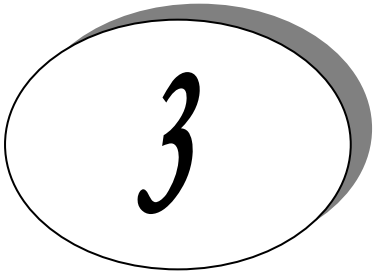
C++ Aptitude and OOPS



Note : All the programs are tested under Turbo C++ 3.0, 4.5 and Microsoft VC++ 6.0 compilers.

It is assumed that,

- Programs run under Windows environment,
- The underlying machine is an x86 based system,
- Program is compiled using Turbo C/C++ compiler.

The program output may depend on the information based on this assumptions (for example sizeof(int) == 2 may be assumed).

1)
```
class Sample
{
public:
    int *ptr;
    Sample(int i)
    {
    ptr = new int(i);
    }
    ~Sample()
    {
    delete ptr;
    }
    void PrintVal()
    {
    cout << "The value is " << *ptr;
    }
};
void SomeFunc(Sample x)
{
cout << "Say i am in someFunc " << endl;
}
int main()
{
Sample s1= 10;
SomeFunc(s1);
s1.PrintVal();
}
```

Answer:

Say i am in someFunc
Null pointer assignment(Run-time error)

Explanation:

As the object is passed by value to SomeFunc the destructor of the object is called when the control returns from the function. So when PrintVal is called it meets up with ptr that has been freed.The solution is to pass the Sample object by *reference* to SomeFunc:

```
void SomeFunc(Sample &x)
{
cout << "Say i am in someFunc " << endl;
}
```

because when we pass objects by refernece that object is not destroyed. while returning from the function.

2) Which is the parameter that is added to every non-static member function when it is called?

Answer:

'this' pointer

3)
```
class base
    {
    public:
    int bval;
    base(){ bval=0;}
    };

class deri:public base
    {
    public:
    int dval;
    deri(){ dval=1;}
    };
void SomeFunc(base *arr,int size)
{
for(int i=0; i<size; i++,arr++)
    cout<<arr->bval;
cout<<endl;
}

int main()
{
base BaseArr[5];
SomeFunc(BaseArr,5);
deri DeriArr[5];
SomeFunc(DeriArr,5);
}
```

Answer:

00000
01010

Explanation:

The function SomeFunc expects two arguments.The first one is a pointer to an array of base class objects and the second one is the sizeof the array.The first call of someFunc

calls it with an array of bae objects, so it works correctly and prints the bval of all the objects. When Somefunc is called the second time the argument passed is the pointeer to an array of derived class objects and not the array of base class objects. But that is what the function expects to be sent. So the derived class pointer is promoted to base class pointer and the address is sent to the function. SomeFunc() knows nothing about this and just treats the pointer as an array of base class objects. So when arr++ is met, the size of base class object is taken into consideration and is incremented by sizeof(int) bytes for bval (the deri class objects have bval and dval as members and so is of size >= sizeof(int)+sizeof(int)).

4) class base
```
    {
    public:
       void baseFun(){ cout<<"from base"<<endl;}
    };
 class deri:public base
    {
    public:
       void baseFun(){ cout<< "from derived"<<endl;}
    };
void SomeFunc(base *baseObj)
{
    baseObj->baseFun();
}
int main()
{
base baseObject;
SomeFunc(&baseObject);
deri deriObject;
SomeFunc(&deriObject);
}
```
Answer:

from base
from base

Explanation:

As we have seen in the previous case, SomeFunc expects a pointer to a base class. Since a pointer to a derived class object is passed, it treats the argument only as a base class pointer and the corresponding base function is called.

