|  |
| --- |
| Question 4 |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;  class Point {  public:      Point() { cout << "Constructor called"; }  };    int main()  {     Point t1, \*t2;     return 0;  } | |
| A | Compiler Error |
| B | Constructor called Constructor called |
| C | Constructor called |

Explanation:

Only one object t1 is constructed here. t2 is just a pointer variable, not an object

Point \*t1, \*t2; // No constructor call

t1 = new Point(10, 15); // Normal constructor call

t2 = new Point(\*t1); // Copy constructor call

Point t3 = \*t1; // Copy Constructor call

Point t4; // Normal Constructor call

t4 = t3; // Assignment operator call

|  |
| --- |
| Question 6 |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;    class X  {  public:      int x;  };    int main()  {      X a = {10};      X b = a;      cout << a.x << " " << b.x;      return 0;  } | |
| A | Compiler Error |
| B | 10 followed by Garbage Value |
| C | 10 10 |
| D | 10 0 |

Explanation:

The following may look like an error, but it works fine. X a = {10}; Like structures, class objects can be initialized. The line "X b = a;" calls copy constructor and is same as "X b(a);". Please note that, if we don't write our own copy constructor, then compiler creates a default copy constructor which assigns data members one object to other object.

|  |
| --- |
| Question 7 |

What is the output of following program?

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    class Point  {      int x, y;  public:     Point(const Point &p) { x = p.x; y = p.y; }     int getX() { return x; }     int getY() { return y; }  };    int main()  {      Point p1;      Point p2 = p1;      cout << "x = " << p2.getX() << " y = " << p2.getY();      return 0;  } | |
| A | x = garbage value y = garbage value |
| B | x = 0 y = 0 |
| C | Compiler Error |

Explanation:

There is compiler error in line "Point p1;". The class Point doesn't have a constructor without any parameter. If we write any constructor, then compiler doesn't create the [default constructor](http://en.wikipedia.org/wiki/Default_constructor). It is not true other way, i.e., if we write a default or parameterized constructor, then compiler creates a copy constructor. See the next question.

|  |
| --- |
| Question 8 |

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    class Point  {      int x, y;  public:     Point(int i = 0, int j = 0) { x = i; y = j; }     int getX() { return x; }     int getY() { return y; }  };    int main()  {      Point p1;      Point p2 = p1;      cout << "x = " << p2.getX() << " y = " << p2.getY();      return 0;  } | |
| A | Compiler Error |
| B | x = 0 y = 0 |
| C | x = garbage value y = garbage value |

Explanation:

Compiler creates a copy constructor if we don't write our own. Compiler writes it even if we have written other constructors in class. So the above program works fine. Since we have default arguments, the values assigned to x and y are 0 and 0.

|  |
| --- |
| Question 9  Wrong |

Predict the output of following program.

|  |  |
| --- | --- |
| #include<iostream>  #include<stdlib.h>  using namespace std;    class Test  {  public:     Test()     { cout << "Constructor called"; }  };    int main()  {      Test \*t = (Test \*) malloc(sizeof(Test));      return 0;  } | |
| A | Constructor called |
| B | Empty |
| C | Compiler Error |
| D | Runtime error |

Explanation:

Unlike new, malloc() doesn't call constructor (See [this](http://www.geeksforgeeks.org/malloc-vs-new/)) If we replace malloc() with new, the constructor is called, see [this](http://ideone.com/l7APkK).

|  |
| --- |
| Question 10 |

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    class Test  {  public:        Test() { cout << "Hello from Test() "; }  } a;    int main()  {      cout << "Main Started ";      return 0;  } | |
| A | Main Started |
| B | Main Started Hello from Test() |
| C | Hello from Test() Main Started |
| D | Compiler Error: Global objects are not allowed |

Explanation:

Output is

Hello from Test() Main Started

There is a global object 'a' which is constructed before the main functions starts, so the constructor for a is called first, then main()' execution begins.

Question 12 Explanation:

Objects must be passed by reference in copy constructors. Compiler checks for this and produces compiler error if not passed by reference. The following program compiles fine and produces output as 10.

#include <iostream >

using namespace std;

class Point {

int x;

public:

Point(int x) { this->x = x; }

Point(const Point **&**p) { x = p.x;}

int getX() { return x; }

};

int main()

{

Point p1(10);

Point p2 = p1;

cout << p2.getX();

return 0;

}

The reason is simple, if we don't pass by reference, then argument p1 will be copied to p. So there will be a copy constructor call to call the copy constructor, which is not possible.

