FRESHERS NOW

Texas Instruments Technical Questions and Answers

Q1. What are manipulators?

ANS:

Manipulators are the instructions to the output stream to modify the output in various ways. The manipulators provide a clean and easy way for formatted output in comparison to the formatting flags of the IOS class. When manipulators are used, the formatting instructions are inserted directly into the stream.

Q2. What are put and get pointers?

ANS:

These are the long integers associated with the streams. The value present in the put pointer specifies the byte number in the file from where next write would take place in the file. The get pointer specifies the byte number in the file from where the next reading should take place.

Q3. What is the purpose of istream class?

ANS:

The istream class performs activities specific to input. It is derived from the IOS class. The most commonly used member function of this class is the overloaded >> operator which can extract values of all basic types. We can extract even a string using this operator.

Q4. Can we use - this- pointer inside static member function?

ANS:

No! The this pointer cannot be used inside a static member function. This is because a static member function is never called through an object.

Q5. What is the difference between an inspector and a mutator?

ANS:

An inspector is a member function that returns information about an objects state (information stored in objects data members) without changing the objects state.
A mutator is a member function that changes the state of an object.

Q6. How would you give an alternate name to a namespace?

ANS:

An alternate name given to namespace is called a namespace-alias. namespace-alias is generally used to save the typing effort when the names of namespaces are very long or complex. The following syntax is used to give an alias to a namespace.
namespace myname = my_old_very_long_name ;

Q7. How name mangling can be prevented?

ANS:

To avoid name mangling the function should be declared with an extern C attribute. Functions declared as extern C are treated as C-style functions. Hence the compiler does not mangle them. The following code snippet shows how to declare such a function.

```
#include
extern C void display( )
{
cout << See the effect of C in C++  ;
}
void main( )
{
display( ) ;
}
```

Q8. What is RTTI?

ANS:

RTTI stands for Run Time Type Information. We use virtual function mechanism where we can call derived class member functions using base class pointer. However, many times we wish to know the exact type of the object. We can know the type of the object using RTTI. A function that returns the type of the object is known as RTTI functions. C++ supports two ways to obtain information about the objects class at run time, they are typeid( ) operator and dynamic_cast operator.

Q9. What is a container?

ANS:

A container is an object that holds other objects. Various collection classes like List, Hash Table, Abstract Array, etc. are the examples of containers. We can use the classes to hold objects of any derived classes. The containers provide various methods using which we can get the number of objects stored in the container and iterate through the objects stored in it.

Q10. What is the disadvantage of a template function?

ANS:

A template function cannot be distributed in the obj form. This is because with which parameters the template function is going to be called is decided at the run time only. Therefore an obj form of a template function cannot be made by merely compiling it.

Q11. When should we use the :: (scope resolution) operator to invoke the virtual functions?

ANS:

Generally, :: operator is used to call a virtual function from constructor or destructor. This is because, if we call a virtual function from base class constructor or destructor the virtual function of the base class would get called even if the object being constructed or destroyed would be the object of the derived class. Thus, whenever we want to bypass the dynamic binding mechanism we must use the :: operator to call a virtual function.

Q12. Can we have a reference to an array?

ANS:

Yes, we can have a reference to an array.

```
int a[ ] = { 8, 2, 12, 9 } ;
```

int (&r) [4] = a ; // reference to an array
Here, r is a reference to an array of four elements. We can even print the elements of array with the help of reference. This is shown in the following code segment:

```
for ( int i = 0 ; i < 4 ; i++ )
cout << r [i] << endl ;
```

Q13. Explain exception Handling in C++.

ANS:

In C++ we can handle run-time errors generated by c++ classes by using three new keywords: throw, catch, and try. We also have to create an exception class. If during the course of execution of a member function of this class a run-time error occurs, then this member function informs the application that an error has occurred. This process of informing is called throwing an exception. The following code shows how to deal with exception handling.

```
class sample
{
public :
class errorclass
{
} ;
void fun( )
{
if ( some error occurs )
throw errorclass( ) // throws exception
}
} ;
```

```
//application
void main( )
{
try
{
sample s ;
s.fun( ) ;
}
catch ( sample::errorclass )
{
// do something about the error
}
}
```

Q14. Why creating array of references is not possible?

