TEMPLATES

TEMPLATES

- Introduction
- Function Templates
- Function Template with multiple parameters
- Class Templates
- Class Template with multiple parameters
- Overloading Template Function
- Member Function Template

Templates:

- ☐ Template is 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 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.

Function 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.

- In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

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

- The format for declaring function templates with type parameters is:

  template <class identifier> function_declaration;

  template <typename identifier> function_declaration;

- The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call GetMax to compare two integer values of type int we can write:

```
GetMax <int> (4,5);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Example:

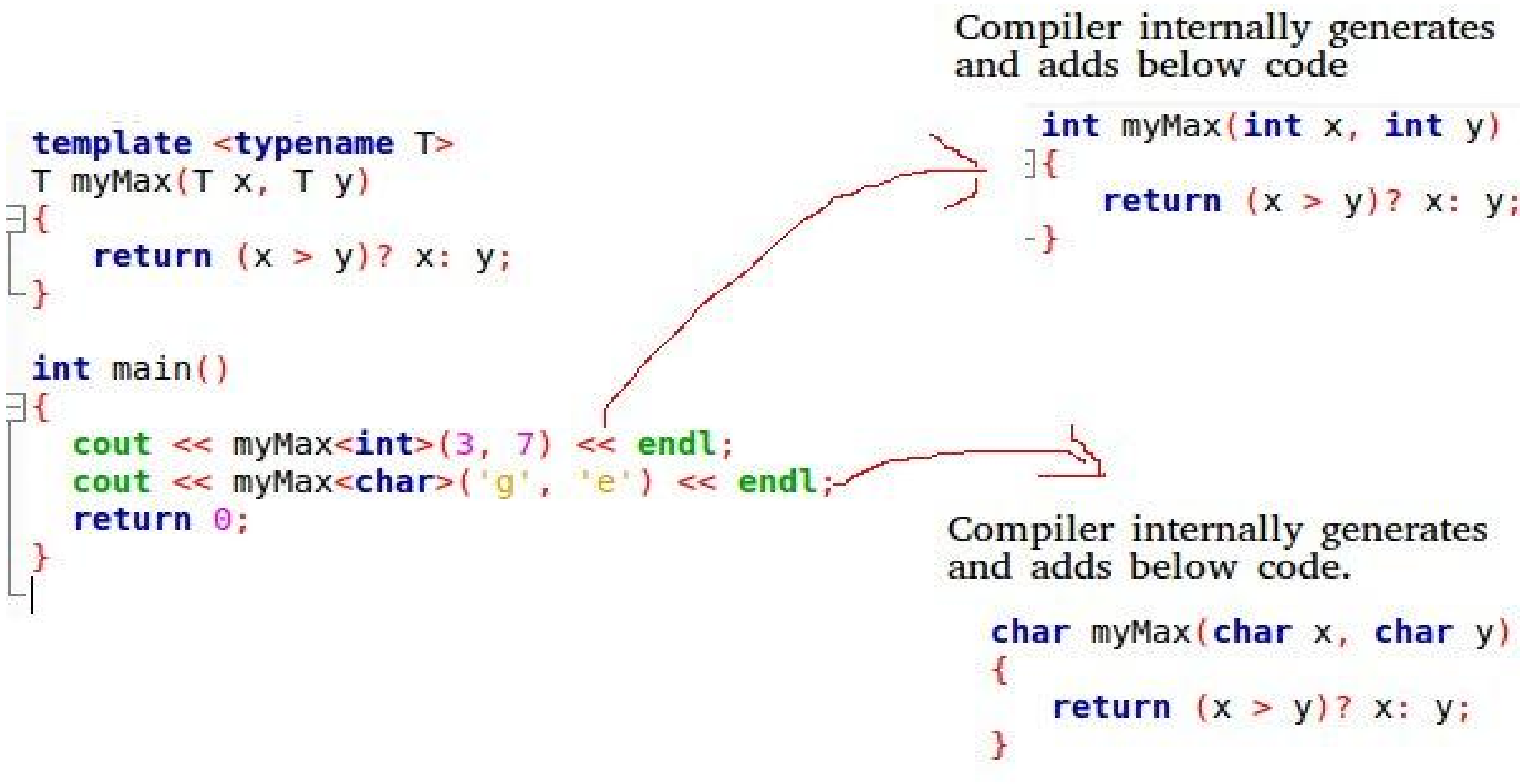
```
#include <iostream>
using namespace std;
 // One function works for all data types.  This would work even for user defined types
//if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
   return (x > y)? x: y;
}
int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for 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
```

How templates work?

Templates are expended at compiler time. This is like 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.



```
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;
}
```

Function Template with multiple parameters

Example:

```
#include <iostream>
using namespace std;
template<class A,class B,class C>
void show(A a1, B b1, C c1)
{
cout<<a1<<endl;
cout<<b1<<endl;
cout<<c1<<endl;
}
int main()
{
   show('A',22,44.55);
   show(99.99,56,'C');
}
```

Explanation: In the above program, a template with A, B, and C classes are declared. The function show() has three arguments of type A, B, and C respectively. In main(), the function show() is invoked, and three values of different data types are passed. The function show() displays the values on the screen.

Example2:

```
#include<iostream.h>
#include<conio.h>
template <class T1, class T2>
class data
{
   public:
   void show (T1 a, T2 b)
   {cout<<"\na="<<a <<" b="<<b;}
};
int main()
{
   clrscr();
   int i[]={3,5,2,6,8,9,3};
   float f[]={3.1,5.8,2.5,6.8,1.2,9.2,4.7};
   data <int,float> h;
   for (int m=0;m<7;m++)
   h.show(i[m],f[m]);
   return 0;
}
```

OUTPUT