5) class base
```
    {
    public:
       virtual void baseFun(){ cout<<"from base"<<endl;}
    };
 class deri:public base
    {
    public:
       void baseFun(){ cout<< "from derived"<<endl;}
    };
void SomeFunc(base *baseObj)
{
```

```
        baseObj->baseFun();
}
int main()
{
base baseObject;
SomeFunc(&baseObject);
deri deriObject;
SomeFunc(&deriObject);
}
```

Answer:

from base
from derived

Explanation:

Remember that baseFunc is a virtual function. That means that it supports run-time polymorphism. So the function corresponding to the derived class object is called.

```
void main()
{
        int a, *pa, &ra;
        pa = &a;
        ra = a;
        cout <<"a="<<a <<"*pa="<<*pa <<"ra"<<ra ;
}
/*
Answer :
        Compiler Error: 'ra',reference must be initialized
Explanation :
        Pointers are different from references. One of the main
differences is that the pointers can be both initialized and assigned,
whereas references can only be initialized. So this code issues an error.
*/

const int size = 5;
void print(int *ptr)
{
        cout<<ptr[0];
}

void print(int ptr[size])
{
        cout<<ptr[0];
}

void main()
{
        int a[size] = {1,2,3,4,5};
        int *b = new int(size);
        print(a);
        print(b);
```

```
}
/*
Answer:
        Compiler Error : function 'void print(int *)' already has a body

Explanation:
        Arrays cannot be passed to functions, only pointers (for arrays, base addresses)
can be passed. So the arguments int *ptr and int prt[size] have no difference
as function arguments. In other words, both the functoins have the same signature and
so cannot be overloaded.
*/

class some{
public:
        ~some()
        {
                cout<<"some's destructor"<<endl;
        }
};

void main()
{
        some s;
        s.~some();
}
/*
Answer:
        some's destructor
        some's destructor
Explanation:
        Destructors can be called explicitly. Here 's.~some()' explicitly calls the
destructor of 's'. When main() returns, destructor of s is called again,
hence the result.
*/

#include <iostream.h>

class fig2d
{
        int dim1;
        int dim2;

public:
        fig2d() { dim1=5; dim2=6;}

        virtual void operator<<(ostream & rhs);
};

void fig2d::operator<<(ostream &rhs)
{
```

```
        rhs <<this->dim1<<" "<<this->dim2<<" ";
}

/*class fig3d : public fig2d
{
        int dim3;
public:
        fig3d() { dim3=7;}
        virtual void operator<<(ostream &rhs);
};
void fig3d::operator<<(ostream &rhs)
{
        fig2d::operator <<(rhs);
        rhs<<this->dim3;
}
*/

void main()
{
        fig2d obj1;
//      fig3d obj2;

        obj1 << cout;
//      obj2 << cout;
}
/*
Answer :
        5 6
Explanation:
        In this program, the << operator is overloaded with ostream as argument.
This enables the 'cout' to be present at the right-hand-side. Normally, 'cout'
is implemented as global function, but it doesn't mean that 'cout' is not possible
to be overloaded as member function.
   Overloading << as virtual member function becomes handy when the class in which
it is overloaded is inherited, and this becomes available to be overrided. This is as opposed
to global friend functions, where friend's are not inherited.
*/

class opOverload{
public:
        bool operator==(opOverload temp);
};

bool opOverload::operator==(opOverload temp){
        if(*this  == temp ){
                cout<<"The both are same objects\n";
                return true;
        }
        else{
                cout<<"The both are different\n";
```

```
                return false;
        }
}

void main(){
        opOverload a1, a2;
        a1= =a2;
}
```

Answer :
        Runtime Error: Stack Overflow
Explanation :
        Just like normal functions, operator functions can be called recursively. This program just illustrates that point, by calling the operator == function recursively, leading to an infinite loop.

```
class complex{
        double re;
        double im;
public:
        complex() : re(1),im(0.5) {}
        bool operator==(complex &rhs);
        operator int(){}
};

bool complex::operator == (complex &rhs){
        if((this->re == rhs.re) && (this->im == rhs.im))
                return true;
        else
                return false;
}

int main(){
        complex  c1;
        cout<<  c1;
}
```

Answer : Garbage value

Explanation:
        The programmer wishes to print the complex object using output re-direction operator,which he has not defined for his lass.But the compiler instead of giving an error sees the conversion function
and converts the user defined object to standard object and prints
some garbage value.

```
class complex{
        double re;
```

```
        double im;
public:
        complex() : re(0),im(0) {}
        complex(double n) { re=n,im=n;};
        complex(int m,int n) { re=m,im=n;}
        void print() { cout<<re; cout<<im;}
};

void main(){
        complex c3;
        double i=5;
        c3 = i;
        c3.print();
}
```

