Texas Instruments
Date : 8/9/2005

1.

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.

2.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.

3.

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.

4.

The istream_withassign class has been derived from the istream class and overloaded assignment operator has been added to it. The _withassign classes are much like their base classes except that they include overloaded assignment operators. Using these operators the objects of the _withassign classes can be copied. The istream, ostream, and iostream classes are made uncopyable by making their overloaded copy constructor and assignment operators private.

5.

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 ;
}
```

6.

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

7.

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.

8.

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
}
```

9.

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. We have provided such a conversion function in the rectangle class. This conversion operator function returns a circle object. By default conversion operators have the name and return type same as the object type to which it converts to. Here the type of the object is circle and hence the name of the operator function as well as the return type is circle.

10.

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.

11.

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.

12.

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 zeroeth row. Next the statement would get expanded as base address of zeroeth row[1] that can be further expanded as *(base address + 1). This gives us a value in zeroth row and first column.

13.

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:

On/Off flags

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.

14.

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 won't 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.

15.

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.

16.

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 ?\n? (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.

17.

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( ) ;
}
```

18.

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.

19.

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. Manipulators are of two types, those that take an argument and those that don?t.

20.

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.

21.

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.

22.

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.

23.

What do the nocreate and noreplace flag ensure when they are used for opening a file?

Ans: nocreate and noreplace are file-opening modes. A bit in the ios class defines these modes. The flag nocreate ensures that the file must exist before opening it. On the other hand the flag noreplace 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.

24.

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 '\n'). 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.

25.

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.

26.

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.

27.

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.

28. 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. Ans: When we want to retrieve the streams of bytes from memory we can use istrestream. The following example shows the use of istrstream class.

```
#include
void main( )
{
int age ;
float salary ;
char name[50] ;
char str[] = "22 12004.50 K. Vishwanatth" ;
istrstream s ( str ) ;
s >> age >> salary >> name ;
cout << age << endl << salary << endl << name ;
cout << endl << s.rdbuf( ) ;
}
```

Here, s is the object of the class istrstream. When we are creating the object s, the constructor of istrstream gets called that receives a pointer to the zero terminated character array str. The statement s >> age >> salary >> name ; extracts the age, salary and the name from the istrstream object s. However, while extracting the name, only the first word of name gets extracted. The balance is extracted using rdbuf( ).

29.

When the constructor of a base class calls a virtual function, why doesn't 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 << "base's show( )" << endl ;
}
} ;
class derived : public base
{
private :
int j ;
public :
derived ( int ii, int jj = 0 ) : base ( ii )
{
j = jj ;
show( ) ;
}
void show( )
{
cout << "derived's show( )" << endl ;
}
} ;

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

The output of this program would be:

```
base's show( )
derived's show( )
```

30.

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 initialised in the initialisation 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.

31.

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.

32.

assert( ) macro...

We can use a macro called assert( ) to test for conditions that should not occur in a code. This macro expands to an if statement. If test evaluates to 0, assert prints an error message and calls abort to abort the program.

```
#include
#include
void main( )
{
int i ;
cout << "\nEnter an integer: " ;
cin >> i ;
assert ( i >= 0 ) ;
cout << i << endl ;
}
```

33.

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.

34.

Can we distribute function templates and class templates in object libraries?

Ans: No! We can compile a function template or a class template into object code (.obj file). The code that contains a call to the function template or the code that creates an object from a class template can get compiled. This is because the compiler merely checks whether the call matches the declaration (in case of function template) and whether the object definition matches class declaration (in case of class template). Since the function template and the class template definitions are not found, the compiler leaves it to the linker to restore this. However, during linking, linker doesn't

find the matching definitions for the function call or a matching definition for object creation. In short the expanded versions of templates are not found in
the object library. Hence the linker reports error.

35.

What is the difference between an inspector and a mutator ?

Ans: An inspector is a member function that returns information about an object's state (information stored in object's data members) without changing the object's state. A mutator is a member function that changes the state of an object. In the class Stack given below we have defined a mutator and an inspector.

```
class Stack
{
public :
int pop( ) ;
int getcount( ) ;
}
```

In the above example, the function pop( ) removes top element of stack thereby changing the state of an object. So, the function pop( ) is a mutator. The function getcount( ) is an inspector because it simply counts the number of elements in the stack without changing the stack.

36.

Namespaces:

The C++ language provides a single global namespace. This can cause problems with global name clashes. For instance, consider these two C++ header files:

```
// file1.h
float f ( float, int ) ;
class sample { ... } ;
// file2.h
class sample { ... } ;
```

With these definitions, it is impossible to use both header files in a single program; the sample classes will clash.A namespace is a declarative region that attaches an additional identifier to any names declared inside it. The additional identifier thus avoids the possibility that a name will conflict with names declared elsewhere in the program. It is possible to use the same name in separate namespaces without conflict even if the names appear in the same translation unit. As long as they appear in separate namespaces, each name will be unique because of the addition of the namespace identifier. For example:

```
// file1.h
namespace file1
{
float f ( float, int ) ;
class sample { ... } ;
}
// file2.h
namespace file2
{
class sample { ... } ;
}
```

Now the class names will not clash because they become file1::sample and file2::sample, respectively.

37.

What would be the output of the following program?