```
a = 3 b = 3.1
a = 5 b = 5.8
a = 2 b = 2.5
a = 6 b = 6.8
a = 8 b = 1.2
a = 9 b = 9.2
a = 3 b = 4.7
```

Explanation: In the above program, array elements of the integer and float array are passed to the function show(). The function show() has two arguments of template type, that is, a and b. The show() function receives integer and float values and displays them. The output of the program is shown above.

Class templates

A class can have members that use template parameters as types.

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36); //creating object of mypair class with int types
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);  //creating object of mypair class with float types
```

Example 2

```
#include <iostream>
using namespace std;
template <class T>
class mypair {
    T a, b;
  public:
   T mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

```
int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

output:

```
100
```

Example3:

```
#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
```

Class Template with multiple parameters

Can there be more than one arguments to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

```
#include<iostream>
using namespace std;
template<class T, class U>
class A  {
    T x;
    U y;
Public:
    A() {    cout<<"Constructor Called"<<endl;   }
};
int main()  {
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

Output:

```
Constructor Called
Constructor Called
```

Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

```
#include<iostream>
using namespace std;
template<class T, class U = char>
class A  {
public:
    T x;
    U y;
    A() {   cout<<"Constructor Called"<<endl;   }
};

int main()  {
   A<char> a;  // This will call A<char, char>
   return 0;
}
```

Output:

```
Constructor Called
```

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.

Templates and Static variables.

Function templates and static variables:

Each instantiation of function template has its own copy of local static variables. For example, in the following program there are two instances: void fun(int) and void fun(double). So two copies of static variable i exist.

Example:

```
#include <iostream>
using namespace std;
template <typename T>
void fun(const T& x)
{
  static int i = 10;
  cout << ++i<<endl;
  cout<<x<<endl;
  return;
}
int main()
{
  fun<int>(1);  // prints 11
  cout << endl;
  fun<int>(2);  // prints 12
  cout << endl;
  fun<double>(1.1); // prints 11
  cout << endl;
  getchar();
  return 0; }
```

Output:

```
11
1
12
2
11
1.1
```

Class templates and static variables:

The rule for class templates is same as function templates

Each instantiation of class template has its own copy of member static variables. For example, in the following program there are two instances of classTest. So two copies of static variable count exist.

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

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

```
int main()
{
  Test<int> a;
 // value of count for Test<int> is 1 now
  Test<int> b;
// value of count for Test<int> is 2 now
  Test<double> c;
 // value of count for Test<double> is 1 now
  cout << Test<int>::count   << endl;
 // prints 2
  cout << Test<double>::count << endl;
//prints 1

  return 0;
}
```

Output:

```
  2
  1
```

What is template specialization?

Template specialization allows us to have different code for a particular data type

What if we want a different code for a particular data type?

Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?

It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
    // code to implement Quick Sort
}
 // Template Specialization: A function specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
    // code to implement counting sort
}
```

Another example could be a class Set that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

An Example Program for function template specialization

For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

```cpp
#include <iostream>
using namespace std;
template <class T>
void fun(T a)
{
   cout << "The main template fun(): "<< a
<< endl;
}
 template<>
void fun(int a)
{
    cout << "Specialized Template for int
type: "<< a << endl;
}
int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(10.14);
}
```

Output:

```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14
```

An Example Program for class template specialization

In the following program, a specialized version of class Test is written for int data type.

```
#include <iostream>
using namespace std;
template <class T>
class Test
{
  // Data memnbers of test
public:
   Test()
   {
       // Initialization of data members
       cout << "General template object \n";
   }
   // Other methods of Test
};
template <>
class Test <int>
{
public:
   Test()
   {
  cout << "Specialized template object\n";
   }
};
int main()
{
    Test<int> a;
    Test<char> b;
    Test<float> c;
    return 0;
}
```

Output:

```
Specialized template object
General template object
General template object
```

How does template specialization work?

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Overloading Template Function

You may overload a function template either by a non-template function or by another function template.

If you call the name of an overloaded function template, the compiler will try to deduce its template arguments and check its explicitly declared template arguments. If successful, it will instantiate a function template specialization, then add this specialization to the set of candidate functions used in overload resolution. The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions.

Example:

```
#include <iostream>
using namespace std;
template<class T>
void f(T x, T y) {
cout << "Template" << endl;
}
void f(int w, int z) {
  cout << "Non-template" << endl;
}
int main() {
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');  return 0;
}
```

output:

```
Non-template
Template
Non-template
```

Explanation

The function call f(1, 2) could match the argument types of both the template function and the non-template function. The non-template function is called because a non-template function takes precedence in overload resolution.

The function call f('a', 'b') can only match the argument types of the template function. The template function is called.

Argument deduction fails for the function call f(1, 'b'); the compiler does not generate any template function specialization and overload resolution does not take place. The non-template function resolves this function call after using the standard conversion from char to int for the function argument 'b

Member Function Template

Member functions can themselves be function templates, specifying additional parameters, as in the following example.

```
template<typename T>
class X
{
public:
   template<typename U>
   void mf(const U &u);
};

template<typename T>
template <typename U>
void X<T>::mf(const U &u)
{
}
int main()
{
//create object for class and call the methods here
}
```

Member Functions of Template Classes:

Member functions can be defined inside or outside of a class template. They are defined like function templates if defined outside the class template.

```
// member_function_templates
template<class T, int i>
class MyStack
{
    T*  pStack;
    T StackBuffer[i];
   static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};
template< class T, int i >
 MyStack< T, i >::MyStack( void )
{
};
template< class T, int i >
void MyStack< T, i >::push( const T item )
{
};
template< class T, int i >
T& MyStack< T, i >::pop( void )
{
};
int main()
{
}
```

Note that just as with any template class member function, the definition of the class's constructor member function includes the template argument list twice.