CAP444
OBJECT ORIENTED PROGRAMMING USING C++

Unit5

Created By:
Kumar Vishal
(SCA), LPU

Unit-5

Generic programming with templates :

- need of template,
- class template,
- function template,
- overloading of function template,
- recursion with template function,
- class template and inheritance,
- difference between templates and macros

Template in C++:

- A features which provide us a generic function and generic class
- It represents in two ways:
  - Function template
  - Class template

Function template

- We can define a template for a function.
- Generic functions use the concept of a function template.
- The type of the data that the function will define it depends up on the type of the data passed as a parameter.
- It is created by using the keyword template

Syntax:

```
template <class Ttype>
returnType functionName(parameterList)
{
   // body of function.
}
```

class or typename

Template in C++:

- A features which provide us a generic function and generic class
- It represents in two ways:
  - Function template
  - Class template

What is a template?

A. A template is a formula for creating a generic class
B. A template is used to manipulate the class
C. A template is used for creating the attributes
D. None of the above

Function template

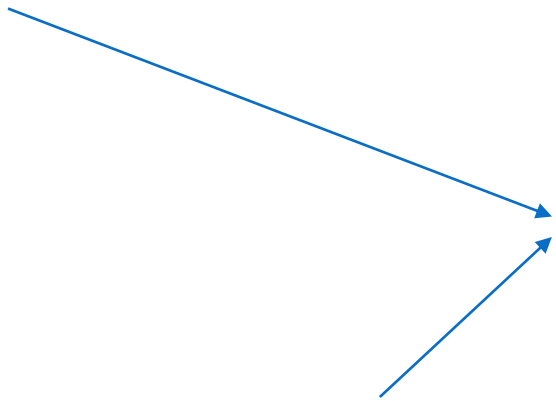
- We can define a template for a function.
- Generic functions use the concept of a function template.
- The type of the data that the function will define it depends up on the type of the data passed as a parameter.
- It is created by using the keyword template

Syntax:

```
template <class Ttype>
returnType functionName(parameterList)
{
   // body of function.
}
```

Example:

```
template <class T>
T add(T a, T b)
{
   T c=a+b;
   return c;
}
```



Placeholder

```
template <typename T>
T add(T a, T b)
{
   T c=a+b;
   return c;
}
```

Which keyword can be used in template?

a) class

b) typename

c) both class & typename

d) function

What will be output?

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

template <class T>
T add(T a, T b)
{
   T c;
   c=a+b;
   return c;
}
int main()
{
cout<<add(5.5f,4);
return 0;
}
```

A. 9.5
B. 9.0
C. Compilation Error
D. Nothing will print

Function template with different type parameters

Like :

template <class T1,class T2>

here, T1 and T2 is different type

Example:

```
template <class T1,class T2>
T1 add(T1 a, T2 b)
{
   T1 c;
   c=a+b;
   return c;
}
```

Restrictions of Generic Functions

Generic functions perform the same operation for all the different data type.

For example If function is performing addition then it will perform addition only it will not perform subtraction, multiplication etc.…

There is the difference between function overloading and function template.

overloading of function template

- Function template also can be overload
- Same function name with different signature

is called function overloading and same thing we can apply for template also.

For example there is a template function for insert which take one parameter and with the same name insert we can create one more template with two parameters.

Example

What is the output of this program?

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

template <typename T>
void print_mydata(T output)
{
cout<<output;
}
int main()
{
double d = 5.5;
string s("Hello World");
print_mydata( d );
print_mydata( s );
return 0;
}
```

5.5

Hello World

5.5Hello World

None

recursion with template function

- The process in which a function call by itself is called recursion and the corresponding function is called as recursive function.
- We can create recursive template function

Example:

Which statement is correct about function template?

Statement1:Function template also can be overload

Statement2:We can create recursive template function

A. Statement1 is correct
B. Statement2 is correct
C. both Statements are correct
D. None

Class template

Class template :

To create class in generic form.

Syntax

template <class type>

class className

{

}

What will be out put?

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
            A(){
                        cout<<"Created";
            }
            ~A(){
                        cout<<"Destroyed";
            }
};

int main()
{
            A a;
            return 0;
}
```

A: Created

B. Destroyed

C. CreatedDestroyed

D. Compilation Error

What will be out put?

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
            A(){
                        cout<<"Created";
            }
            ~A(){
                        cout<<"Destroyed";
            }
};

int main()
{
            A <int>a;
            return 0;
}
```

A: Created

B. Destroyed

C. CreatedDestroyed

D. Compilation Error

class template and inheritance:

A. **template to template**

B. **class to template**

C. **template to class**

class template and inheritance

One normal class inheriting template class:

```
template<class T>
class Base
{
public:
   T VariableName;
};
```

```
class Der : public Base<int>
{

};
```

What is the output of this program?

```
#include <iostream>
using namespace std;
template<class T>
class A
{
  public:
  func(T a,T b){return a+b;}
};
int main()
{
  A <int>a1;
  cout<<a1.func(3,2);
  return 0;
}
```

5

Compilation Error

Nothing will print

None

One template class inheriting other template class:

```
template<class T>
class Base
{
public:
    T VariableName;
};
```

```
template<class T>
class Derive: public Base<int>
{
public:
    T VariableName;
};
```

Which statement is correct about template class?