```
#include
class user
{
int i ;
float f ;
char c ;
public :
void displaydata( )
{
cout << endl << i << endl << f << endl << c ;
}
} ;
void main( )
{
cout << sizeof ( user ) ;
user u1 ;
cout << endl << sizeof ( u1 ) ;
u1.displaydata( ) ;
}
```

Ans: The output of this program would be,

```
9 or 7
```

9 or 7
Garbage
Garbage
Garbage

Since the user class contains three elements, int, float and char its size would be 9 bytes (int-4, float-4, char-1) under Windows and 7 bytes (int-2, float-4, char-1) under DOS. Second output is again the same because u1 is an object of the class user. Finally three garbage values are printed out because i, f and c are not initialized anywhere in the program.

Note that if you run this program you may not get the answer shown here. This is because packing is done for an object in memory to increase the access efficiency. For example, under DOS, the object would be aligned on a 2-byte boundary. As a result, the size of the object would be reported as 6 bytes. Unlike this, Windows being a 32-bit OS the object would be aligned on a 4-byte boundary. Hence the size of the object would be reported as 12 bytes. To force the alignment on a 1-byte boundary, write the following statement before the class declaration.

```
#pragma pack ( 1 )
```

38.

Write a program that will convert an integer pointer to an integer and vice-versa.

Ans: The following program demonstrates this.

```
#include
void main( )
{
int i = 65000 ;
int *iptr = reinterpret_cast ( i ) ;
cout << endl << iptr ;
iptr++ ;
cout << endl << iptr ;
i = reinterpret_cast ( iptr ) ;
cout << endl << i ;
i++ ;
cout << endl << i ;
}
```

39.

What is a const_cast?

Ans. The const_cast is used to convert a const to a non-const. This is shown in the following

program:

```
#include
void main( )
{
const int a = 0 ;
int *ptr = ( int * ) &a ; //one way
ptr = const_cast_ ( &a ) ; //better way
}
```

Here, the address of the const variable a is assigned to the pointer to a non-const variable. The const_cast is also used when we want to change the data members of a class inside the const member functions. The following code snippet shows this:

```
class sample
{
private:
int data;
public:
void func( ) const
{
(const_cast (this))->data = 70 ;
}
} ;
```

40.

What is forward referencing and when should it be used?

Ans: Forward referencing is generally required when we make a class or a function as a friend.

Consider 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 ) ;
}
```

On compiling this program it gives error on the following statement of test class. It gives an error that sample is undeclared identifier. friend void fun (sample, 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.

```
class sample ;
```

41.

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 ;
```

42.

Using a smart pointer can we iterate through a container?

Ans: Yes. A container is a collection of elements or objects. It helps to properly organize and store the data. Stacks, linked lists, arrays are examples of containers. Following program shows how to iterate through a container using a smart pointer.

```
#include
class smartpointer
{
private :
int *p ; // ordinary pointer
public :
smartpointer ( int n )
{
p = new int [ n ] ;
int *t = p ;
for ( int i = 0 ; i <= 9 ; i++ )
*t++ = i * i ;
}
int* operator ++ ( int )
{
return p++ ;
}
int operator * ( )
{
return *p ;
}
} ;
void main( )
{
smartpointer sp ( 10 ) ;
for ( int i = 0 ; i <= 9 ; i++ )
cout << *sp++ << endl ;
}
```

Here, sp is a smart pointer. When we say *sp, the operator * ( ) function gets called. It returns the integer being pointed to by p. When we say sp++ the operator ++ ( ) function gets called. It increments p to point to

the next element in the array and then returns the address of this new location.

43.

Can objects read and write themselves?

Ans: Yes! This can be explained with the help of following example:

```
#include
#include
class employee
{
private :
char name [ 20 ] ;
int age ;
float salary ;
public :
void getdata( )
{
cout << "Enter name, age and salary of employee : " ;
cin >> name >> age >> salary ;
}
void store( )
{
ofstream file ;
file.open ( "EMPLOYEE.DAT", ios::app | ios::binary ) ;
file.write ( ( char * ) this, sizeof ( *this ) ) ;
file.close( ) ;
}
void retrieve ( int n )
{
ifstream file ;
file.open ( "EMPLOYEE.DAT", ios::binary ) ;
file.seekg ( n * sizeof ( employee ) ) ;
file.read ( ( char * ) this, sizeof ( *this ) ) ;
file.close( ) ;
}
void show( )
{
cout << "Name : " << name
<< endl << "Age : " << age
<< endl << "Salary :" << salary << endl ;
}
} ;
void main( )
{
employee e [ 5 ] ;
for ( int i = 0 ; i <= 4 ; i++ )
{
e [ i ].getdata( ) ;
e [ i ].store( ) ;
}
for ( i = 0 ; i <= 4 ; i++ )
{
e [ i ].retrieve ( i ) ;
e [ i ].show( ) ;
}
}
```

Here, employee is the class whose objects can write and read themselves. The getdata( ) function has been used to get the data of employee and store it in the data members name, age and salary. The store( ) function is used to write an object to the file. In this function a file has been opened in append mode and each time data of current object has been stored after the last record (if any) in the file.Function retrieve( ) is used to get the data of a particular employee from the file. This retrieved data has been stored in the data members name, age and salary. Here this has been used to store data since it contains the address of the current object. The function show( ) has been used to display the data of employee.

44.

Why is it necessary to use a reference in the argument to the copy constructor?

Ans : If we pass the copy constructor the argument by value, its copy would get constructed using the copy constructor. This means the copy constructor would call itself to make this copy. This process would go on and on until the compiler runs out of memory. This can be explained with the help of following example:

```
class sample
{
int i ;
public :
sample ( sample p )
{
i = p.i ;
}
} ;

void main( )
{
sample s ;
sample s1 ( s ) ;
}
```

While executing the statement sample s1 (s), the copy constructor would get called. As the copy construct here accepts a value, the value of s would be passed which would get collected in p. We can think of this statement as sample p = s. Here p is getting created and initialized. Means again the copy constructor would get called. This would result into recursive calls. Hence we must use a reference as an argument in a copy constructor.

45.

46. Virtual Multiple Inheritance:

A class b is defined having member variable i. Suppose two classes d1 and d2 are derived from class b and a class multiple is derived from both d1 and d2. If variable i is accessed from a member function of multiple then it gives error as 'member is ambiguous'. To avoid this error derive classes d1 and d2 with modifier virtual as shown in the following program.

```
#include
class b
{
public :
int i ;
public :
fun( )
{
i = 0 ;
}
} ;
class d1 : virtual public b
{
public :
fun( )
{
i = 1 ;
}
} ;
class d2 : virtual public b
{
public :
fun( )
{
i = 2 ;
}
} ;
class multiple : public d1, public d2
{
public :
fun( )
{
```

```
i = 10 ;
}
} ;
void main( )
{
multiple d ;
d.fun( ) ;
cout << d.i ;
}
```

46.

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( ).

47.

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.

48.

When should I overload new operator on a global basis or a class basis?

Ans: We overload operator new in our program, when we want to initialize a data item or a class object at the same place where it has been allocated memory. The following example shows how to overload new operator on global basis.

```
#include
#include
void * operator new ( size_t s )
{
void *q = malloc ( s ) ;
return q ;
}
void main( )
```

```
{
int *p = new int ;
*p = 25 ;
cout << *p ;
}
```

When the operator new is overloaded on global basis it becomes impossible to initialize the data members of a class as different classes may have different types of data members. The following example shows how to overload new operator on class-by-class basis.

```
#include
#include
class sample
{
int i ;
public :
void* operator new ( size_t s, int ii )
{
sample *q = ( sample * ) malloc ( s ) ;
q -> i = ii ;
return q ;
}
} ;
class sample1
{
float f ;
public :
void* operator new ( size_t s, float ff )
{
sample1 *q = ( sample1 * ) malloc ( s ) ;
q -> f = ff ;
return q ;
}
} ;
void main( )
{
sample *s = new ( 7 ) sample ;
sample1 *s1 = new ( 5.6f ) sample1 ;
}
```

Overloading the operator new on class-by-class basis makes it possible to allocate memory for an object and initialize its data members at the same place.

49.

How would you define a pointer to a data member of the type pointer to pointer?

Ans: The following program demonstrates this...

```
#include
class sample
{
public :
sample ( int **pp )
{
p = pp ;
}
int **p ;
} ;
int **sample::*ptr = &sample::p ;
void main( )
{
int i = 9 ;
int *pi = &i ;
sample s ( &pi ) ;
cout << ** ( s.*ptr ) ;
}
```

Here, ptr is the pointer to data member p of class sample, which in turn is a pointer pointing to an int.

50.

How do I write a code to catch multiple types of exceptions in one single catch block?

Ans: The following program demonstrates the use of a single catch block to catch multiple exceptions.

```
#include
class test
{
} ;
class sample
{
```

```
public :
void fun1( )
{
throw 99 ;
}
void fun2( )
{
throw 3.14f ;
}
void fun3( )
{
throw "error" ;
}
void fun4( )
{
throw test( ) ;
}
} ;
void main( )
{
try
{
sample s ;
s.fun4( ) ;
s.fun1( ) ;
s.fun2( ) ;
s.fun3( ) ;
}
catch ( ... )
{
cout << "strange" ;
}
}
```

Here, different types of exceptions are thrown by the member functions of the class sample. While catching the exception instead of four different catch blocks we can as well define one single catch block. Note the syntax for defining the catch block, where we have used three dots (?) in the formal parameter list. This indicates that any thrown exception should get caught in the same catch block. When the exception is thrown from the fun4( ) control reaches the catch block, ignoring the rest of the calls.

51.

Can we return an error value from the constructor of a class?

Ans: No. We cannot return any error value from the constructor, as the constructor doesn't have any return type. However, by throwing an exception we can pass value to catch block. This is shown in the following example:

```
#include
class sample
{
public :
sample ( int i )
{
if ( i == 0 )
throw "error" ;
}
} ;
void main( )
{
try
{
sample s ( 0 ) ;
}
catch ( char * str )
{
cout << str ;
}
}
```

In this program, the statement throw "error" ; would throw an exception when an object s of the class sample would get created. The catch block would collect the string error.

52.

How do I define the member function of a template class, which has to be defined outside the template class. The function receives an object of its own class as a parameter and returns the value of the same type.

Ans: The following example shows how we can define such a function.