Answer:

5,5

Explanation:

Though no operator= function taking complex, double is defined, the double on the rhs is converted into a temporary object using the single argument constructor taking double and assigned to the lvalue.

```
void main()
{
        int a, *pa, &ra;
        pa = &a;
        ra = a;
        cout <<"a="<<a <<"*pa="<<*pa <<"ra"<<ra ;
}
```

Answer :

Compiler Error: 'ra',reference must be initialized

Explanation :

Pointers are different from references. One of the main differences is that the pointers can be both initialized and assigned, whereas references can only be initialized. So this code issues an error.

Try it Yourself

1) Determine the output of the 'C++' Codelet.

```
        class base
        {
        public :
                out()
                {
                        cout<<"base ";
                }
        };
        class deri{
```

```
public : out()
{
cout<<"deri ";
}
};
void main()
{       deri dp[3];
        base *bp = (base*)dp;
        for (int i=0; i<3;i++)
        (bp++)->out();
}
```

2) Justify the use of virtual constructors and destructors in C++.

3) Each C++ object possesses the 4 member fns,(which can be declared by the programmer explicitly or by the implementation if they are not available). What are those 4 functions?

4) What is wrong with this class declaration?

```
class something
{
        char *str;
        public:
          something(){
          st = new char[10]; }
         ~something()
         {
                delete str;
         }
 };
```

5) Inheritance is also known as -------- relationship. Containership as _____ relationship.

6) When is it necessary to use member-wise initialization list (also known as header initialization list) in C++?

7) Which is the only operator in C++ which can be overloaded but NOT inherited.

8) Is there anything wrong with this C++ class declaration?

```
class temp
{
 int value1;
 mutable int value2;
 public :
        void fun(int val)
        const{
        ((temp*) this)->value1 = 10;
        value2 = 10;
        }
 };
```

1. What is a modifier?

Answer:

A modifier, also called a modifying function is a member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as 'mutators'.

2. What is an accessor?

Answer:

An accessor is a class operation that does not modify the state of an object. The accessor functions need to be declared as *const* operations

3. Differentiate between a template class and class template.

Answer:

Template class:

A generic definition or a parameterized class not instantiated until the client provides the needed information. It's jargon for plain templates.

Class template:

A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. It's jargon for plain classes.

4. When does a name clash occur?

Answer:

A name clash occurs when a name is defined in more than one place. For example., two different class libraries could give two different classes the same name. If you try to use many class libraries at the same time, there is a fair chance that you will be unable to compile or link the program because of name clashes.

5. Define namespace.

Answer:

It is a feature in c++ to minimize name collisions in the global name space. This namespace keyword assigns a distinct name to a library that allows other libraries to use the same identifier names without creating any name collisions. Furthermore, the compiler uses the namespace signature for differentiating the definitions.

6. What is the use of 'using' declaration.

Answer:

A using declaration makes it possible to use a name from a namespace without the scope operator.

7. What is an Iterator class?

Answer:

A class that is used to traverse through the objects maintained by a container class. There are five categories of iterators:

- input iterators,
- output iterators,
- forward iterators,
- bidirectional iterators,
- random access.

An iterator is an entity that gives access to the contents of a container object without violating encapsulation constraints. Access to the contents is granted on a one-at-a-time basis

in order. The order can be storage order (as in lists and queues) or some arbitrary order (as in array indices) or according to some ordering relation (as in an ordered binary tree). The iterator is a construct, which provides an interface that, when called, yields either the next element in the container, or some value denoting the fact that there are no more elements to examine. Iterators hide the details of access to and update of the elements of a container class.