ANS:

The array name always refers or points to the Zeroth element. If array is of references then the array name would point to the Zeroth element which happens to be a reference. Creating pointer to a reference is not valid. So, creating array of references too is not possible.

Q15. Why an overloaded new operator defined in a class is static?

ANS:

An overloaded new function is by default static even if it is not declared so. This is because non-static member functions can be called through an object only. But when an overloaded new operator function gets called the object does not stand created. Since new operator function itself is responsible for creating the object. Hence to be able to call a function without an object, the function must be static.

Q16. How to restrict the number of floating-point digits displayed?

ANS:

When we display floating-point values, we can use the set precision manipulator to specify the desired number of digits to the right of the decimal point.
For example,

cout << setprecision (3) << 12.34678 ;

This statement would give the output as 12.347.

Q17. When can we use the function ostrstream::freeze( )?

ANS:

While outputting data to memory in the in-memory formatting we need to create an object of the class ostrstream. The constructor of ostrstream receives the address of the buffer but if we want that the ostrstream object should do its own memory management then we need to create an ostrstream object with no constructor arguments as:

ostrstream s ;

Once str( ) has been called then the block of memory allocated by ostrstream cannot be moved. This is logical. It cant move the block since we are now expecting it to be at a particular location. In such a case we say that ostrstream has freezed itself. Once frozen we cant add any more characters to it. Adding characters to a frozen ostrstream results in undefined behavior. In addition, the ostrstream is no longer responsible for cleaning up the storage.

Q18. What are Early Binding and Dynamic Binding?

ANS:

The term binding refers to the connection between a function call and the actual code executed as a result of the call. Early Binding: If which function is to be called is known at the compile-time it is known as static or early binding. Dynamic Binding: If which function is to be called is decided at run time it is called as late or dynamic binding. Dynamic binding is so called because the actual function called at run-time depends on the contents of the pointer. For example, call to virtual functions, call to functions to be linked from dlls use late binding.

Q19. Can we declare a static function as virtual?

ANS:

No. The virtual function mechanism is used on the specific object that determines which virtual function to call. Since the static functions are not any way related to objects, they cannot be declared as virtual.

Q20. What is forward referencing and when should it be used?

ANS:

Consider the following program:

```
class test
{
public :
friend void fun ( sample, test ) ;
} ;
class sample
{
public :
friend void fun ( sample, test ) ;
} ;
void fun ( sample s, test t )
{
// code
}
void main( )
{
sample s ;
test t ;
fun ( s, t ) ;
}
```

This program would not compile. It gives an error that sample is undeclared identifier in the statement friend void fun (sample, test) ; of the class test. This is so because the class sample is defined below the class test and we are using it before its definition. To

overcome this error we need to give forward reference of the class sample before the definition of class test. The following statement is the forward reference of class sample. Forward referencing is generally required when we make a class or a function as a friend.

Q21. We all know that a const variable needs to be initialized at the time of declaration. Then how come the program given below runs properly even when we have not initialized p?

```
#include
void main( )
{
const char *p ;
p = A const pointer ;
cout << p ;
}
```

ANS:

The output of the above program is A const pointer. This is because in this program p is declared as const char* which means that value stored at p will be constant and not p and so the program works properly.

Q22. Can user-defined object be declared as static data member of another class?

ANS:

Yes. The following code shows how to initialize a user-defined object.

```
#include
class test
{
int i ;
public :
test ( int ii = 0 )
{
i = ii ;
```

```
}
} ;
class sample
{
static test s ;
} ;
test sample::s ( 26 ) ;
```

Here we have initialized the object s by calling the one-argument constructor. We can use the same convention to initialize the object by calling multiple-argument constructor.