```
sample sample::fun ( sample s )
{
// code
}
```

Here, the first sample indicates the return type of the function and the next sample is used for the scope of function.

53.

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( ) ;
}
```

54.

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.

55.

What is RTTI?

Ans: RTTI stands for 'Run Time Type Information'. We use virtual function mechanism where we can call derived class's member functions using base class's 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 object's class at run time, they are typeid( ) operator and dynamic_cast operator.

56.

What is Data Conversion?

Ans: Assignments between types whether they are basic or user-defined, are handled by the compiler. If the variables are of different basic types compiler calls a special routine to convert the value. But if we want to convert between user-defined data type and basic types we have to write conversion routine ourselves. A conversion routine to convert user-defined data type string to integer is shown below:

```
class string
{
private :
char str[20] ;
public :
string( )
{
}
string ( char *s )
{
strcpy ( str, s ) ;
}
operator int( )
{
return 123 ; // Write logic to convert string to integer
}
} ;
main( )
{
string s2 = "123" ;
int i1 = int ( s2 ) ;
cout << endl << i1 ;
}
```

57.

How to obtain type information using typeid( ) operator?

Ans: typeid( ) operator takes an object, a reference or a pointer and returns its type. Following program shows how to use the typeid( ) operator.

```
#include
#include
class Base
{
public :
virtual void show( )
{
}
};

class Der1 : public Base
{
} ;
void main( )
{
Base *b1 ;
cout << endl << typeid ( b1 ).name( ) ;

Der1 d1 ;
b1 = &d1 ;
cout << endl << typeid ( *b1 ).name( ) ;
cout << endl << typeid ( 12 ).name( ) << endl << typeid ( 12.5 ).name( )
;
}
```

The output of this program will be

```
Base*
Der1
int
double
```

RTTI operators must be used for polymorphic class (class having virtual function) only. For non-polymorphic class static type information is returned.

58.

How to use RTTI with class templates?

Ans: Templates can generate different classes. We may wish to get the type of class, which we are working in. The following program shows how to use RTTI operator typeid( ) with class template.

```
#include
#include
template
class base
{
public :
base( )
{
```

```
cout << typeid ( *this ).name( ) << "Constructor" << endl ;
}
T add ( T a, T b )
{
return a + b ;
}
~base( )
{
cout << typeid ( *this ).name( ) << "Destructor" << endl ;
}
} ;
void main( )
{
base b1 ;
cout << b1.add ( 10, 20 ) << endl ;
base b2 ;
cout << b2.add ( 5.5, 10.5 ) << endl ;
}
```

59.

We can use following C++ operators for typecasting.static_cast is used for castless conversions, narrowing conversions, conversion from void* and implicit type conversions. const_cast is used to convert a const to a non-const. reinterpret_cast is used to assign one kind of pointer to another.

60.

What will be the output of the following program?

```
#include
class A
{
public :
A( )
{
cout << "Reached in Constructor\n" ;
}
} ;
void main( )
{
A a( ) ;
A b ;
}
Output : Reached in Constructor
```

Constructor gets called only once when the object b is created. When the statement A a( ) ; gets executed constructor does not get called. This is because compiler takes this statement as a prototype declaration of function a( ) that returns an object of class A. However, if we pass arguments like

```
A a ( 10 ) ;
```

Compiler would search for one argument constructor and if not found would flash an error.

61.

What is a container?

Ans: A container is an object that holds other objects. Various collection classes like List, Hash Table, AbstractArray, 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.

62.

Function template overloading

One can declare several function templates with the same name and even declare a combination of function templates and ordinary functions with the same name. When an overloaded function is called, overload resolution is necessary to find the right function or template function to invoke.

For example:

```
template < class T > T sqrt ( T ) ;
template < class T > complex < T > sqrt ( complex < T > ) ;double sqrt (
double ) ;
void f ( complex < double > z )
{
sqrt ( 2 ) ; // sqrt < int > ( int )
sqrt ( 2.0 ) ; // sqrt ( double )
sqrt ( z ) ; // sqrt < complex < double > ( complex < double > )
}
```

In the same way that a template function is a generalization of the notion of a function, the rules for resolution in the presence of function templates are generalizations of the function overload resolution rules. Basically, for each template we find the specialization that is best for the set of function arguments. Then we apply the usual function overload resolution rules to these specializations and all ordinary functions.

63.

Exception Handling in C++

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
}
}
```

64.

Consider the following code:

```
#include
class base
{
public :
int data ;
} ;
class d1 : public base
{
} ;
class d2 : public base
{
} ;
class der : public d1, public d2
{
public :
void showdata( )
{
cout << data ;
}
} ;
void main( )
{
der d ;
d.showdata( ) ;
}
```

If you run this program it is bound to give you errors. This is because of the rules of inheritance:

1. Each base class not specified virtual will have its own sub-object representing it. In the above program, if we create object of d1 it will have a sub-object of class base containing a data member data. If we create an object of class der it will have sub-objects of classes d1 and d2 and both the sub-objects will refer to a separate copy of data. Hence, to access data from class der we will have to mention the class name. For example, d1::data or d2::data.

2. If we want that only one sub-object should exist we must use the concept of virtual base class. The single object of this will represent every base class of given name that is specified to be virtual

class. After making d1 and d2 as virtual base class if we create an object of der only one sub-object would exist and so accessing data would no longer give us errors.

65.

How to declare a pointer to a member function?

Ans: Suppose, I wish to declare a pointer to a member function that receives an int and returns an int. I will have to declare it as int (A::*) (int). Following is an example.