The simplest and safest iterators are those that permit read-only access to the contents of a container class. The following code fragment shows how an iterator might appear in code:

```
cont_iter:=new cont_iterator();
x:=cont_iter.next();
while x/=none do
    ...
    s(x);
    ...
    x:=cont_iter.next();
end;
```

In this example, cont_iter is the name of the iterator. It is created on the first line by instantiation of cont_iterator class, an iterator class defined to iterate over some container class, cont. Succesive elements from the container are carried to x. The loop terminates when x is bound to some empty value. (Here, none)In the middle of the loop, there is s(x) an operation on x, the current element from the container. The next element of the container is obtained at the bottom of the loop.

9. List out some of the OODBMS available.

Answer:

- GEMSTONE/OPAL of Gemstone systems.
- ONTOS of Ontos.
- Objectivity of Objectivity inc.
- Versant of Versant object technology.
- Object store of Object Design.
- ARDENT of ARDENT software.
- POET of POET software.

10. List out some of the object-oriented methodologies.

Answer:

- Object Oriented Development (OOD) (Booch 1991,1994).
- Object Oriented Analysis and Design (OOA/D) (Coad and Yourdon 1991).
- Object Modelling Techniques (OMT) (Rumbaugh 1991).
- Object Oriented Software Engineering (Objectory) (Jacobson 1992).
- Object Oriented Analysis (OOA) (Shlaer and Mellor 1992).
- The Fusion Method (Coleman 1991).

11. What is an incomplete type?

Answer:

Incomplete types refers to pointers in which there is non availability of the implementation of the referenced location or it points to some location whose value is not available for modification.

Example:

```
int *i=0x400  // i points to address 400
*i=0;        //set the value of memory location pointed by i.
```

Incomplete types are otherwise called uninitialized pointers.

12. What is a dangling pointer?

Answer:

A dangling pointer arises when you use the address of an object after its lifetime is over.

This may occur in situations like returning addresses of the automatic variables from a function or using the address of the memory block after it is freed.

13. Differentiate between the message and method.

Answer:

Message	***Method***
Objects communicate by sending messages to each other.	Provides response to a message.
A message is sent to invoke a method.	It is an implementation of an operation.

14. What is an adaptor class or Wrapper class?

Answer:

A class that has no functionality of its own. Its member functions hide the use of a third party software component or an object with the non-compatible interface or a non-object- oriented implementation.

15. What is a Null object?

Answer:

It is an object of some class whose purpose is to indicate that a real object of that class does not exist. One common use for a null object is a return value from a member function that is supposed to return an object with some specified properties but cannot find such an object.

16. What is class invariant?

Answer:

A class invariant is a condition that defines all valid states for an object. It is a logical condition to ensure the correct working of a class. Class invariants must hold when an object is created, and they must be preserved under all operations of the class. In particular all class invariants are both preconditions and post-conditions for all operations or member functions of the class.

17. What do you mean by Stack unwinding?

Answer:

It is a process during exception handling when the destructor is called for all local objects between the place where the exception was thrown and where it is caught.

18. Define precondition and post-condition to a member function.

Answer:

Precondition:

A precondition is a condition that must be true on entry to a member function. A class is used correctly if preconditions are never false. An operation is not responsible for doing anything sensible if its precondition fails to hold.

For example, the interface invariants of *stack class* say nothing about pushing yet another element on a stack that is already full. We say that *isful()* is a precondition of the *push* operation.

Post-condition:

A post-condition is a condition that must be true on exit from a member function if the precondition was valid on entry to that function. A class is implemented correctly if post-conditions are never false.

For example, after pushing an element on the stack, we know that *isempty()* must necessarily hold. This is a post-condition of the *push* operation.

19. What are the conditions that have to be met for a condition to be an invariant of the class?

Answer:

- The condition should hold at the end of every constructor.
- The condition should hold at the end of every mutator(non-const) operation.

20. What are proxy objects?

Answer:

Objects that stand for other objects are called proxy objects or surrogates.