Statement1:One normal class inheriting template class

Statement2:One template class inheriting other template class

A. Statement1 is correct
B. Statement2 is correct
C. both Statements are correct
D. None

What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
template <class type>
class Test
{
   public:
type Funct1(type Var1)
   {
      return Var1;
   }
   type Funct2(type Var2)
   {
      return Var2;
   }
};
int main()
  {
     Test<int> Var1;
     Test<double> Var2;
     cout << Var1.Funct1(200);
     cout << Var2.Funct2(3.123);
     return 0;
  }
```

a) 100

b) 200

c) 3.123

d) 2003.123

What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
template <class type>
class Test
{
   public:
Test()    {    };
    ~Test()    {
    };
  type Funct1(type Var1)
    {
       return Var1;
    }

};
int main()
 {
    Test<int> Var1;
     cout << Var1.Funct1(200);
      return 0;
 }
```

a) 100

b) 200

c) 3.123

d) 2003.123

What will be output?

```
#include <iostream>

using namespace std;
template<class T>
class A
{
   public:
            T func(T a, T b){
                     return a/b;
            }
};

int main()
{
            A <int>a1;
            cout<<a1.func(3,2);
            cout<<a1.func(3.0,2.0);
            return 0;
}
```

A. 11

B. 12

C. 10

D. Error

Macros

- The preprocessors statements means a instructions to the compiler to preprocess the information before actual compilation starts.
- All preprocessor statement begin with #
- There are number of preprocessor statements like #include, #define, #if, #else, #line, etc.
- The #define preprocessor statement creates symbolic constants. The symbolic constant is called a macro

syntax:

#define macro-name replacement-text

Macro

- No longer
- No repetition
- Define in short

Predefine Macros

__LINE__

This contains the current line number of the program when it is being compiled.

__FILE__

This contains the current file name of the program when it is being compiled.

__DATE__

This contains a string of the form month/day/year that is the date of the translation of the source file into object code.

__TIME__

This contains a string of the form hour: minute: second that is the time at which the program was compiled.

Example:

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

#define PI 3.14159

int main () {
   cout << "Value of PI :" << PI << endl;
   return 0;
}
```

What will be the output of the following C++ code?

```
#include <iostream>
using namespace std;
int main ()
{
   cout << "Value of __LINE__ : " << __LINE__ << endl;
   cout << "Value of __FILE__ : " << __FILE__ << endl;
   cout << "Value of __DATE__ : " << __DATE__ << endl;
   cout << "Value of __TIME__ : " << __TIME__ << endl;
   return 0;
}
```

a) 5
b) details about your file
c) compile time error
d) runtime error

```
#include <iostream>
#define kv cout<<"kumar vishal"<<endl;
using namespace std;

int main()
{
 kv
 return 0;
}
```

What will be output?

```
#include <iostream>

using namespace std;
#define pi=3.14
int main()
{
  double r=2;
  cout<<"Area"<<pi*r*r<<endl;
  return 0;
}
```

A. Area12.56
B. Area0
C. Compilation error
D. Run time error

What will be output?

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

#define pi 3.14
int main()
{
  double r=2;
  cout<<"Area"<<pi*r*r<<endl;
  return 0;
}
```

A. Area12.56
B. Area0
C. Compilation error
D. Run time error

What will be output?

```
#include <iostream>
#define kv(name)
cout<<name;
using namespace std;
int main()
{
 kv("kumar")
 kv("vishal")
 kv("Rohit")
 return 0;
}
```

A. kumarvishalRohit
B. kumarvishal
C. kumarRohit
D. None

Macro in single line

```
#define kv cout<<"kumar vishal"<<endl;
```

Macro in multi line: use \

```
#include <iostream>
#define product(x,y)\
{cout<<\
(x*y);}
using namespace std;
int main()
{
 product(5,4)
 return 0;
}
```

difference between macro and function:

MACRO	FUNCTION
Macro is Preprocessed	Function is Compiled
Using Macro increases the code length	Using Function keeps the code length unaffected
Speed of Execution using Macro is Faster	Speed of Execution using Function is Slower
Before Compilation, macro name is replaced by macro value	During function call, transfer of control takes place
Macros are useful when small code is repeated many times	Functions are useful when large code is to be written

Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates.

```
template<class T1, class T2 = char>
class A  {
public:
    T1 x;
    T2 y;
    A() {   cout<<"Constructor Called"<<endl;   }
};
```

Any Query?