**When do we use Initializer List in C++?**

Initializer List is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon. Following is an example that uses initializer list to initialize x and y of Point class.

|  |
| --- |
| #include<iostream>  using namespace std;    class Point {  private:      int x;      int y;  public:      Point(int i = 0, int j = 0):x(i), y(j) {}      /\*  The above use of Initializer list is optional as the          constructor can also be written as:          Point(int i = 0, int j = 0) {              x = i;              y = j;          }      \*/        int getX() const {return x;}      int getY() const {return y;}  };    int main() {    Point t1(10, 15);    cout<<"x = "<<t1.getX()<<", ";    cout<<"y = "<<t1.getY();    return 0;  }    /\* OUTPUT:     x = 10, y = 15  \*/ |

The above code is just an example for syntax of Initializer list. In the above code, x and y can also be easily initialed inside the constructor. But there are situations where initialization of data members inside constructor doesn’t work and Initializer List must be used. Following are such cases:

**1) For initialization of non-static const data members:**  
const data members must be initialized using Initializer List. In the following example, “t” is a const data member of Test class and is initialized using Initializer List.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test {      const int t;  public:      Test(int t):t(t) {}  //Initializer list must be used      int getT() { return t; }  };    int main() {      Test t1(10);      cout<<t1.getT();      return 0;  }    /\* OUTPUT:     10  \*/ |

**2) For initialization of reference members:**  
Reference members must be initialized using Initializer List. In the following example, “t” is a reference member of Test class and is initialized using Initializer List.

|  |
| --- |
| // Initialization of reference data members  #include<iostream>  using namespace std;    class Test {      int &t;  public:      Test(int &t):t(t) {}  //Initializer list must be used      int getT() { return t; }  };    int main() {      int x = 20;      Test t1(x);      cout<<t1.getT()<<endl;      x = 30;      cout<<t1.getT()<<endl;      return 0;  }  /\* OUTPUT:      20      30   \*/ |

**3) For initialization of member objects which do not have default constructor:**  
In the following example, an object “a” of class “A” is data member of class “B”, and “A” doesn’t have default constructor. Initializer List must be used to initialize “a”.

|  |
| --- |
| #include <iostream>  using namespace std;    class A {      int i;  public:      A(int );  };    A::A(int arg) {      i = arg;      cout << "A's Constructor called: Value of i: " << i << endl;  }    // Class B contains object of A  class B {      A a;  public:      B(int );  };    B::B(int x):a(x) {  //Initializer list must be used      cout << "B's Constructor called";  }    int main() {      B obj(10);      return 0;  }  /\* OUTPUT:      A's Constructor called: Value of i: 10      B's Constructor called  \*/ |

If class A had both default and parameterized constructors, then Initializer List is not must if we want to initialize “a” using default constructor, but it is must to initialize “a” using parameterized constructor.

**4) For initialization of base class members :** Like point 3, parameterized constructor of base class can only be called using Initializer List.

|  |
| --- |
| #include <iostream>  using namespace std;    class A {      int i;  public:      A(int );  };    A::A(int arg) {      i = arg;      cout << "A's Constructor called: Value of i: " << i << endl;  }    // Class B is derived from A  class B: A {  public:      B(int );  };    B::B(int x):A(x) { //Initializer list must be used      cout << "B's Constructor called";  }    int main() {      B obj(10);      return 0;  } |

**5) When constructor’s parameter name is same as data member**  
If constructor’s parameter name is same as data member name then the data member must be initialized either using [this pointer](http://msdn.microsoft.com/en-us/library/y0dddwwd.aspx) or Initializer List. In the following example, both member name and parameter name for A() is “i”.

|  |
| --- |
| #include <iostream>  using namespace std;    class A {      int i;  public:      A(int );      int getI() const { return i; }  };    A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used  /\* The above constructor can also be written as  A::A(int i) {      this->i = i;  }  \*/    int main() {      A a(10);      cout<<a.getI();      return 0;  }  /\* OUTPUT:      10  \*/ |

**6) For Performance reasons:**  
It is better to initialize all class variables in Initializer List instead of assigning values inside body. Consider the following example:

|  |
| --- |
| // Without Initializer List  class MyClass {      Type variable;  public:      MyClass(Type a) {  // Assume that Type is an already                       // declared class and it has appropriate                       // constructors and operators        variable = a;      }  }; |

Here compiler follows following steps to create an object of type MyClass  
1. Type’s constructor is called first for “a”.  
2. The assignment operator of “Type” is called inside body of MyClass() constructor to assign

variable = a;