Example:

```
template<class T>
class Array2D
{
    public:
        class Array1D
        {
            public:
          T& operator[] (int index);
          const T& operator[] (int index) const;
          ...
        };
        Array1D operator[] (int index);
        const Array1D operator[] (int index) const;
        ...
};
```

The following then becomes legal:

```
Array2D<float>data(10,20);
.....
cout<<data[3][6];    // fine
```

Here data[3] yields an Array1D object and the operator [] invocation on that object yields the float in position(3,6) of the original two dimensional array. Clients of the Array2D class need not be aware of the presence of the Array1D class. Objects of this latter class stand for one-dimensional array objects that, conceptually, do not exist for clients of Array2D. Such clients program as if they were using real, live, two-dimensional arrays. Each Array1D object stands for a one-dimensional array that is absent from a conceptual model used by the clients of Array2D. In the above example, Array1D is a proxy class. Its instances stand for one-dimensional arrays that, conceptually, do not exist.

21. Name some pure object oriented languages.

Answer:

- Smalltalk,
- Java,
- Eiffel,
- Sather.

22. Name the operators that cannot be overloaded.

Answer:

sizeof . .* .-> :: ?:

23. What is a node class?

Answer:

A node class is a class that,

- relies on the base class for services and implementation,
- provides a wider interface to te users than its base class,
- relies primarily on virtual functions in its public interface
- depends on all its direct and indirect base class
- can be understood only in the context of the base class
- can be used as base for further derivation
- can be used to create objects.

A node class is a class that has added new services or functionality beyond the services inherited from its base class.

24. What is an orthogonal base class?

Answer:

If two base classes have no overlapping methods or data they are said to be independent of, or orthogonal to each other. Orthogonal in the sense means that two classes operate in different dimensions and do not interfere with each other in any way. The same derived class may inherit such classes with no difficulty.

25. What is a container class? What are the types of container classes?

Answer:

A container class is a class that is used to hold objects in memory or external storage. A container class acts as a generic holder. A container class has a predefined behavior and a well-known interface. A container class is a supporting class whose purpose is to hide the topology used for maintaining the list of objects in memory. When a container class contains a group of mixed objects, the container is called a heterogeneous container; when the container is holding a group of objects that are all the same, the container is called a homogeneous container.

26. What is a protocol class?

Answer:

An abstract class is a protocol class if:

- it neither contains nor inherits from classes that contain member data, non-virtual functions, or private (or protected) members of any kind.
- it has a non-inline virtual destructor defined with an empty implementation,
- all member functions other than the destructor including inherited functions, are declared pure virtual functions and left undefined.

27. What is a mixin class?

Answer:

A class that provides some but not all of the implementation for a virtual base class is often called mixin. Derivation done just for the purpose of redefining the virtual functions in the base classes is often called mixin inheritance. Mixin classes typically don't share common bases.

28. What is a concrete class?

Answer:

A concrete class is used to define a useful object that can be instantiated as an automatic variable on the program stack. The implementation of a concrete class is defined. The concrete class is not intended to be a base class and no attempt to minimize dependency on other classes in the implementation or behavior of the class.

29.What is the handle class?

Answer:

A handle is a class that maintains a pointer to an object that is programmatically accessible through the public interface of the handle class.

Explanation:

In case of abstract classes, unless one manipulates the objects of these classes through pointers and references, the benefits of the virtual functions are lost. User code may become dependent on details of implementation classes because an abstract type cannot be allocated statistically or on the stack without its size being known. Using pointers or references implies that the burden of memory management falls on the user. Another limitation of abstract class object is of fixed size. Classes however are used to represent concepts that require varying amounts of storage to implement them.

A popular technique for dealing with these issues is to separate what is used as a single object in two parts: a handle providing the user interface and a representation holding all or most of the object's state. The connection between the handle and the representation is typically a pointer in the handle. Often, handles have a bit more data than the simple representation pointer, but not much more. Hence the layout of the handle is typically stable, even when the representation changes and also that handles are small enough to move around relatively freely so that the user needn't use the pointers and the references.

30. What is an action class?

Answer:

The simplest and most obvious way to specify an action in C++ is to write a function. However, if the action has to be delayed, has to be transmitted 'elsewhere' before being performed, requires its own data, has to be combined with other actions, etc then it often becomes attractive to provide the action in the form of a class that can execute the desired action and provide other services as well. Manipulators used with iostreams is an obvious example.