```
#include
class A
{
public :
int fun ( int f )
{
cout << "in fun\n" ;

return f * f ;
}
} ;
typedef int ( A:: *pfun ) ( int ) ;

void main( )
{
pfun p = A::fun ;
A a ;
int s = ( a.*p ) ( 6 ) ;
cout << s ;
}
```

66.

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.

67.

How to declare a pointer to the data members of a class?

Ans: Following program shows how to declare a pointer to non-function members of a class.

```
#include
class A
{
public :

int a ;
void print( )
{
cout << a ;
}
} ;
void main( )
{
int A::*pa = &A::a ;

A obj ;
obj.*pa = 20 ;
obj.print( ) ;
}
```

Here, we have initialised the data member a using the pointer pa.

68.

How to allocate memory for a multidimensional array dynamically?

Ans: Many times we need to allocate memory for a multidimensional array dynamically. Because of complexity of pointers many find this difficult. Following program allocates memory for a 3 x 3 array dynamically, copies contents of a 3 x 3 array in it and prints the contents using the pointer.

```
#include
#include
int a[ ][3] = {
1, 2, 3,
4, 5, 6,
7, 8, 9
} ;
void main( )
{
int **p ;
p = new int *[3] ;
for ( int i = 0 ; i < 3 ; i++ )
p[i] = new int[3] ;
for ( i = 0 ; i < 3 ; i++ )
for ( int j = 0 ; j < 3 ; j++ )
p[i][j] = a[i][j] ;
for ( i = 0 ; i < 3 ; i++ )
{
for ( j = 0 ; j < 3 ; j++ )
cout << p[i][j] ;
cout << "\n" ;
}
}
```
69.

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.

70.

How do I use operators .* and ->* in a program?

Ans: The following code snippet demonstrates the use of .* and ->* operators.

```
#include
class sample
{
public :
int i ;
void fun( )
{
cout << "fun" << endl ;
}
} ;
void ( sample::*pf )( ) = &sample::fun ;
```

```
int sample::*pdm = &sample::i ;
void main( )
{
sample s ;
sample *p = new sample ;
( s .* pf )( ) ;
( p ->* pf )( ) ;
s .* pdm = 1 ;
p ->* pdm = 2 ;
cout << s .* pdm << endl ;
cout << p ->* pdm << endl ;
}
```

In the above program pf is a pointer to a function fun( ) of class sample, and pdm is a pointer to a data member i of the same class sample. The object s of the class sample is created statically. Next, p is a pointer to an object created dynamically. The using the operator .* and ->* the member functions are called and also the public data member is accessed.

71.

What happens when we add an int value to a user defined type of object?

Ans: Whenever an int value is added to an object of user defined type, the object would search for an overloaded operator int( ). This operator must be defined in such a way that it always returns an int value. However, we need not specify the return type as on doing so the compiler flashes an error.

```
#include
class sample
{
int i ;

public :
sample ( )
{
i = 10 ;
}
operator int( )
{
return this -> i ;
}
} ;
void main( )
```

```
{
sample s ;
int i ;
i = s + 10 ;
cout << i ;
}
```

In the above program on adding 10 to an object s, the value of i would become 20.

72.

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 ;
```

73.

When friend function becomes indispensable...

Ans: Consider the following program.

```
#include
class distance
{
private :
int feet ;
public :
distance( )
{
feet = 0 ;
}
distance ( int f )
{
feet = f ;
}
distance operator + ( distance x )
{
```

```
int f = feet + x.feet ;
return distance ( f ) ;
}
} ;
void main( )
{
distance d1 ( 20 ), d2, d3 ;
d2 = d1 + 10 ;
d3 = 10 + d2 ;
}
```

If you run this program it is bound to give errors. The error lies in the statement d3 = 10 + d2 ; We may think that since we have overloaded + operator this statement would add 10 to d2. But this does not happen. This is because the specified statement will get converted as d3 = 10.operator+ (d2) ; This means that this statement should call the operator+( ) function that takes an object of distance class as parameter written in

the float class, which is not possible. The solution is to write operator+( ) as a 'friend' function. Declare operator+ function in distance class as given below:

```
friend distance operator + ( distance x1, distance x2 ) ;
```

and define it outside the class as shown below:

```
distance operator + ( distance x1, distance x2 )
{
int f = x1.feet + x2.feet ;
return distance ( f ) ;
}
```

When compiler would see that the 'friend' operator+( ) function is available it would convert the statement d3 = 10 + d2 as operator+ (10, d2). Now since 10 is passed as a parameter not as a calling object there would be no error. Thus in such cases 'friend' function becomes indispensable.