3. And then finally destructor of “Type” is called for “a” since it goes out of scope.

Now consider the same code with MyClass() constructor with Initializer List

|  |
| --- |
| // With Initializer List  class MyClass {      Type variable;  public:      MyClass(Type a):variable(a) {   // Assume that Type is an already                       // declared class and it has appropriate                       // constructors and operators      }  }; |

With the Initializer List, following steps are followed by compiler:  
1. Copy constructor of “Type” class is called to initialize : variable(a). The arguments in initializer list are used to copy construct “variable” directly.  
2. Destructor of “Type” is called for “a” since it goes out of scope.

As we can see from this example if we use assignment inside constructor body there are three function calls: constructor + destructor + one addition assignment operator call. And if we use Initializer List there are only two function calls: copy constructor + destructor call. See [this](https://www.geeksforgeeks.org/archives/23018) post for a running example on this point.

|  |
| --- |
| Question 16 |

Predict the output of following program?

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class Test  {  private:      int x;  public:      Test(int i)      {          x = i;          cout << "Called" << endl;      }  };    int main()  {      Test t(20);      t = 30; // conversion constructor is called here.      return 0;  } | |
| A | Compiler Error |
| B | Called  Called |
| C | Called |

Explanation:

If a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows automatic conversion to the class being constructed. A conversion constructor can be called anywhere when the type of single argument is assigned to the object. The output of the given program is

Called

Called

|  |
| --- |
| Question 17  Wrong |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;    class Test  {  public:     Test(Test &t) { }     Test()        { }  };    Test fun()  {      cout << "fun() Calledn";      Test t;      return t;  }    int main()  {      Test t1;      Test t2 = fun();      return 0;  } | |
| A | fun() Called | |
| B | Empty Output | |
| C | Compiler Error: Because copy constructor argument is non-const | |

Explanation:

# Why copy constructor argument should be const in C++?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.   
One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that [copy elision](https://www.geeksforgeeks.org/copy-elision-in-c/) is not done by compiler.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {     /\* Class data members \*/  public:     Test(Test &t) { /\* Copy data members from t\*/}     Test()        { /\* Initialize data members \*/ }  };    Test fun()  {      cout << "fun() Called\n";      Test t;      return t;  }    int main()  {      Test t1;      Test t2 = fun();      return 0;  } |

Output:

Compiler Error in line "Test t2 = fun();"

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

|  |
| --- |
| Test(const Test &t) { cout << "Copy Constructor Called\n"; } |

Or if we change the line “Test t2 = fun();” to following two lines, then also the program works fine.

|  |
| --- |
| Test t2;  t2 = fun(); |

The function fun() returns by value. So the compiler creates a temporary object which is copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). **The reason for compiler error is, compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn’t make sense to modify compiler created temporary objects as they can die any moment**.

**Private Destructor**

**Also read :** [Can a constructor be private in C++ ?](https://www.geeksforgeeks.org/can-constructor-private-cpp/)  
**Predict the output of following programs.**

|  |
| --- |
| // CPP program to illustrate  // Private Destructor  #include <iostream>  using namespace std;    class Test {  private:      ~Test() {}  };  int main()  {  } |

The above program compiles and runs fine. Hence, we can say that : It is **not** compiler error to create private destructors.

Now, What do you say about below program.

|  |
| --- |
| // CPP program to illustrate  // Private Destructor  #include <iostream>  using namespace std;    class Test {  private:      ~Test() {}  };  int main()  {      Test t;  } |

The above program fails in compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private.  
**Now, What about the below program?**

|  |
| --- |
| // CPP program to illustrate  // Private Destructor  #include <iostream>  using namespace std;    class Test {  private:      ~Test() {}  };  int main()  {      Test\* t;  } |

The above program works fine. There is no object being constructed, the program just creates a pointer of type “Test \*”, so nothing is destructed.

**Next, What about the below program?**

|  |
| --- |
| // CPP program to illustrate  // Private Destructor    #include <iostream>  using namespace std;    class Test {  private:      ~Test() {}  };  int main()  {      Test\* t = new Test;  } |

The above program also works fine. When something is created using dynamic memory allocation, it is programmer’s responsibility to delete it. So compiler doesn’t bother.

