Templates in C++

Templates

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- Generic program means we can create a single function or a class to work with different data types using templates.
- 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
- The simple idea is to pass data type as a parameter so that we don’t need to write same code for different data types

Function Templates

Function Templates: Generic functions that can be used for different data types.

- Syntax:

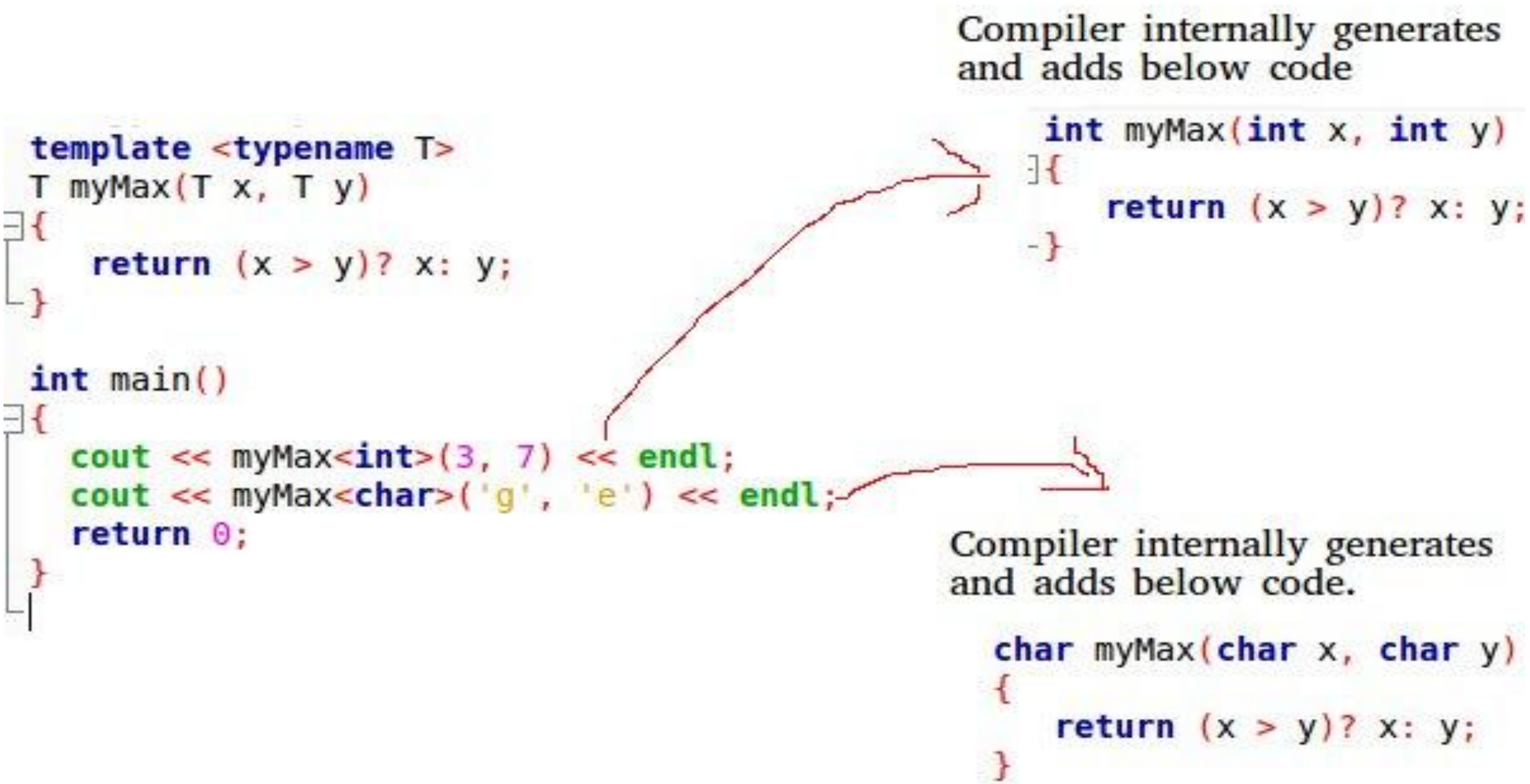
```
template<class T> Return_type
   function_name(arglist of type T)
{
   //Body of function with type T
}
Or
template<typename T> Return_type
   function_name(arglist of type T)
{
   //Body of function with type T
}
```

- Here, T is a placeholder name for a data type used by the function.

Working of templates

- Templates are expanded at compiler time.
- Templates are similar to macros, the difference is, 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.

Working of templates (Contd...)