Q23. How do I write my own zero-argument manipulator that should work same as hex?

ANS:

This is shown in following program.

```
#include
ostream& myhex ( ostream &o )
{
o.setf ( ios::hex) ;
return o ;
}
void main( )
{
cout << endl << myhex << 2000 ;
}
```

Q24. While overloading a binary operator can we provide default values?

ANS:

No! This is because even if we provide the default arguments to the parameters of the overloaded operator function we would end up using the binary operator incorrectly. This is explained in the following example:

```
sample operator + ( sample a, sample b = sample (2, 3.5f ) )
```

```
{
}
void main( )
{
sample s1, s2, s3 ;
s3 = s1 + ; // error
}
```

Q25. Can we get the value of IOS format flags?

ANS:

Yes! The ios::flags( ) member function gives the value format flags. This function takes no arguments and returns a long (typedefed to fmtflags) that contains the current format flags.

Q26. When should I use unitbuf flag?

ANS:

The unit buffering (unitbuf) flag should be turned on when we want to ensure that each character is output as soon as it is inserted into an output stream. The same can be done using unbuffered output but unit buffering provides a better performance than the unbuffered output.

Q27. How do I get the current position of the file pointer?

ANS:

We can get the current position of the file pointer by using the tellp( ) member function of ostream class or tellg( ) member function of istream class. These functions return (in bytes) positions of put pointer and get pointer respectively.

Q28. What is the difference between the manipulator and setf( ) function?

ANS:

The difference between the manipulator and setf( ) function are as follows:

The setf( ) function is used to set the flags of the ios but manipulators directly insert the formatting instructions into the stream. We can create user-defined manipulators but setf( ) function uses data members of ios class only. The flags put on through the setf( ) function can be put off through unsetf( ) function. Such flexibility is not available with manipulators.

Q29. What do the no create and no replace flag ensure when they are used for opening a file?

ANS:

no create and no replace are file-opening modes. A bit in the ios class defines these modes. The flag no create ensures that the file must exist before opening it. On the other hand the flag no replace ensures that while opening a file for output it does not get overwritten with new one unless ate or app is set. When the app flag is set then whatever we write gets appended to the existing file. When ate flag is set we can start reading or writing at the end of existing file.

Q30. Is there any function that can skip certain number of characters present in the input stream?

ANS:

Yes! This can be done using cin::ignore( ) function. The prototype of this function is as shown below:
istream& ignore (int n = 1, int d =EOF) ;
Sometimes it happens that some extra characters are left in the input stream while taking the input such as, the ? ? (Enter) character. This extra character is then passed to the next input and may pose problem.

To get rid of such extra characters the cin::ignore( ) function is used. This is equivalent to fflush (stdin) used in C language. This function ignores the first n characters (if present) in the input stream, stops if delimiter d is encountered.

Q31. Would the following code work?

```
#include
void main( )
{
ostream o ;
o << Dream. Then make it happen! ;
}
```

ANS:

No! This is because we cannot create an object of the ostream class since its constructor and copy constructor are declared private.

Q32. Can we use this pointer in a class specific, operator-overloading function for new operator?

ANS:

No! The this pointer is never passed to the overloaded operator new() member function because this function gets called before the object is created. Hence there is no question of the this pointer getting passed to operator new( ).

Q33. Can we allocate memory dynamically for a reference?

ANS:

No! It is not possible to allocate memory dynamically for a reference. This is because, when we create a reference, it gets tied with some variable of its type. Now, if we try to allocate memory dynamically for a reference, it is not possible to mention that to which variable the reference would get tied.

Q34. Can we allocate memory dynamically for a reference?

ANS:

No, it is not possible to allocate memory dynamically for a reference. A reference is initialized at the time of creation. Trying to allocate memory dynamically for a reference

creates a problem in initializing it. Thus, the compiler does not allow us to dynamically allocate the memory for references.

Q35. What is a pure virtual destructor?