**In the case where the destructor is declared private, an instance of the class can also be created using malloc() function.** Same is implemented in below program.

|  |
| --- |
| // CPP program to illustrate  // Private Destructor    #include <bits/stdc++.h>  using namespace std;    class Test {  public:      Test() // Constructor      {          cout << "Constructor called\n";      }    private:      ~Test() // Private Destructor      {          cout << "Destructor called\n";      }  };    int main()  {      Test \* t = (Test \*)malloc(sizeof(Test));  cout << "Main End...\n";      return 0;  } |

Output:

Main End...

However, The below program fails in compilation. When we call delete, destructor is called.

|  |
| --- |
| // CPP program to illustrate  // Private Destructor  #include <iostream>  using namespace std;    class Test {  private:      ~Test() {}  };  int main()  {      Test\* t = new Test;      delete t;  } |

We noticed in the above programs, when a class has private destructur, only dynamic objects of that class can be created. Following is a way to **create classes with private destructors and have a function as friend of the class.** The function can only delete the objects.

|  |
| --- |
| // CPP program to illustrate  // Private Destructor  #include <iostream>    // A class with private destuctor  class Test {  private:      ~Test() {}      friend void destructTest(Test\*);  };    // Only this function can destruct objects of Test  void destructTest(Test\* ptr)  {      delete ptr;  }    int main()  {      // create an object      Test\* ptr = new Test;        // destruct the object      destructTest(ptr);        return 0;  } |

**What is the use of private destructor?**

Whenever we want to control destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling. See [this](http://blogs.msdn.com/b/larryosterman/archive/2005/07/01/434684.aspx) for more details.

|  |
| --- |
| Question 2 |

Predict the output of following C++ progran

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;    int i;    class A  {  public:      ~A()      {          i=10;      }  };    int foo()  {      i=3;      A ob;      return i;  }    int main()  {      cout << foo() << endl;      return 0;  } | |
| A | 0 |
| B | 3 |
| C | 10 |
| D | None of the above |

Explanation:

Why the output is i= 3 and not 10?

While returning from a function, destructor is the last method to be executed. The destructor for the object “ob” is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3. See [this](http://www.geeksforgeeks.org/playing-with-destructors-in-c/) for more details.

How to make the program to output “i = 10” ?

*1) Return by Reference:*  
Since reference gives the l-value of the variable,by using return by reference the program will output “i = 10”.

|  |
| --- |
| #include <iostream>  using namespace std;    int i;    class A  {  public:      ~A()      {          i = 10;      }  };    int& foo()  {      i = 3;      A ob;      return i;  }    int main()  {      cout << "i = " << foo() << endl;      return 0;  } |

The function foo() returns the l-value of the variable i. So, the address of i will be copied in the return value. Since, the references are automatically dereferened. It will output “i = 10”.

*2. Create the object ob in a block scope*

|  |
| --- |
| #include <iostream>  using namespace std;    int i;    class A  {  public:      ~A()      {          i = 10;      }  };    int foo()  {      i = 3;      {          A ob;      }      return i;  }    int main()  {      cout << "i = " << foo() << endl;      return 0;  } |

Since the object ob is created in the block scope, the destructor of the object will be called after the block ends, thereby changing the value of i to 10. Finally 10 will copied to the return value.

Question 3

Like constructors, can there be more than one destructors in a class?

|  |  |
| --- | --- |
| A | Yes |
| B | No |

3 Explanation:

There can be only one destructor in a class. Destructor's signature is always *~ClassNam()* and they cannot be passed arguments

|  |
| --- |
| Question 4 |

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class A  {      int id;      static int count;  public:      A() {          count++;          id = count;          cout << "constructor for id " << id << endl;      }      ~A() {          cout << "destructor for id " << id << endl;      }  };    int A::count = 0;    int main() {      A a[3];      return 0;  } | |
| A | constructor for id 1  constructor for id 2  constructor for id 3  destructor for id 3  destructor for id 2  destructor for id 1 |
| B | constructor for id 1  constructor for id 2  constructor for id 3  destructor for id 1  destructor for id 2  destructor for id 3 |
| C | Compiler Dependent. |
| D | constructor for id 1  destructor for id 1 |

Explanation:

In the above program, id is a static variable and it is incremented with every object creation. Object a[0] is created first, but the object a[2] is destroyed first. Objects are always destroyed in reverse order of their creation. The reason for reverse order is, an object created later may use the previously created object. For example, consider the following code snippet.

A a;

B b(a);

In the above code, the object ‘b’ (which is created after ‘a’), may use some members of ‘a’ internally. So destruction of ‘a’ before ‘b’ may create problems. Therefore, object ‘b’ must be destroyed before ‘a’.