74.

How to use a memory as a stream?

Ans: Suppose, details of an employee such as name, designation, age, etc. are stored in different types of variables. Now, if we wish to concatenate these details in a character array we will have to use various string manipulation functions like strcpy( ) and strcat( ). Instead of using these functions we can use more easy and clean way to gather the details in the char array in the form of streams. We can declare the memory allocated for the array as stream and use the << operator to store variables having different types in this memory. Following program shows how to achieve this.

```
#include
void main( )
{
char buff [50] ;
char str[ ] = "Sanjay" ;
char desig[ ] = "Manager" ;
char jd[ ] = "27/12/1995" ;
int age = 35 ;
ostrstream o ( buff, sizeof ( buff ) ) ;
o << str << endl << desig << endl << jd << endl << age << ends ;
cout << buff ;
}
```

As shown in the program we can also use the manipulators and formatting flags. The output of this program will be:

```
Sanjay
Manager
27/12/1995
35
```

75.

How would you declare and initialize reference to a data member?

Ans: Sometimes we may need to declare a data member, which is a reference to another data member of the class as shown below:

```
class A
{
public :
char *p ;
char *&rp ;
} ;
```

We can't initialize a reference to a data member at the time of declaration. It should be initialized using 'member wise initialization as shown below.

```
#include
class A
{
public :
char *p ;
char *&rp ;
A( ) : rp ( p )
{
p = "" ;
}
A ( char *s ) : rp ( p )
{
p = s ;
}
} ;
void main( )
{
A a ( "abcd" ) ;
cout << a.rp ;
}
```

76.

iostream library has made it easy to read data from various input devices and write data to the output devices. The following program shows how to print a disk file 'data.dat' on the printer using stream classes. Every hardware device has a familiar name given by the operating system. The printer is generally connected to the first parallel port. So, the file name for the printer should be PRN or lpt1.

```
#include
void main( )
{
ifstream i ( "data.dat" ) ;
ofstream o ;
o.open ( "PRN" ) ;
char ch ;
while ( 1 )
{
i.get ( ch ) ;
if ( i.eof( ) )
break ;
o.put ( ch ) ;
}
o.put ( '\x0C' ) ;
}
```

77.

We know that a destructor is automatically called when an object of a class goes out of scope. There is another case where destructor is called automatically. If an object is created in a try block and an exception is thrown after the object is created, then the destructor is called automatically.

78.

Can a function call be at the left hand side of the assignment operator?
Ans: Yes. Following program shows how it is possible.

```
#include
class ref
{
private :
struct data
{
int a ; char *p ;
} d1, d2 ;
public :
data &set ( )
{
return d1 ;
}
data &get ( )
{
cin >> d2.a >> d2.p ;
return d2 ;
}
} ;
void main( )
{
ref r ;
r.set( ) = r.get( ) ;
r.print( ) ;
}
```

In the above program the functions get( ) and set( ) both return a reference to the object of the structure data. We have assigned the reference returned by get( ) to the reference returned by set( ) function. That is, we are assigning d2 to d1. So, the values of d2 would get assigned to d1. You can check this out by printing the values of d1.

79.

If a class contains a virtual function a pointer called VPTR is created. This VPTR becomes a part of every object of that class. The first two bytes (in DOS) are occupied by VPTR. We can prove this by displaying the first two bytes of memory allocated for the objects. Following program shows how this can be achieved.

```
#include
class vir
{
public :
virtual void f( )
{
}
} ;
void main( )
{
vir v, v1 ;
int *p1 = ( int* ) &v ;
int *p2 = ( int* ) &v1 ;
cout << endl << *p1 << " " << *p2 ;
}
```

80.

Exception Handling in C++

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
}
}
```

81.

Accessing a private data member from a different Object...Different objects of the same class can access each other's members, even if these members are private. For example:

```
#include < iostream.h >
class sample
{
float f ;
public :
sample ( float ff )
{
f = ff ;
}
void fun ( sample* objptr )
{
objptr -> n = 0 ;
cout << "Value of this objects f is : " << f << endl ;
cout << "Value of other objects f" << objptr -> n << endl ;
} // another object's private member!
} ;
```

```
void main( )
{
sample s1 ( 6.5f ) , s2 ( 2.5f ) ;
s1.f ( &s2 ) ; // s1 changes s2's n
}
```

Typically, this coding style should be avoided. However, you should be aware that private members of an object can be changed by another object of the same type. Therefore, in certain special conditions, this coding style may be useful.

82.

Can you access private data members of a class from out side the class?
Ans: Yes. This program shows how.

