**Question 1**

|  |
| --- |
| #include<iostream>    using namespace std;  class P {  public:     void print()     { cout <<" Inside P::"; }  };    class Q : public P {  public:     void print()     { cout <<" Inside Q"; }  };    class R: public Q {  };    int main(void)  {    R r;      r.print();    return 0;  } |

Output:  
*Inside Q*

The print function is not defined in class R. So it is looked up in the inheritance hierarchy. *print()* is present in both classes *P* and *Q*, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance heirarchy until a matching function is found

**Question 2**

|  |
| --- |
| #include<iostream>  #include<stdio.h>    using namespace std;    class Base  {  public:    Base()    {      fun(); //note: fun() is virtual    }    virtual void fun()    {      cout<<"\nBase Function";    }  };    class Derived: public Base  {  public:    Derived(){}    virtual void fun()    {      cout<<"\nDerived Function";    }  };    int main()  {    Base\* pBase = new Derived();    delete pBase;    return 0;  } |

Output:  
*Base Function*

See following excerpt from [C++ standard](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1905.pdf) for explanation.

*When a virtual function is called directly or indirectly from a constructor (including from the mem-initializer for a data member) or from a destructor, and the object to which the call applies is the object under construction or destruction, the function called is the one defined in the constructor or destructor’s own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor’s class, or overriding it in one of the other base classes of the most derived object.*

Because of this difference in behavior, it is recommended that object’s virtual function is not invoked while it is being constructed or destroyed. See [this](https://www.securecoding.cert.org/confluence/display/cplusplus/OOP30-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors) for more details.

**Question 3**

|  |
| --- |
| #include<iostream>  using namespace std;    class Test {      int value;  public:      Test(int v);  };    Test::Test(int v) {      value = v;  }    int main() {      Test t[100];      return 0;  } |

Output:

Compiler error

The class Test has one user defined constructor “Test(int v)” that expects one argument. It doesn’t have a constructor without any argument as the compiler doesn’t create the default constructor if user defines a constructor (See [this](https://www.geeksforgeeks.org/g-fact-26/)). Following modified program works without any error.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test {      int value;  public:      Test(int v = 0);  };    Test::Test(int v) {      value = v;  }    int main() {      Test t[100];      return 0;  } |

**Question 4**

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  public:    Test();  };    Test::Test()  {      cout<<"Constructor Called \n";  }    int main()  {      cout<<"Start \n";      Test t1();      cout<<"End \n";      return 0;  } |

Output:

Start

End

Note that the line “Test t1();” is not a constructor call. Compiler considers this line as declaration of function t1 that doesn’t recieve any parameter and returns object of type Test.

**Question 5**

|  |
| --- |
| #include<iostream>    using namespace std;    class Test {      int value;  public:      Test (int v = 0) {value = v;}      int getValue() { return value; }  };    int main() {      const Test t;      cout << t.getValue();      return 0;  } |

Output: Compiler Error.

A const object cannot call a non-const function. The above code can be fixed by either making getValue() const or making t non-const. Following is modified program with getValue() as const, it works fine and prints 0.

|  |
| --- |
| #include<iostream>    using namespace std;    class Test {      int value;  public:      Test (int v = 0) { value = v; }      int getValue() const { return value; }  };    int main() {      const Test t;      cout << t.getValue();      return 0;  } |

**Question 6**

|  |
| --- |
| #include<iostream>    using namespace std;    class Test {      int &t;  public:      Test (int &x) { t = x; }      int getT() { return t; }  };    int main()  {      int x = 20;      Test t1(x);      cout << t1.getT() << " ";      x = 30;      cout << t1.getT() << endl;      return 0;  } |

Output: Compiler Error  
Since t is a reference in Test, it must be initialized using Initializer List. Following is the modified program. It works and prints “20 30”.