ANS:

Like a pure virtual function we can also have a pure virtual destructor. If a base class contains a pure virtual destructor it becomes necessary for the derived classes to implement the destructor. An ordinary pure virtual function does not have a body but pure virtual destructor must have a body. This is because all the destructors in the hierarchy of inheritance are always called as a part of destruction.

Q36. What is a wild pointer?

ANS:

A wild pointer is the one that points to a garbage value. For example, an uninitialized pointer that contains garbage value or a pointer that refers to something that no longer exists.

Q37. How do I refer to a name of class or function that is defined within a namespace?

ANS:

There are two ways in which we can refer to a name of class or function that is defined within a namespace: Using scope resolution operator through the using keyword. This is shown in following example:

```
namespace name1
{
class sample1
{
// code
} ;
}
namespace name2
```

```
{
class sample2
{
// code
} ;
}
using namespace name2 ;
void main( )
{
name1::sample1 s1 ;
sample2 s2 ;
}
```

Here, class sample1 is referred using the scope resolution operator. On the other hand we can directly refer to class sample2 because of the statement using namespace name2 ; the using keyword declares all the names in the namespace to be in the current scope. So we can use the names without any qualifiers.

Q38. How do I carry out conversion of one object of user-defined type to another?

ANS:

To perform conversion from one user-defined type to another we need to provide conversion function. Following program demonstrates how to provide such conversion function.

```
class circle
{
private :
int radius ;
public:
circle ( int r = 0 )
{
radius = r ;
}
} ;
class rectangle
{
```

```
private :
int length, breadth ;
public :
rectangle( int l, int b )
{
length = l ;
breadth = b ;
}
operator circle( )
{
return circle ( length ) ;
}
} ;
void main( )
{
rectangle r ( 20, 10 ) ;
circle c;
c = r ;
}
```

Here, when the statement c = r ; is executed the compiler searches for an overloaded assignment operator in the class circle which accepts the object of type rectangle. Since there is no such overloaded assignment operator, the conversion operator function that converts the rectangle object to the circle object is searched in the rectangle class.

Q39. How do I write code that allows to create only one instance of a class?

ANS:

This is shown in following code snippet.

```
#include
class sample
{
static sample *ptr ;
private:
sample( )
{
```

```
}
public:
static sample* create( )
{
if ( ptr == NULL )
ptr = new sample ;
return ptr ;
}
} ;
sample *sample::ptr = NULL ;
void main( )
{
sample *a = sample::create( ) ;
sample *b = sample::create( ) ;
}
```

Here, the class sample contains a static data member ptr, which is a pointer to the object of same class. The constructor is private which avoids us from creating objects outside the class. A static member function called create( ) is used to create an object of the class. In this function the condition is checked whether or not ptr is NULL, if it is then an object is created dynamically and its address collected in ptr is returned. If ptr is not NULL, then the same address is returned. Thus, in main( ) on execution of the first statement one object of sample gets created whereas on execution of second statement, b holds the address of the first object. Thus, whatever number of times you call create( ) function, only one object of sample class will be available.

Q40. How do I write code to add functions, which would work as get and put properties of a class?

ANS:

This is shown in following code.

```
#include
class sample
{
int data ;
public:
```

```
__declspec ( property ( put = fun1, get = fun2 ) )
int x ;
void fun1 ( int i )
{
if ( i < 0 )
data = 0 ;
else
data = i ;
}
int fun2( )
{
return data ;
}
} ;
void main( )
{
sample a ;
a.x = -99 ;
cout << a.x ;
}
```

Here, the function fun1( ) of class sample is used to set the given integer value into data, whereas fun2( ) returns the current value of data. To set these functions as properties of a class we have given the statement as shown below:
__declspec (property (put = fun1, get = fun2)) int x ;

As a result, the statement a.x = -99 ; would cause fun1( ) to get called to set the value in data. On the other hand, the last statement would cause fun2( ) to get called to return the value of data.

Q41. How do I write code to make an object work like a 2-D array?