Explanation:

A common form of action class is a simple class containing just one virtual function.

```
class Action
{
    public:
        virtual int do_it( int )=0;
        virtual ~Action( );
```

```
    }
```

Given this, we can write code say a member that can store actions for later execution without using pointers to functions, without knowing anything about the objects involved, and without even knowing the name of the operation it invokes. For example:

```
class write_file : public Action
  {
        File& f;
        public:
          int do_it(int)
          {
              return fwrite( ).suceed( );
          }
   };
  class error_message: public Action
  {
         response_box db(message.cstr( ),"Continue","Cancel","Retry");
         switch (db.getresponse( ))
         {
              case 0: return 0;
              case 1: abort();
              case 2: current_operation.redo( );return 1;
         }
   };
```

A user of the Action class will be completely isolated from any knowledge of derived classes such as write_file and error_message.

31. When can you tell that a memory leak will occur?

Answer:

A memory leak occurs when a program loses the ability to free a block of dynamically allocated memory.

32.What is a parameterized type?

Answer:

A template is a parameterized construct or type containing generic code that can use or manipulate any type. It is called parameterized because an actual type is a parameter of the code body. Polymorphism may be achieved through parameterized types. This type of polymorphism is called parameteric polymorphism. Parameteric polymorphism is the mechanism by which the same code is used on different types passed as parameters.

33. Differentiate between a deep copy and a shallow copy?

Answer:

Deep copy involves using the contents of one object to create another instance of the same class. In a deep copy, the two objects may contain ht same information but the target object will have its own buffers and resources. the destruction of either object will not affect the remaining object. The overloaded assignment operator would create a deep copy of objects.

Shallow copy involves copying the contents of one object into another instance of the same class thus creating a mirror image. Owing to straight copying of references and

pointers, the two objects will share the same externally contained contents of the other object to be unpredictable.

Explanation:

Using a copy constructor we simply copy the data values member by member. This method of copying is called shallow copy. If the object is a simple class, comprised of built in types and no pointers this would be acceptable. This function would use the values and the objects and its behavior would not be altered with a shallow copy, only the addresses of pointers that are members are copied and not the value the address is pointing to. The data values of the object would then be inadvertently altered by the function. When the function goes out of scope, the copy of the object with all its data is popped off the stack.

If the object has any pointers a deep copy needs to be executed. With the deep copy of an object, memory is allocated for the object in free store and the elements pointed to are copied. A deep copy is used for objects that are returned from a function.

34. What is an opaque pointer?

Answer:

A pointer is said to be opaque if the definition of the type to which it points to is not included in the current translation unit. A translation unit is the result of merging an implementation file with all its headers and header files.

35. What is a smart pointer?

Answer:

A smart pointer is an object that acts, looks and feels like a normal pointer but offers more functionality. In C++, smart pointers are implemented as *template classes* that encapsulate a pointer and override standard pointer operators. They have a number of advantages over regular pointers. They are guaranteed to be initialized as either null pointers or pointers to a heap object. Indirection through a null pointer is checked. No delete is ever necessary. Objects are automatically freed when the last pointer to them has gone away. One significant problem with these smart pointers is that unlike regular pointers, they don't respect inheritance. Smart pointers are unattractive for polymorphic code. Given below is an example for the implementation of smart pointers.

Example:

```
template <class X>
class smart_pointer
{
        public:
            smart_pointer();                    // makes a null pointer
            smart_pointer(const X& x)           // makes pointer to copy of x

            X& operator *( );
            const X& operator*( ) const;
            X* operator->() const;

            smart_pointer(const smart_pointer <X> &);
            const smart_pointer <X> & operator =(const smart_pointer<X>&);
            ~smart_pointer();
        private:
            //...
 };
```

This class implement a smart pointer to an object of type X. The object itself is located on the heap. Here is how to use it:

```
smart_pointer <employee> p= employee("Harris",1333);
```

Like other overloaded operators, p will behave like a regular pointer,

```
cout<<*p;
p->raise_salary(0.5);
```

36. What is reflexive association?

Answer:

The 'is-a' is called a reflexive association because the reflexive association permits classes to bear the is-a association not only with their super-classes but also with themselves. It differs from a 'specializes-from' as 'specializes-from' is usually used to describe the association between a super-class and a sub-class. For example:

Printer is-a printer.