|  |
| --- |
| #include<iostream>    using namespace std;    class Test {      int &t;  public:      Test (int &x):t(x) {  }      int getT() { return t; }  };    int main() {      int x = 20;      Test t1(x);      cout << t1.getT() << " ";      x = 30;      cout << t1.getT() << endl;      return 0;  } |

**Question 7**

|  |
| --- |
| class Test1 {      int y;  };    class Test2 {      int x;      Test1 t1;  public:      operator Test1() { return t1; }      operator int() { return x; }  };    void fun ( int x)  { };  void fun ( Test1 t ) { };    int main() {      Test2 t;      fun(t);      return 0;  } |

Output: Compiler Error  
There are two conversion operators defined in the Test2 class. So Test2 objects can automatically be converted to both int and Test1. Therefore, the function call fun(t) is ambiguous as there are two functions void fun(int ) and void fun(Test1 ), compiler has no way to decide which function to call. In general, conversion operators must be overloaded carefully as they may lead to ambiguity.

**Question 8**

|  |
| --- |
| #include <iostream>  using namespace std;    class X {  private:    static const int a = 76;  public:    static int getA() { return a; }  };    int main() {    cout <<X::getA()<<endl;    return 0;  } |

Output: The program compiles and prints 76  
Generally, it is not allowed to initialize data members in C++ class declaration, but static const integral members are treated differently and can be initialized with declaration.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class.

**Question 9**

|  |
| --- |
| template <class S, class T> class Pair  {  private:      S x;      T y;  /\* ... \*/  };    template <class S> class Element  {  private:      S x;  /\* ... \*/  };    int main ()  {      Pair <Element<int>, Element<char>> p;      return 0;  } |

Output:

Compiler Error: '>>' should be '> >' within a nested template argument list

When we use nested templates in our program, we must put a space between two closing angular brackets, otherwise it conflicts with operator >>. For example, following program compiles fine.

|  |
| --- |
| template <class S, class T> class Pair  {  private:      S x;      T y;  /\* ... \*/  };    template <class S> class Element  {  private:      S x;  /\* ... \*/  };    int main ()  {      Pair <Element<int>, Element<char> > p;   // note the space between '>' and '>'      return 0;  } |

**Question 10**

|  |
| --- |
| #include<iostream>  using namespace std;    class Point  {  private:      int x;      int y;  public:      Point(const Point&p) { x = p.x; y = p.y; }      void setX(int i) {x = i;}      void setY(int j) {y = j;}      int getX() {return x;}      int getY() {return y;}      void print() { cout << "x = " << getX() << ", y = " << getY(); }  };  int main()  {      Point p1;      p1.setX(10);      p1.setY(20);      Point p2 = p1;      p2.print();      return 0;  } |

Output: Compiler Error in first line of main(), i.e., “Point p1;”

Since there is a user defined constructor, compiler doesn’t create the default constructor (See [this GFact](https://www.geeksforgeeks.org/archives/8316)). If we remove the copy constructor from class Point, the program works fine and prints the output as “x = 10, y = 20”

**Question 11**

|  |
| --- |
| #include <iostream>  using namespace std;    class Fraction  {  private:      int den;      int num;  public:     void print() { cout << num << "/" << den; }     Fraction() { num = 1; den = 1; }     int &Den() { return den; }     int &Num() { return num; }  };    int main()  {     Fraction f1;     f1.Num() = 7;     f1.Den() = 9;     f1.print();     return 0;  } |

Output: 7/9  
The methods Num() and Den() return references to num and den respectively. Since references are returned, the returned values can be uses as an lvalue, and the private members den and num are modified. The program compiles and runs fine, but this kind of class design is strongly discouraged (See [this](https://www.securecoding.cert.org/confluence/display/cplusplus/OOP35-CPP.+Do+not+return+references+to+private+data)). Returning reference to private variable allows users of the class to change private data directly which defeats the purpose of encapsulation.

