Unit 4

Templates & Introduction to Standard Template Library

- ***Basic of templates –***

  A C++ template 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.

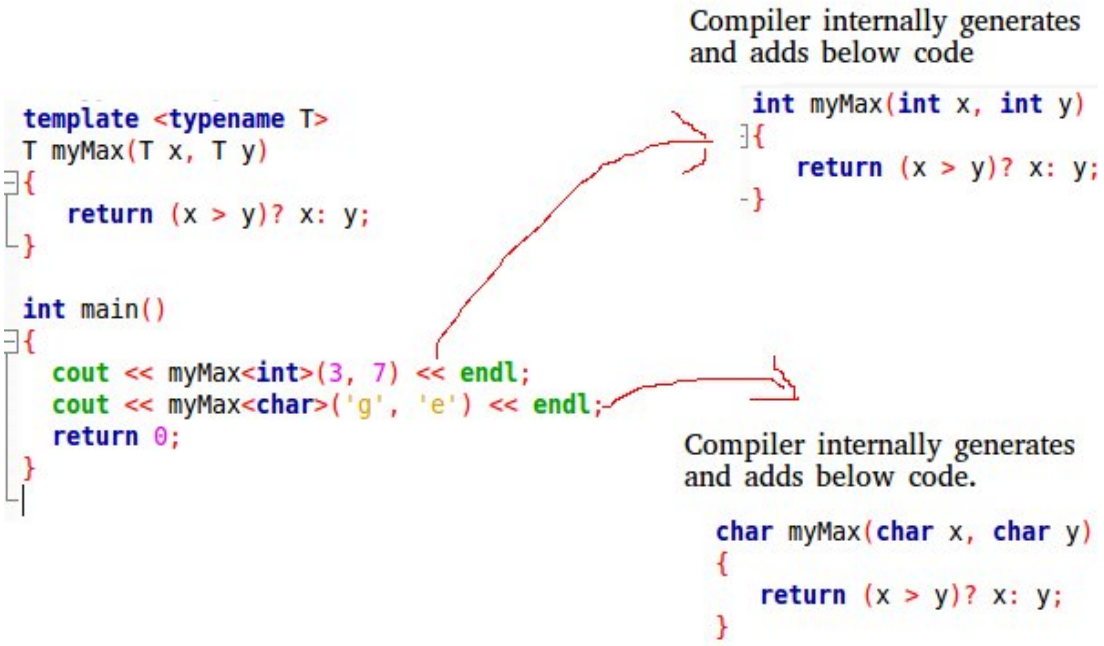
  Templates can be represented in two ways:

  - Function templates
  - Class templates

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

How Do Templates Work?

Templates are expanded at compiler time. 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 the same function/class.



- **Function Templates:**

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.

- **Class Template:**

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

- ***Function templates***
  We can create a single function to work with different data types by using a function template.

Defining a Function Template

A function template starts with the keyword `template` followed by template parameter(s) inside `<>` which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...) {
   // code
}
```

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword.

When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

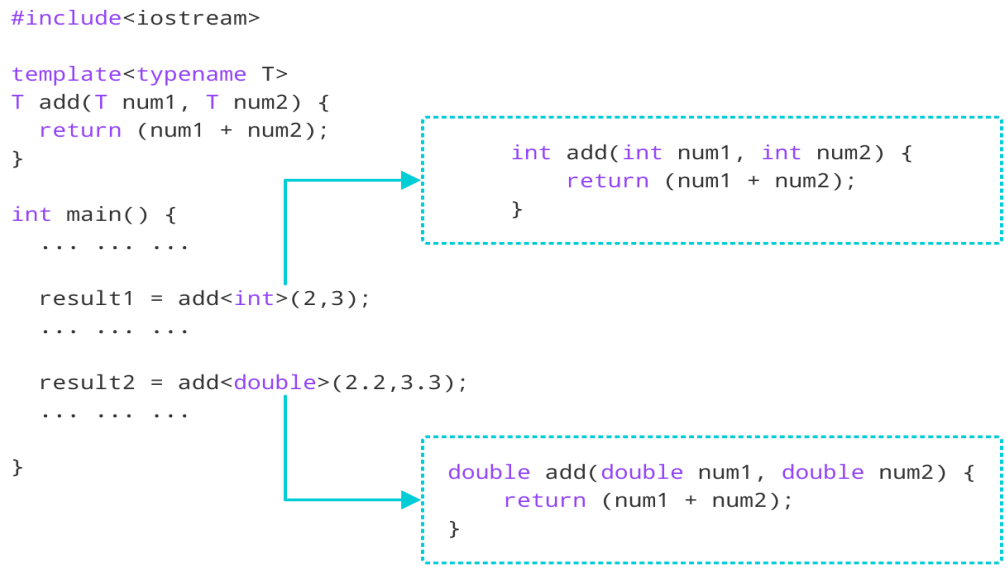
Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

```
functionName<dataType>(parameter1, parameter2,...);
```

For example, let us consider a template that adds two numbers:

```
template <typename T>
T add(T num1, T num2) {
  return (num1 + num2);
}
```



Ex 1.

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

template <typename T>
T add(T num1, T num2) {
   return (num1 + num2);
}
int main() {
   int result1;
   double result2;
   // calling with int parameters
   result1 = add<int>(2, 3);
   cout << "2 + 3 = " << result1 << endl;

   // calling with double parameters
   result2 = add<double>(2.2, 3.3);
   cout << "2.2 + 3.3 = " << result2 << endl;

   return 0;
}
```

Ex2.// C++ Program to demonstrate Use of template.

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

    return 0;
}
```

Ex3.// function template

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

template <class T>
T GetMax (T a, T b)
```

```
{
        T result;
        result = (a>b)? a : b;
        return (result);
}

int main ()
 {
  int i=5, j=6, k;
  float l=10.12, m=15.20, n;

  k=GetMax<int>(i,j);
  n=GetMax<float>(l,m);

 cout << k << endl;
  cout << n << endl;

  return 0;
}
```