ANS:

Take a look at the following program.

```
#include
class emp
```

```
{
public :
int a[3][3] ;
emp( )
{
int c = 1 ;
for ( int i = 0 ; i <= 2 ; i++ )
{
for ( int j = 0 ; j <= 2 ; j++ )
{
a[i][j] = c ;
c++ ;
}
}
}
int* operator[] ( int i )
{
return a[i] ;
}
} ;
void main( )
{
emp e ;
cout << e[0][1] ;
}
```

The class emp has an overloaded operator [ ] function. It takes one argument an integer representing an array index and returns an int pointer. The statement cout << e[0][1] ; would get converted into a call to the overloaded [ ] function as e.operator[ ] (0). 0 would get collected in i. The function would return a[i] that represents the base address of the Zeroth row. Next the statement would get expanded as base address of Zeroth row[1] that can be further expanded as *(base address + 1). This gives us a value in zeroth row and first column.

Q42. What are formatting flags in ios class?

ANS:

The ios class contains formatting flags that help users to format the stream data. Formatting flags are a set of enum definitions. There are two types of formatting flags:

1. On/Off flags
2. Flags that work in-group

The On/Off flags are turned on using the setf( ) function and are turned off using the unsetf( ) function. To set the On/Off flags, the one argument setf( ) function is used. The flags working in groups are set through the two-argument setf( ) function. For example, to left justify a string we can set the flag as,

```
cout.setf ( ios::left ) ;
cout << KICIT Nagpur ;
```

To remove the left justification for subsequent output we can say,

```
cout.unsetf ( ios::left ) ;
```

The flags that can be set/unset include skipws, showbase, showpoint, uppercase, showpos, unitbuf and stdio. The flags that work in a group can have only one of these flags set at a time.

Q43. What is the purpose of ios::basefield in the following statement?
cout.setf (ios::hex, ios::basefield) ;

ANS:

This is an example of formatting flags that work in a group. There is a flag for each numbering system (base) like decimal, octal and hexadecimal. Collectively, these flags are referred to as basefield and are specified by ios::basefield flag. We can have only one of these flags on at a time. If we set the hex flag as setf (ios::hex) then we will set the hex bit but we wont clear the dec bit resulting in undefined behavior. The solution is to call setf( ) as setf (ios::hex, ios::basefield). This call first clears all the bits and then sets the hex bit.

Q44. Write a program that implements a date class containing day, month and year as data members. Implement assignment operator and copy constructor in this class.

ANS:

This is shown in following program:

```
#include
class date
{
private :
int day ;
int month ;
int year ;
public :
date ( int d = 0, int m = 0, int y = 0 )
{
day = d ;
month = m ;
year = y ;
}
// copy constructor
date ( date &d )
{
day = d.day ;
month = d.month ;
year = d.year ;
}
// an overloaded assignment operator
date operator = ( date d )
{
day = d.day ;
month = d.month ;
year = d.year ;
return d ;
}
void display( )
{
```

```
cout << day << / << month << / << year ;
}
} ;
void main( )
{
date d1 ( 25, 9, 1979 ) ;
date d2 = d1 ;
date d3 ;
d3 = d2 ;
d3.display( ) ;
}
```

Q45. What is the limitation of cin while taking input for character array?

ANS:

To understand this consider following statements,

```
char str[5] ;
cin >> str ;
```

While entering the value for str if we enter more than 5 characters then there is no provision in cin to check the array bounds. If the array overflows, it may be dangerous. This can be avoided by using get( ) function. For example, consider following statement,

```
cin.get ( str, 5 ) ;
```

On executing this statement if we enter more than 5 characters, then get( ) takes only first five characters and ignores rest of the characters. Some more variations of get( ) are available, such as shown below:
get (ch) ? Extracts one character only
get (str, n) ? Extracts up to n characters into str
get (str, DELIM) ? Extracts characters into array str until specified delimiter (such as). Leaves delimiting character in stream.
get (str, n, DELIM) ? Extracts characters into array str until n characters or DELIM character, leaving delimiting character in stream.