**Question 12**

|  |
| --- |
| #include<iostream>  using namespace std;    /\* local variable is same as a member's name \*/  class Test  {  private:      int x;  public:      void setX (int x) { Test::x = x; }      void print() { cout << "x = " << x << endl; }  };    int main()  {      Test obj;      int x = 40;      obj.setX(x);      obj.print();      return 0;  } |

Output:

x = 40

Scope resolution operator can always be used to access a class member when it is made hidden by local variables. So the line “Test::x = x” is same as “this->x = x”

Predict the output of following C++ program.

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {      // data members of A  public:      A ()           { cout << "\n A's constructor"; /\* Initialize data members \*/ }      A (const A &a) { cout << "\n A's Copy constructor";  /\* copy data members \*/}      A& operator= (const A &a) // Assignemt Operator      {          // Handle self-assignment:          if(this == &a) return \*this;            // Copy data members          cout << "\n A's Assignment Operator";  return \*this;      }  };    class B  {      A a;      // Other members of B  public:      B(A &a) { this->a = a; cout << "\n B's constructor"; }  };    int main()  {      A a1;      B b(a1);      return 0;  } |

Output:

A's constructor

A's constructor

A's Assignment Operator

B's constructor

The first line of output is printed by the statement “A a1;” in main().  
The second line is printed when B’s member ‘a’ is initialized. This is important.  
The third line is printed by the statement “this->a = a;” in B’s constructor.  
The fourth line is printed by cout statement in B’s constructor.

If we take a closer look at the above code, the constructor of class B is not efficient as member ‘a’ is first constructed with default constructor, and then the values from the parameter are copied using assignment operator. It may be a concern when class A is big, which generally is the case with many practical classes. See the following optimized code.

|  |
| --- |
| #include<iostream>  using namespace std;    class A  {      // data members of A  public:      A()           { cout << "\n A's constructor"; /\* Initialize data members \*/ }      A(const A &a) { cout << "\n A's Copy constructor"; /\* Copy data members \*/ }      A& operator= (const A &a) // Assignemt Operator      {          // Handle self-assignment:          if(this == &a) return \*this;            // Copy data members          cout << "\n A's Assignment Operator";  return \*this;      }  };    class B  {      A a;      // Other members of B  public:      B(A &a):a(a) {  cout << "\n B's constructor"; }  };    int main()  {      A a;      B b(a);      return 0;  } |

Output:

A's constructor

A's Copy constructor

B's constructor

The constructor of class B now uses initializer list to initialize its member ‘a’. When Initializer list is used, the member ‘a’ of class B is initialized directly from the parameter. So a call to A’s constructor is reduced.

**Question 13**

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {      int id;  public:      A (int i) { id = i; }      void print () { cout << id << endl; }  };    int main()  {      A a[2];      a[0].print();      a[1].print();      return 0;  } |

There is a compilation error in line “A a[2]”. There is no default constructor in class A. When we write our own parameterzied constructor or copy constructor, compiler doesn’t create the default constructor (See [this Gfact](https://www.geeksforgeeks.org/archives/8316)). We can fix the error, either by creating a default constructor in class A, or by using the following syntax to initialize array member using parameterzied constructor.

// Initialize a[0] with value 10 and a[1] with 20

A a[2] = { A(10), A(20) }

**Question 14**

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {     int aid;  public:     A(int x)     { aid = x; }     void print()     { cout << "A::aid = " <<aid; }  };    class B  {      int bid;  public:      static A a;      B (int i) { bid = i; }  };    int main()  {    B b(10);    b.a.print();    return 0;  } |

Compiler Error: undefined reference to `B::a’  
The class B has a static member ‘a’. Since member ‘a’ is static, it must be defined outside the class. Class A doesn’t have Default constructor, so we must pass a value in definition also. Adding a line “A B::a(10);” will make the program work.