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

- Note that it is not necessary to explicitly specify the template type in the function call(e.g. the <int> part of myMax<int>) is optional.
- The compiler can deduce it from the parameter types; hence myMax(3,7) also work just like my Max(3,7).

Function Templates with Multiple Parameters

- In **myMax(T x, T y)** function of previous example user must enter x and y of same type otherwise program generate syntax error.
- Example:

  cout << **myMax (“Hi”, 88)** << **endl**; //syntax error
- For overcoming this situation we can have Function Templates with Multiple Parameters.
- Syntax is:

```
template<class T1, class T2>
void someFunc(T1 var1, T2 var2 )
{
// some code in here...
}
```

Example : Function Templates with Multiple Parameters

```
#include <iostream>
using namespace std;
template<class T1, class T2>
void Display(T1 x, T2 y )
{
cout<<x<<“and”<<y<<“\n”;
}
int main() {
cout << “Calling function for int and string:\n";
Display(2018,”JUET”);
cout << “Calling function for float and int:\n";
Display(12.45,456);
return 0;
}
```

Output:

```
Calling function for int and string:
2018 and JUET

Calling function for float and int:
12.45 and 456
```

Overloading of Template Functions

- Function templates and non-template functions may be overloaded.
- When a template function and a non-template function are both viable for resolving a function call, the non-template function is selected.

```
template<class T> void Display(T x )
{  cout<<“Display1 x=”<<x<<“\n”; }
template<class T1, class T2> void Display(T1 x,
   T2 y )
{ cout<<Display2 : x <<x<<“and”<<y<<“\n”;}
void Display(int  x ) {  cout<<“Display3:
   x=”<<x<<“\n”; }
int main()
{
    Display(100);//  This will invoke
   not-template Display Display(30.6);
    Display(100,68.33);
    Display(‘C’);
}
```

Output:

```
Display3:x=100
Display1:x=30.6
Display2:x=100 and 68.33
Display1:x=C
```

Class Templates

- Class templates are used for data storage classes, i.e. to make a class generic class concept can be used.
- Works same way as function templates.
- Syntax:

```
template<class T>
class classname
{
     //class member specification with type T
     //wherever appropriate
};
```

- T can be substituted by any data type including user-defined types.
- A class created from class template is called a *template class.*
- Syntax for defining object of template class is:

Example of class template

```
Template <class T>
 class X
{
public:
  T square(T t)
{
   return t*t;
}
};
int main()
{
      X<int> obj1;
      int i= obj1.square(10);
      cout<<i<<endl;
     X<float>obj2;
     cout<< obj2.square(2.2);
}
```

Output:

```
100
4.84
```

Member Function Templates

- member functions of the template classes themselves are parameterized by the type argument and therefore these functions must be defined by the function templates.
- It takes the following general form:

```
Template <class T>
returntype classname <T> :: functionname(arglist)
{
……
……..
…….
}
```

Example: reading two numbers from keyboard and find the sum

```
# include<iostream>
using namespace std;
template <class T>
class sample
{  private:
     T value1, value2;
  public:
     void getdata();
     void sum();
};
template<class T>
void sample<T>:: getdata()
{  cin>>value1>>value2; }
template<class T>
void sample<T>::sum()
{   T  value;
  value=value1+value 2;
 cout<<"sum="<<value;
}
```

```
int main()
{
   sample<int> obj;
   sample<float>obj1;
   cout<<" Enter Two Integer numbers";
   obj.getdata();
   obj.sum();
   cout<<"Enter any two Floating point numbers";
   obj1.getdata();
   obj1.sum();
}
```

```
Enter Two Integer numbers 10
20
Sum= 30
Enter any two Floating point numbers
11.11
22.22
Sum= 33.33
```

Non-Type Template Arguments

- In addition to the type argument T, we can also use other arguments such as strings, constant expressions and built-in types.
- Example:

```
template<class T, int size>
class Array
{   T a[size]; //automatic array initialization
    //…………
    //………..
};
```

- This template supplies the size of the array as an argument. This implies that the size of the array is known to the complier at the compile time itself.
- **The arguments must be specified whenever a template class is created.**
- Array <int,10> a1; //array of 10 integers
- Array <float,5> a2; //array of 5 floats
- Array <char,20> a3; //string of size 20

O/P

```
#include <iostream>
using namespace std;
 template <class T>
class Test
{
private:
    T val;
public:
    static int count;
    Test()  {   count++;   }
};
```

```
template<class T>
int Test<T>::count = 0;

int main()
{
    Test<int> a;
    Test<int> b;
    Test<double> c;
    cout << Test<int>::count   << endl;
    cout << Test<double>::count << endl;
    return 0;
}
```

```
O/P:
2
1
```