```
#include
class emp
private :
int i ;
public :
emp( )
{
i = 10 ;
}
} ;
void main( )
emp *p = new emp ;
int *pi = (int*) p ;
cout << *pi ;
*pi = 20 ;
cout << *pi ;
}
```

The pointer to the class is typecasted in an integer pointer. With the help of this pointer private data member 'i' is accessed in main( ).

83.

Why creating array of references is not possible?
Ans: The array name always refers or points to the zeroeth element. If array is of references then the array name would point to the zeroeth element

which happens to be a reference. Creating pointer to a reference is not valid. So, creating array of references too is not possible.

84.

How do I call a virtual function of a class using a pointer to a function ?

Ans :

```
#include
class Cvirtual
{
public :
virtual float vfun( )
{
cout << "from vfun" << endl ;
return 7.03f ;
}
} ;
void main( )
{
Cvirtual obj ;
int **p = ( int ** ) &obj ;
float ( *pf1 ) ( ) ;
pf1 = ( float ( * ) ( ) ) **p ;
float f = ( *pf1 ) ( ) ;
cout << "return val = " << f << endl ;
}
```

In the above program class Cvirtual consists of a virtual function vfun(). In variable p we have stored the address of an object of class Cvirtual. While doing so, we have type casted the address of obj to int **, because obj holds a hidden data member called vptr, which in turn holds the address of virtual function vfun( ). In pf1, a pointer to a function, we are collecting the address of the virtual function vfun( ). Thus the value returned by vfun( ) would then get collected in f.

85.

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 doesn't 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.

86.

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.

87.

When we are required to find offset of an element within a structure? or, how do we call the function of an outer class from a function in the inner class? (The inner class is nested in the outer class)

Ans:

```
#include
class outer
{
int i ;
float f ;
public :
class inner
{
public :
infunc( )
{
outer *pout ;
pout = (outer*) this - ( size_t ) &( ( ( outer* ) 0 ) -> in ) ;
pout -> outfunc( ) ;
}
} ;
inner in ;
outfunc( )
{
cout << "in outer class's function" ;
}
} ;
void main( )
```

```
{
outer out ;
out.in.infunc( )
}
```

In the above example we are calling outer::outfunc( ) from inner::infunc(). To call outfunc( ) we need a pointer to the outer class. To get the pointer we have subtracted offset of the inner class's object (base address of outer class's object - address of inner class's object) from address of inner class's object.

88.

```
void f ( float n, int i = 10 ) ;
void f ( float a ) ;
void main( )
{
f ( 12.6 ) ;
}

void f ( float n, int i )
{

}

void f ( float n )
{

}
```

The above program results in an error (ambiguous call) since without the default argument the two functions have arguments that are matching in number, order and type.

89.

Some programs need to exercise precise control over the memory areas where data is placed. For example, suppose we wish to read the contents of the boot sector into a structure. For this the byte arrangement of the structure elements must match the arrangement of various fields in the boot sector of the disk.

The #pragma pack directives offer a way to fulfill this requirement. The #pragma pack directive specifies packing alignment for structure and union members. The #pragma takes effect at the first structure or union declaration after the #pragma is seen. Consider the following structure:

```
#pragma pack (1)
struct emp
{
int a ;
float s ;
char ch ;
} ;
#pragma pack( )
```

Here, #pragma pack (1) lets each structure element to begin on a 1-byte boundary. Hence the size of the structure will be 9. (int - 4, float - 4, char - 1). If we use #pragma pack (2) each structure element can begin on a 2-byte boundary. Hence the size of the structure will be 10. (int - 4, float - 4, char - 2).

90.

How to restrict a friend class's access to the private data members?

Ans: If we declare a class as a friend of our class the friend class can access the private data members of our class. However, if we want we can restrict this access to some selective functions of the class. Following program shows how to achieve this:

```
#include
class X
{
public :
void print ( class Z &z ) ;
} ;
class Z
{
private :
int i ;
public :
Z ( int ii )
{
i = ii ;
}
friend X::print ( class Z &z ) ;
} ;
void X::print ( Z &z1 )
```

```
{
cout << z1.i ;
}
main( )
{
Z z ( 10 ) ;
X x ;
x.print ( 10 ) ;
}
```

In the above program only the X::print( ) function can access the private data members of class Z.

91.

What is name mangling?

Ans: C++ enables you to assign the same function name to more than one functions but with different parameter types. This feature is called function overloading. But when we give several functions the same name, how does the compiler decide which particular function is to be called? C++ solves this problem by applying a process called name mangling. Name mangling applies a decorated name to the function. The mangled name includes tokens that identify the functions' return type and the types of its arguments.

```
class test
{
public :
void fun ( int a, char b ) ;
void fun ( char *c, float y ) ;
} ;
void main( )
{
test s1 ;
s1.fun ( 65, 'A' ) ;
s1.fun ( "Anil", 5.5f ) ;
}
```

At the time of resolving the calls to fun( ) function the linker would not be able to find the definition of the overloaded function fun( ) and it would report an error. If you look at these errors you will see the mangled names like, (?fun@test@@QAEXJJ@Z) and (?fun@test@@QAEXMM@Z). Note that different compilers may use different name mangling schemes.

92.

How would you call a C function from C++ code?
Ans: Using extern "C".
The function prototype must be preceded by extern "C". More than one C functions can be grouped inside braces as shown below:

```
extern "C"
{
void f( ) ;
void f1( ) ;
}
// In cfunc.c
#include
void f( )
{
printf ( "in f( )" ) ;
}
// In func.cpp
#include
extern "C" void f( ) ;
void main( )
{
f( ) ;
}
```

Ensure that both .c and .cpp files are in the same project.

93.

How to restrict the number of floating-point digits displayed ?
Ans: When we display floating-point values, we can use the setprecision 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.

94.

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.

95.

How friend function helps to increase the versatility of overloaded operators?

Ans: Consider the following statement,

s2 = s1 * 2 ;

where, s1 and s2 are objects of sample class. This statement would work if the overloaded operator * (sample s) or conversion function is provided in the class. Internally this statement would get converted to,

s2 = s1.operator * (2) ;

The function materializes because it is called with an object s1. The this pointer of s1 would get passed implicitly. To collect 2 in s, first the compiler would call the one-argument constructor, then it would build a

nameless object, which then would get collected in s. However, if we write the above statement as,

s2 = 2 * s1 ;

then it won't compile. This is because the call now would get treated as,

s2 = 2.operator * (s1) ;

and 2 is not an object. The friend function helps to get rid of such a situation. This is shown in the following program.

```
#include
class sample
{
private :
int i ;
public :
sample ( int ii = 0 )
{
i = ii ;
}
void showdata( )
{
cout << i << endl ;
}
friend sample operator * ( sample, sample ) ;
} ;
sample operator * ( sample s1, sample s2 )
{
sample temp ;
temp.i = s1.i * s2.i ;
return ( temp ) ;
}
void main( )
{
sample s1 ( 10 ), s2 ;

s2 = s1 * 2 ;
s2.showdata( ) ;

s1 = 2 * s2 ;
s1.showdata( ) ;
}
```

Here the operator *( ) function takes two parameters. This is because the operator function is no longer a member function of the class. It is a friend of the class sample. Thus the statement s2 = s1 * 2 ; would not take the form s2.operator * (2). This example shows that using friend permits the overloaded operators to be more versatile.

96.

What is a const_cast?

Ans: The const_cast is used to convert a const to a non-const. This is shown in the following program.

```
#include
void main( )
{
const int a = 0 ;
int *ptr = ( int * ) &a ; // one way
ptr = const_cast ( &a ) ; // better way
}
```

Here, the address of the const variable a is assigned to the pointer to a non-const variable. The const_cast is also used when we want to change the data members of a class inside the const member functions. The following code snippet shows how to do this.

```
class sample
{
private :
int data ;
public :
void fun( ) const
{
```

```
( const_cast ( this ) ) -> data = 70 ;
}
} ;
```
97.

Using a smart pointer we can make an object appear like a pointer.

If a class overloads the operator -> then any object of that class can appear like a pointer when the operator -> ( ) is called. The following program illustrates this.

```
#include
class test
{
public :
void fun( )
{
cout << "fun of smart pointer" ;
}
} ;
class smartpointer
{
test t ;
public :
test* operator ->( )
{
return &t ;
}
} ;
void main( )
{
smartpointer sp ;
sp -> fun( ) ;
}
```

The beauty of overloading operator -> is that even though sp is an object we can make it work like a pointer. The operator -> ( ) returns the address of the object of the type test. Using this address of the test object the function fun( ) of the class test gets called. Thus even though fun( ) is not a member of smartpointer class we can still call it using sp.

98.

Can we apply delete on this pointer inside a member function?

Ans : Yes! If the member function of a class is called using a pointer to an object, which is allocated dynamically, the object would get deleted. But if the member function is called using the object, which is allocated statically, then a runtime error would occur. This is because we cannot call delete on statically allocated objects. This is illustrated in the following example.

```
class sample
{
private :
int i ;
public :
void fun( )
{
delete this ;
}
} ;

void main( )
{
sample *s = new sample ;
s -> fun( ) ; // no error

sample s1 ;
s1.fun( ) ; // would throw a runtime error
}
```

99.

Why can't data members of a class be initialized at the time of declaration as given in the following code?

```
class emp
{
private :
int j = 10 ;
} ;
```

Ans: Memory for data members of a class is allocated only when object of that class is created. One cannot store data in a memory location, which does not exist at all. Therefore initialization at the time of declaration is not possible.

100.

Why in a copy constructor an object is collected in a reference to object as shown below?

```
#include
class emp
{
public :
emp( )
{
}
emp ( emp& )
{
cout << "copy" ;
}
} ;
void main( )
{
emp e ;
emp e1 = e ;
}
```

Ans: A copy constructor is called when an object is created and initialised at the same time. It is also called when object is passed to a function. So, If we pass the object to copy constructor copy constructor would get called recursively. Thus it will stuck up in an infinite loop.

101.

What is 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.

102. 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 ;

Now s will do its own memory management. We can stuff as many bytes into it as we want. If it falls short of memory, it will allocate more memory. If it cannot, it may even move the block of memory. When the object goes out of scope, the heap storage is automatically released. This is a more flexible approach if we do not know how much space we are going to need. If we want the physical address of the memory used by s we can obtain it by calling the str( ) member function:

char* p = s.str( ) ;

Once str( ) has been called then the block of memory allocated by ostrstream cannot be moved. This is logical. It can't 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 can't 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. You took over that responsibility when you asked for the char * with str( ). We can clean the storage in two ways: Using the delete operator as shown below:

ostrstream s ;

char *p ;

p = s.str( ) ;

delete p ;

By unfreezing the ostrstream. You do this by calling freeze( ), with an argument 1. During freezing it is called with the default argument of 0.