The following program works fine and produces the output as “A::aid = 10”

|  |
| --- |
| #include <iostream>  using namespace std;    class A  {     int aid;  public:     A(int x)     { aid = x; }     void print()     { cout << "A::aid = " <<aid; }  };    class B  {      int bid;  public:      static A a;      B (int i) { bid = i; }  };    A B::a(10);    int main()  {    B b(10);    b.a.print();    return 0;  } |

**Question 15 -- Imp**

|  |
| --- |
| #include<iostream>  using namespace std;    class base  {  public:      virtual void show()  { cout<<" In Base \n"; }  };    class derived: public base  {      int x;  public:      void show() { cout<<"In derived \n"; }      derived()   { x = 10; }      int getX() const { return x;}  };    int main()  {      derived d;      base \*bp = &d;      bp->show();      cout << bp->getX();      return 0;  } |

Output: Compiler Error: ‘class base’ has no member named ‘getX’  
In the above program, there is pointer ‘bp’ of type ‘base’ which points to an object of type derived. The call of show() through ‘bp’ is fine because ‘show()’ is present in base class. In fact, it calls the derived class ‘show()’ because ‘show()’ is virtual in base class. But the call to ‘getX()’ is invalid, because getX() is not present in base class. When a base class pointer points to a derived class object, it can access only those methods of derived class which are present in base class and are virtual.

**Question 16**

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public:      int fun()      { cout << "Base::fun() called"; }      int fun(int i) { cout << "Base::fun(int i) called"; }  };    class Derived: public Base  {  public:      int fun(char x)   { cout << "Derived::fun(char ) called"; }  };    int main()  {      Derived d;      d.fun();      return 0;  } |

Output: Compiler Error.  
In the above program, fun() of base class is not accessible in the derived class. If a derived class creates a member method with name same as one of the methods in base class, then all the base class methods with this name become hidden in derived class

## Hiding of all overloaded methods with same name in base class

In C++, if a derived class redefines base class member method then all the base class methods with same name become hidden in derived class.   
For example, the following program doesn’t compile. In the following program, Derived redefines Base’s method fun() and this makes fun(int i) hidden.

|  |
| --- |
| #include<iostream>    using namespace std;    class Base  {  public:      int fun()      {          cout<<"Base::fun() called";      }      int fun(int i)      {          cout<<"Base::fun(int i) called";      }  };    class Derived: public Base  {  public:      int fun()      {          cout<<"Derived::fun() called";      }  };    int main()  {      Derived d;      d.fun(5);  // Compiler Error      return 0;  } |

Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden. For example, in the following program, Derived::fun(char ) makes both Base::fun() and Base::fun(int ) hidden.

|  |
| --- |
| #include<iostream>    using namespace std;    class Base  {  public:      int fun()      {          cout<<"Base::fun() called";      }      int fun(int i)      {          cout<<"Base::fun(int i) called";      }  };    class Derived: public Base  {  public:      int fun(char c)  // Makes Base::fun() and Base::fun(int ) hidden      {          cout<<"Derived::fun(char c) called";      }  };    int main()  {      Derived d;      d.fun();  // Compiler Error      return 0;  } |

Note that the above facts are true for both static and nonstatic methods.

# ****Question 17****

|  |
| --- |
| #include<iostream>  using namespace std;  class Base  {     protected:        int x;     public:        Base (int i){ x = i;}  };    class Derived : public Base  {     public:        Derived (int i):x(i) { }        void print() { cout << x ; }  };    int main()  {      Derived d(10);      d.print();  } |

Output: Compiler Error  
In the above program, x is protected, so it is accessible in derived class. Derived class constructor tries to use [initializer list](https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/) to directly initialize x, which is not allowed even if x is accessible. The members of base class can only be initialized through a constructor call of base class. Following is the corrected program.

|  |
| --- |
| #include<iostream>  using namespace std;  class Base {     protected:        int x;     public:        Base (int i){ x = i;}  };    class Derived : public Base {     public:        Derived (int i):Base(i) { }        void print() { cout << x; }  };    int main()  {      Derived d(10);      d.print();  } |

