* ***Purpose*** **: Classwork.**

* ***Date*** **: 07/01/2026**

* ***Author*** **: Vikas Srivastava**

* ***ID*** **: 55984**

* ***Batch ID*** **: 25SUB4505**

---

1. ***Code :***

classTemplateFour.cpp

```cpp
// Purpose - Write a program to demonstrate class templates and template specialization in C++, showing how a generic template works for basic data types and a specialized version handles const char* differently.
#include <iostream>
#include <cstring>
using namespace std;

template <typename T>
class Sample{ //generic class
  T data;
public:
  Sample(T arg=T()):data(arg) {}
  void disp(){ cout<<"data: "<<data<<endl; }
};

template <>
class Sample<const char *>{ //specialized class
  char *data;
public:
  Sample(const char* arg) {
    data = new char[strlen(arg) + 1];
    strcpy(data, arg);
  }
  void disp(){ cout<<"const char * --->  data: "<<data<<endl; }
  ~Sample(){ delete []data; }
};

int main(){
  Sample<int> iObj(100);
  Sample<double> dObj(10.234);
  Sample<const char *> sObj("Some string here");

  iObj.disp();
  dObj.disp();
  sObj.disp();
}
```

Output :

```
data: 100
data: 10.234
const char * --->  data: Some string here
```

2. ***Code :***

classTemplateOne.cpp

```cpp
// Purpose - Write a program to demonstrate the use of class templates in C++ for handling different data types using a single generic class.
#include <iostream>
using namespace std;

template <typename T>
class Sample{
  T data;
public:
  Sample(T arg=T()):data(arg) {}
  void disp(){ cout<<"data: "<<data<<endl; }
};

int main(){
  Sample<int> iObj(100);
  Sample<double> dObj(10.234);
  Sample<string> sObj("Some string here");

  iObj.disp();
  dObj.disp();
  sObj.disp();
}
```

Output :

```
data: 100
data: 10.234
data: Some string here
```

3. ***Code :***

classTemplateOne.cpp | classTemplateThree.cpp

```cpp
// Purpose - Write a program to demonstrate a class template in C++ with member functions defined outside the class and its use with different data types.

#include <iostream>
using namespace std;

template <typename T>
class Sample{
  T data;
public:
  Sample(T arg=T()):data(arg) {}
  T getValue();
  void disp();
};

template<typename T>
T Sample<T>::getValue(){
  return data;
}

template<typename T>
void Sample<T>::disp(){
  cout<<"data: "<<data<<endl;
}


int main(){
  Sample<int> iObj(100);
  Sample<double> dObj(10.234);
  Sample<string> sObj("Some string here");

  iObj.disp();
  dObj.disp();
  sObj.disp();

  cout<<"In main: "<<iObj.getValue()<<endl;
  cout<<"In main: "<<dObj.getValue()<<endl;
  cout<<"In main: "<<sObj.getValue()<<endl;
}
```

Output :

```
data: 100
data: 10.234
data: Some string here
In main: 100
In main: 10.234
In main: Some string here
```

4. ***Code :***

classTemplateTwo.cpp

```cpp
// Purpose - Write a program to demonstrate a class template in C++ with member functions defined inside the class and its usage with different data types.

#include <iostream>
using namespace std;

template <typename T>
class Sample{
  T data;
public:
  Sample(T arg=T()):data(arg) {}
  T getValue(){return data; }
  void disp(){ cout<<"data: "<<data<<endl; }
};

int main(){
  Sample<int> iObj(100);
  Sample<double> dObj(10.234);
  Sample<string> sObj("Some string here");

  iObj.disp();
  dObj.disp();
  sObj.disp();

  cout<<"In main: "<<iObj.getValue()<<endl;
  cout<<"In main: "<<dObj.getValue()<<endl;
  cout<<"In main: "<<sObj.getValue()<<endl;
}
```

Output :

```
data: 100
data: 10.234
data: Some string here
In main: 100
In main: 10.234
In main: Some string here
```