Q46. What is strstream?

ANS:

strstream is a type of input/output stream that works with the memory. It allows using section of the memory as a stream object. These streams provide the classes that can be used for storing the stream of bytes into memory. For example, we can store integers, floats and strings as a stream of bytes. There are several classes that implement this in-memory formatting.

The class ostrstream derived from ostream is used when output is to be sent to memory, the class istrstream derived from istream is used when input is taken from memory and strstream class derived from iostream is used for memory objects that do both input and output.

Q47. When the constructor of a base class calls a virtual function, why does not the override function of the derived class gets called?

ANS:

While building an object of a derived class first the constructor of the base class and then the constructor of the derived class gets called. The object is said an immature object at the stage when the constructor of base class is called. This object will be called a matured object after the execution of the constructor of the derived class. Thus, if we call a virtual function when an object is still immature, obviously, the virtual function of the base class would get called. This is illustrated in the following example.

```
#include
class base
{
protected :
int i ;
public :
base ( int ii = 0 )
{
i = ii ;
show( ) ;
}
virtual void show( )
```

```
{
cout << bases show( ) << endl ;
}
} ;
class derived : public base
{
private :
int j ;
public :
derived ( int ii, int jj = 0 ) : base ( ii )
{
j = jj ;
show( ) ;
}
void show( )
{
cout << deriveds show( ) << endl ;
}
} ;

void main( )
{
derived dobj ( 20, 5 ) ;
}
```

The output of this program would be:

```
bases show( )
deriveds show( )
```

Q48. Can I have a reference as a data member of a class? If yes, then how do I initialise it?

ANS:

Yes, we can have a reference as a data member of a class. A reference as a data member of a class is initialized in the initialization list of the constructor. This is shown in following program.

```
#include
```

```
class sample
{
private :
int& i ;
public :
sample ( int& ii ) : i ( ii )
{
}
void show( )
{
cout << i << endl ;
}
} ;
void main( )
{
int j = 10 ;
sample s ( j ) ;
s.show( ) ;
}
```

Here, i refers to a variable j allocated on the stack. A point to note here is that we cannot bind a reference to an object passed to the constructor as a value. If we do so, then the reference i would refer to the function parameter (i.e. parameter ii in the constructor), which would disappear as soon as the function returns, thereby creating a situation of dangling reference.

Q49. Why does the following code fail?

```
#include
class sample
{
private :
char *str ;
public :
sample ( char *s )
{
strcpy ( str, s ) ;
```

```
}
~sample( )
{
delete str ;
}
} ;
void main( )
{
sample s1 ( abc ) ;
}
```

ANS:

Here, through the destructor we are trying to deal locate memory, which has been allocated statically. To remove an exception, add following statement to the constructor.

```
sample ( char *s )
{
str = new char[strlen(s) + 1] ;
strcpy ( str, s ) ;
}
```

Here, first we have allocated memory of required size, which then would get deal located through the destructor.

Q50. Why it is unsafe to deal locate the memory using free( ) if it has been allocated using new?

ANS:

This can be explained with the following example:

```
#include
class sample
{
int *p ;
public :
sample( )
{
```

```
p = new int ;
}
~sample( )
{
delete p ;
}
} ;
void main( )
{
sample *s1 = new sample ;
free ( s1 ) ;
sample *s2 = ( sample * ) malloc ( sizeof ( sample ) ) ;
delete s2 ;
}
```

The new operator allocates memory and calls the constructor. In the constructor we have allocated memory on heap, which is pointed to by p. If we release the object using the free( ) function the object would die but the memory allocated in the constructor would leak. This is because free( ) being a C library function does not call the destructor where we have deal located the memory.

As against this, if we allocate memory by calling malloc( ) the constructor would not get called. Hence p holds a garbage address. Now if the memory is deal located using delete, the destructor would get called where we have tried to release the memory pointed to by p. Since p contains garbage this may result in a runtime error.