Output:

10

# Question 18

|  |
| --- |
| #include<stdlib.h>  #include<iostream>    using namespace std;    class Test {  public:      void\* operator new(size\_t size);      void operator delete(void\*);      Test() { cout<<"\n Constructor called"; }      ~Test() { cout<<"\n Destructor called"; }  };    void\* Test::operator new(size\_t size)  {      cout<<"\n new called";      void \*storage = malloc(size);      return storage;  }    void Test::operator delete(void \*p )  {      cout<<"\n delete called";      free(p);  }    int main()  {      Test \*m = new Test();      delete m;      return 0;  } |

new called

Constructor called

Destructor called

delete called

Let us see what happens when below statement is executed.

Test \*x = new Test;

When we use new keyword to dynamically allocate memory, two things happen: memory allocation and constructor call. The memory allocation happens with the help of operator new. In the above program, there is a user defined operator new, so first user defined operator new is called, then constructor is called.  
The process of destruction is opposite. First, destructor is called, then memory is deallocated.

# Question 19 – Template: To Do

|  |
| --- |
| #include <iostream>  using namespace std;    template <int N>  class A {     int arr[N];  public:     virtual void fun() { cout << "A::fun()"; }  };    class B : public A<2> {  public:     void fun() { cout << "B::fun()"; }  };    class C : public B { };    int main() {     A<2> \*a = new C;     a->fun();     return 0;  } |

Output:

B::fun()

In general, the purpose of using templates in C++ is to avoid code redundancy.  We create generic classes (or functions) that can be used for any datatype as long as logic is identical. Datatype becomes a parameter and an instance of class/function is created at compile time when a data type is passed. C++ Templates also allow nontype (a parameter that represents a value, not a datatype) things as parameters.  
In the above program, there is a generic class A which takes a nontype parameter N. The class B inherits from an instance of generic class A. The value of N for this instance of A is 2. The class B overrides fun() of class A<2>. The class C inherits from B. In main(), there is a pointer ‘a’ of type A<2> that points to an instance of C. When ‘a->fun()’ is called, the function of class B is executed because fun() is virtual and virtual functions are called according to the actual object, not according to pointer. In class C, there is no function ‘fun()’, so it is looked up in the hierarchy and found in class B.

# Question 20 - create an object of class without any reference pointing to it

|  |
| --- |
| #include <iostream>  using std::cout;  class Test  {  public:      Test();      ~Test();  };  Test::Test()  {      cout << "Constructor is executed\n";  }  Test::~Test()  {      cout << "Destructor is executed\n";  }  int main()  {      delete new Test();      return 0;  } |

Output:

Constructor is executed

Destructor is executed

The first statement inside the main () function looks strange, but it is perfectly valid. It is possible to create an object without giving its handle to any pointer in C++. This statement will create an object of class Test without any pointer pointing to it. This can be also done in languages like Java & C#.  
For example consider following statement:

new student(); // valid both in Java & C#

The above statement will create an object of student class without any reference pointing to it.

# **Question 21** - Main class and object in main()

|  |
| --- |
| #include <iostream>  using std::cout;  class main  {  public:      main()  {cout << "ctor is called\n";}      ~main() {cout << "dtor is called\n";}  };  int main()  {      main m;    // LINE 11  } |

Output:

Compiler error:

11 8 [Error] expected ';' before 'm'

The above program looks syntactically correct but it fails in compilation. The reason class name. Class name is main so it is necessary to tell the compiler that main is the name of class. Generally struct or class keyword is not required to write to create an object of the class or struct. But when the name of class is main it becomes necessary to write struct or class when creating object of class or struct. Remember main is not a reserved word.

