(10, 100);     // int, int
2 PrintNumbers(14, 14.5);    // int, double
3 PrintNumbers(59.66, 150); // double, int
```

Compiler translates:

```
1 // const and reference removed for simplicity
2 void PrintNumbers(int t1Data, int t2Data);
3 void PrintNumbers(int t1Data, double t2Data);
4 void PrintNumbers(double t1Data, int t2Data);
```

Explicit type specification in Templates

```
1  PrintNumbers<double, double>(10, 100);     // int, i
2  PrintNumbers<double, double>(14, 14.5);    // int, d
3  PrintNumbers<double, double>(59.66, 150); // double
```

Class Templates

A class template is used to define an abstract type whose behavior is generic and is reusable, adaptable.

Simple Class Example

A simple example of a Class:

```
1  class Item
2  {
3      int Data;
4  public:
5      Item() : Data(0)
6      {}
7
8      void SetData(int nValue)
9      {
10         Data = nValue;
11     }
12
13     int GetData() const
14     {
15         return Data;
16     }
17
18     void PrintData()
19     {
20         cout << Data;
21     }
22 };
```

What if you need flexibility in class definition??

Simple enough, right?? But, when you need similar abstraction for other data-type, you need to duplicate code of entire class (or at least the required methods). It incurs code maintenance issues, increases code size at source code as well as at binary level.

Class Templates to the rescue

```
1  template<class T>
2  class Item
3  {
4      T Data;
5  public:
6      Item() : Data( T() ) //int Data = int();
7      {}
8
9      void SetData(T nValue)
10     {
11         Data = nValue;
12     }
13
14     T GetData() const
15     {
16         return Data;
17     }
18
19     void PrintData()
20     {
21         cout << Data;
22     }
23 };
```

Usage of Class

```
1  Item<int> item1;
2  item1.SetData(120);
3  item1.PrintData();
```

Or you can cause the compiler to generate a float type data member like this:

```
1  Item<float> item2;
2  float n = item2.GetData();
```