5. ***Code :***

classTemplateTwo.cpp | funTemplateFive.cpp

```cpp
// Purpose - Write a program to demonstrate a function template in C++, showing both implicit and explicit template function calls with different data types.

#include <iostream>
using namespace std;

template <typename DT>
void fun(DT var){
  cout<<"var: "<<var<<endl;
}

int main(){
  fun(10); //Implicit call
  fun<int>(10); //Explicit call fun<datatype>(arg);
      //datatype is passed as an argument along with actual
      //arguments. Datatype is passed in <data type> (actual args)

  fun<double>(234.345);
  fun(234.345);
}
```

Output :

```
var: 10
var: 10
var: 234.345
var: 234.345
```

6. ***Code :***

funTemplateFour.cpp

```cpp
// Purpose - Write a program to demonstrate function template overloading in C++ for performing addition with different numbers of arguments and data types.
#include <iostream>
using namespace std;

template<typename T>
T Add(T x, T y){
  return x+y;
}

template<typename T>
T Add(T x, T y, T z){
  return x+y+z;
}

int main(){
  cout<<"Result :"<<Add(10,20)<<endl;
  cout<<"Result :"<<Add(10.234,20.567)<<endl;

  cout<<"Result :"<<Add(10.234,20.567,834.34)<<endl;
  cout<<"Result :"<<Add(10,20,30)<<endl;
}
```

Output :

```
Result :30
Result :30.801
Result :865.141
Result :60
```

7. ***Code :***

funTemplateOne.cpp

```cpp
// Purpose - Write a program to demonstrate a function template in C++ handling arguments of different data types using a single generic function.
#include <iostream>
using namespace std;

template <typename DT>
void fun(DT var){
  cout<<"var: "<<var<<endl;
}

int main(){
  fun(10);// int argument passed
  fun('A');// char argument passed
  fun(10.234f);// Float argument passed
  fun(10.234);// double argument passed
  fun("C Style String");// const char * argument passed
}
```

Output :

```
var: 10
var: A
var: 10.234
var: 10.234
var: C Style String
```

8. ***Code :***

funTemplateSix.cpp

```cpp
// Purpose - Write a program to demonstrate template specialization (function overloading) in C++, showing how a generic function template and a
specialized version handle different data types, especially const char*.

#include <iostream>
using namespace std;

template <typename DT>
void fun(DT var){ //generalized/generic function template
  cout<<"var: "<<var<<endl;
}
void fun(const char *cStr){ //specialized/specific function template
  cout<<"C Style: "<<cStr<<endl;
}

int main(){
  fun(10);
  fun(234.345);
  fun("C Style String Here");
}
```

Output :

```
var: 10
var: 234.345
C Style: C Style String Here
```

9. Code :

funTemplateThree.cpp

```cpp
// Purpose - Write a program to demonstrate a function template in C++ with multiple template parameters, allowing the function to accept arguments of different data types.

#include <iostream>
using namespace std;

template <typename DT, typename DU>
void fun(DT varOne, DU varTwo){
  cout<<"varOne: "<<varOne<<"\tvarTwo: "<<varTwo<<endl;
}

int main(){
  fun(10, 9273.35);
  fun('A', 10);
  fun(10.234f, "Hello");
  fun(10.234, 100);
  fun("C Style String", 123.34);
}
```

Output :

```
varOne: 10  varTwo: 9273.35
varOne: A varTwo: 10
varOne: 10.234  varTwo: Hello
varOne: 10.234  varTwo: 100
varOne: C Style String  varTwo: 123.34
```

10. Code :

funTemplateTwo.cpp

```cpp
// Purpose - Write a program to demonstrate a simple function template in C++ for performing addition of two values of the same data type.
#include <iostream>
using namespace std;

template<typename T>
T Add(T x, T y){
  return x+y;
}

int main(){
  cout<<"Result :"<<Add(10,20)<<endl;
  cout<<"Result :"<<Add(10.234,20.567)<<endl;
}
```

Output :

```
Result :30
Result :30.801
```

11. Code :

sharedPtrCyclicDepFix.cpp

```cpp
// Purpose - Write a program to demonstrate breaking circular references using weak_ptr in C++, allowing objects with mutual references to be properly destroyed.
#include <iostream>
#include <memory>
using namespace std;

struct B;

struct A{
  shared_ptr<B> b;
  ~A(){cout<<"~A()"<<endl; }
};

struct B{
  weak_ptr<A> a;// weak pointer
  ~B(){cout<<"~B()"<<endl; }
};


int main(){
  shared_ptr<A> objA(new A());
  shared_ptr<B> objB(new B());

  objA->b = objB;
  objB->a = objA;
}//destructors are not called
```