Following is a correct version of the above program:

|  |
| --- |
| #include <iostream>  using std::cout;  class main  {  public:     main()  { cout << "ctor is called\n";}     ~main() { cout << "dtor is called\n";}  };  int main()  {     class main m;  } |

Now predict the output of following program:

|  |
| --- |
| #include <iostream>  using std::cout;  class main  {  public:     main()  { cout << "ctor is called\n"; }     ~main() { cout << "dtor is called\n"; }  };  main m;    // Global object  int main()  {  } |

The above program compiles and runs fine because object is global. Global object‘s constructor executes before main() function and it’s destructor executes when main() terminates.

Conclusion: When the class/struct name is main and whenever the local object is created it is mandatory to write class or struct when the object of class / and struct is created. Because C++ program execution begins from main () function. But this rule is not applied to global objects. Again, main isn’t a keyword but treat it as if it were.

# Question 22 – Logical AND and OR

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {      int x = 1 , y = 1, z = 1;        cout << (++x || ++y && ++z ) << endl;        cout << x << " " << y << " " << z ;        return 0;  } |

**Explanation:-**  
It is based on fact that how LOGICAL – OR and LOGICAL-AND work. Note that Compiler reads OR and AND operators from left to right. Let us take the following cases into consideration:-

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {      int x = 1 , y = 1;      cout << ( ++x  || ++y ) << endl;   // outputs 1;      cout << x << " " << y;             // x = 2 , y = 1;      return 0;  } |

Output:

1

2 1

Once compiler detects “true” on the LEFT of logical OR, IT IS NOT GOING TO EVALUATE THE RIGHT SIDE!, because even one is true, the whole “OR” expression becomes true!. SO compiler skips the RIGHT part and displays the result as 1 !So y is not incremented here , because compiler skipped reading it!

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {      int x = 1 , y = 1;      cout << ( ++x && ++y ) << endl;     //outputs 1;      cout << x << " " << y;              // x = 2 , y = 2;      return 0;  } |

Output:

1

2 2

LOGICAL AND needs to evaluate both right and left part (Think about it !)So both left and right part is evaluated, thus incrementing both x and y here.

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {      int x = 1 , y = 1, z = 1;      cout << ( ++x || ++y && ++z ) << endl;    //outputs 1;      cout << x << " " << y << " " << z ;       //x = 2 , y = 1 , z = 1;      return 0;  } |

Output:

1

2 1 1

Here compiler increments x first and then it detects a LOGICAL OR. We have a true quantity on left side . SO compiler won’t read the right part.Thus incrementing x and y,z remains same!

# Question 23

|  |
| --- |
| #include <iostream>  using namespace std;    int main()  {      int a = b = c = 0;      cout << a << "\*" << b << "\*" << c;      return 0;  } |

Output:

Compile time error!

**Explanation:**  
A chained statement cannot be used to initialize variables at the time of declaration. Hence the statement a = b = c = 0; is an illegal statement. However following way a legal syntax and can be used in C++ or C.

int a,b,c;

a = b = c = 0;

# Question 24

|  |
| --- |
| #include <iostream>  using namespace std;    int main()  {      int i;      i = 1 + (1,4,5,6,3);      cout << i;      return 0;  } |

Output:

4

**Explanation:**  
The comma (,) operator is a binary operator that evaluates the first operand and discards the result and then evaluates the second and then returns the value of second. Here the associativity of the comma (,) operator is from left to right and is easy to understand that the expression(1,4,5,6,3) evaluates to be 3 and then the result 1 + 3 is assigned to i.