Output :

```
~A()
~B()
```

12. Code :

sharedPtrOne.cpp

```cpp
// Purpose - Write a program to demonstrate reference counting with shared_ptr in C++, showing how the use count changes as shared_ptr instances are copied and go out of scope.

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

int main(){
  shared_ptr<int> ptrOne(new int(100));

  cout<<"1 Use count/Reference Count: "<<ptrOne.use_count()<<endl;

  {
      shared_ptr<int> ptrTwo = ptrOne; //Local temp object

      cout<<"2. Use count/Reference Count: "<<*ptrOne<<"\t"<<ptrOne.use_count()<<endl;
      cout<<"2. Use count/Reference Count: "<<*ptrTwo<<"\t"<<ptrTwo.use_count()<<endl;
  }
  cout<<"3 Use count/Reference Count: "<<ptrOne.use_count()<<endl;
}
```

Output :

```
1 Use count/Reference Count: 1
2. Use count/Reference Count: 100 2
2. Use count/Reference Count: 100 2
3 Use count/Reference Count: 1
```

13. Code :

sharedPtrTwo.cpp

```cpp
// Purpose - Write a program to demonstrate shared_ptr managing a user-defined class in C++, showing reference counting and automatic destruction when shared_ptr instances go out of scope.
#include <iostream>
#include <memory>
using namespace std;

class Test{
  int data;
public:
  Test(int num = 0):data(num) {cout<<"Test()"<<endl;}
  ~Test(){cout<<"~Test()"<<endl;}
  void disp(){cout<<"Test::disp(): "<<data<<endl; }
};

int main(){
  shared_ptr<Test> ptrOne(new Test(100));

  cout<<"1 Use count/Reference Count: "<<ptrOne.use_count()<<endl;

  {
      shared_ptr<Test> ptrTwo = ptrOne; //local temp object

      cout<<"2. Use count/Reference Count: "<<"\t"<<ptrOne.use_count()<<endl;
      cout<<"2. Use count/Reference Count: "<<"\t"<<ptrTwo.use_count()<<endl;
  }
  cout<<"3 Use count/Reference Count: "<<ptrOne.use_count()<<endl;
}
```

Output :

```
Test()
1 Use count/Reference Count: 1
2. Use count/Reference Count:   2
2. Use count/Reference Count:   2
3 Use count/Reference Count: 1
~Test()
```

14. Code :

uniquePtrThree.cpp

```cpp
// Purpose - Write a program to demonstrate unique_ptr in C++ for exclusive ownership of a dynamically allocated object, showing automatic destruction when the pointer goes out of scope.
#include <memory>
#include <iostream>
using namespace std;

class Test{
  int data;
public:
  Test(int num = 0):data(num) {cout<<"Test()"<<endl;}
  ~Test(){cout<<"~Test()"<<endl;}
  void disp(){cout<<"Test::disp(): "<<data<<endl; }
};

int main(){
  //unique_ptr<Test> ptrOne= make_unique<Test>(100);
  unique_ptr<Test> ptrOne(new Test(100));
  ptrOne->disp();
  cout<<"In main()"<<endl;
}
```

Output :

```
Test()
Test::disp(): 100
In main()
~Test()
```

15. Code :

uniquePtrTwo.cpp

```cpp
// Purpose - Write a program to demonstrate transfer of ownership with unique_ptr in C++ using std::move, showing that the original pointer becomes nullptr after the move.
#include <memory>
#include <iostream>
using namespace std;

int main(){
  unique_ptr<int> ptrOne(new int(100));

  cout<<"*ptrOne: "<<*ptrOne<<endl;
  unique_ptr<int> ptrTwo = move(ptrOne); //moving of resource done here

  if (! ptrOne)
    cout<<"ptrOne is null after the move"<<endl;
  cout<<"*ptrTwo: "<<*ptrTwo<<endl;
}
```

Output :

```
*ptrOne: 100
ptrOne is null after the move
*ptrTwo: 100
```