# Question 25

|  |  |
| --- | --- |
| |  | | --- | | #include <iostream> |   using namespace std;  int a = 90;    int fun(int x, int \*y = &a)  {      \*y = x + \*y;      return x + \*y;  }    int main()  {      int a = 5, b = 10;        a = fun(a);      cout << a << " " << b << endl;        b = fun(::a,&a);      cout << a << " " << b << endl;        return 0;  } |

100 10

195 290

There are two variables with name ‘a’, one is global and other is local. When we call **a = fun(a);**, it calls int fun(int x, int \*y=&a), here pointer to global variable (which is a = 90) is assigned to y. Therefore.  
\*y = x + \*y; // becomes 5 + 90  
return x + \*y; // becomes 5 + 95

# Question 26

|  |
| --- |
| #include <iostream>  using namespace std;    int main()  {      int a = 0, b;      b = (a = 50) + 10;      cout << a << "$" << b;      return 0;  } |

Output:

50$60

**Explanation:**  
The statement b = (a = 50) + 10; uses the concept of embedded assignment. Here, the value of 50 is assigned to variable a and the result 50+10 is assigned to b.

# Question 27

|  |
| --- |
| #include<iostream>  using namespace std;    int main(int x)  {      static int i = 5;        if (--i)      {          cout << i;          main(10);      }      return 0;  } |

Output:

4321

**Explanation:**  
Any non-zero number in C++ is treated as true value. Here in this code the if statement tends to decrease the value of i but inside the block of if, the main() function is called again and again. Here the program seems to be in infinite loop but the variable i is static in nature it has its lifetime till execution hence the program will halt if i becomes 0.

# Question 28 – C vs C++ : character array initialization

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      char array[3] = "abc";      cout << array;      return 0;  } |

Output:

error: initializer - string for array of chars is too long

**Explanation:** The application of array in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing character array in ANSI C, the compiler will allow the following declaration —

char array[3] = "abc" // allowed in ANSI C

This is because C assumes that the programmer intends to leave out the NULL (‘\0’) character in the definition. But in C++ the size should be 1 larger then the number of characters hence following statement is correct.

char array[4] = "abc" // O.K. for C++

# Question 29 – C vs C++ : sizeof of a character constant

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      cout << sizeof('x');      cout << sizeof(char);      return 0;  } |

Output:

1 1

**Explanation:** It is notable that character constants are stored as character type in C++ but this is not the case in C. In ANSI C, the same program would produce the result —

4 1 // result when code is executed in C

because character constants are promoted to int.

# Question 30 – C vs C++ : Void pointer conversion

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      void \*ptr1;      char \*ptr2;      ptr2 = ptr1; // statement 1      return 0;  } |

Output:

error: invalid conversion from 'void\*' to 'char\*

**Explanation:** Assigning any pointer type to a void pointer without using a cast is allowed in C or C++. But we can not assign a void pointer to a non-void pointer without using a cast to non-void pointer type in C++. Hence statement one should be —

ptr2 = (char\*)ptr1; // valid in C++

**Note —** It should be noted that when the same code is compiled in C, it will not produce any error i.e. in C we can assign a void pointer to a non-void pointer without using any type casting.

# Question 31 – C vs C++ : Const variable declaration

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      const int size;      cout << size;      return 0;  } |

Output:

error: uninitialized const 'size'

**Explanation:** At first glance its obvious to think that output will be some garbage value, but it is not the case. Unlike C in C++ any variable declared as constant needs to be initialized at the time of declaration. Hence the given code will not compile and throw the error. It should be noted that the same code will get compiled in ANSI C and will produce output as 0.

# Program 32 – C vs C++ : ENUM

|  |
| --- |
| #include <iostream>  using namespace std;  int main()  {      enum season { spring, summer, autumn, winter };        season myFavSeason\_1 = spring;      season myFavSeason\_2 = 4;  // statement 1        cout << myFavSeason\_1;      cout << endl;      cout << myFavSeason\_2;        return 0;  } |

Output:

error: invalid conversion from 'int' to 'main()::season'

season myFavSeason\_2 = 4;

**Explanation:** Unlike C in C++, each enumerated data type retains its own separate type. This means C++ does not permit an int value to be automatically converted to an enum value. So there will be error in executing the code. However if type casting is done the code will work change statement 1 to the following statement —

season myFavSeason\_2 = (season) 4;