37. What is slicing?

Answer:

Slicing means that the data added by a subclass are discarded when an object of the subclass is passed or returned by value or from a function expecting a base class object.

Explanation:

Consider the following class declaration:

```
class base
{
    ...
    base& operator =(const base&);
    base (const base&);
}
void fun( )
{
    base e=m;
    e=m;
}
```

As base copy functions don't know anything about the derived only the base part of the derived is copied. This is commonly referred to as slicing. One reason to pass objects of classes in a hierarchy is to avoid slicing. Other reasons are to preserve polymorphic behavior and to gain efficiency.

38. What is name mangling?

Answer:

Name mangling is the process through which your c++ compilers give each function in your program a unique name. In C++, all programs have at-least a few functions with the same name. Name mangling is a concession to the fact that linker always insists on all function names being unique.

Example:

In general, member names are made unique by concatenating the name of the member with that of the class e.g. given the declaration:

```
class Bar
{
    public:
      int ival;
```

```
        ...
};
```

ival becomes something like:

```
// a possible member name mangling
ival__3Bar
```

Consider this derivation:

```
class Foo : public Bar
{
    public:
        int ival;
        ...
}
```

The internal representation of a Foo object is the concatenation of its base and derived class members.

```
// Pseudo C++ code
// Internal representation of Foo
class Foo
{
    public:
        int ival__3Bar;
        int ival__3Foo;
        ...
};
```

Unambiguous access of either ival members is achieved through name mangling. Member functions, because they can be overloaded, require an extensive mangling to provide each with a unique name. Here the compiler generates the same name for the two overloaded instances(Their argument lists make their instances unique).

39. What are proxy objects?

Answer:

Objects that points to other objects are called proxy objects or surrogates. Its an object that provides the same interface as its server object but does not have any functionality. During a method invocation, it routes data to the true server object and sends back the return value to the object.

40. Differentiate between declaration and definition in C++.

Answer:

A declaration introduces a name into the program; a definition provides a unique description of an entity (e.g. type, instance, and function). Declarations can be repeated in a given scope, it introduces a name in a given scope. There must be exactly one definition of every object, function or class used in a C++ program.

A declaration is a definition unless:

- it declares a function without specifying its body,
- it contains an extern specifier and no initializer or function body,
- it is the declaration of a static class data member without a class definition,
- it is a class name definition,
- it is a typedef declaration.

A definition is a declaration unless:

- it defines a static class data member,
- it defines a non-inline member function.

41. What is cloning?

Answer:

An object can carry out copying in two ways i.e. it can set itself to be a copy of another object, or it can return a copy of itself. The latter process is called cloning.

42. Describe the main characteristics of static functions.

Answer:

The main characteristics of static functions include,

- It is without the a this pointer,
- It can't directly access the non-static members of its class
- It can't be declared const, volatile or virtual.
- It doesn't need to be invoked through an object of its class, although for convenience, it may.

43. Will the inline function be compiled as the inline function always? Justify.

Answer:

An inline function is a request and not a command. Hence it won't be compiled as an inline function always.

Explanation:

Inline-expansion could fail if the inline function contains loops, the address of an inline function is used, or an inline function is called in a complex expression. The rules for inlining are compiler dependent.

44. Define a way other than using the keyword inline to make a function inline.

Answer:

The function must be defined inside the class.

45. How can a '::' operator be used as unary operator?

Answer:

The scope operator can be used to refer to members of the global namespace. Because the global namespace doesn't have a name, the notation :: member-name refers to a member of the global namespace. This can be useful for referring to members of global namespace whose names have been hidden by names declared in nested local scope. Unless we specify to the compiler in which namespace to search for a declaration, the compiler simple searches the current scope, and any scopes in which the current scope is nested, to find the declaration for the name.

46. What is placement new?

Answer:

When you want to call a constructor directly, you use the placement new. Sometimes you have some raw memory that's already been allocated, and you need to construct an object in the memory you have. Operator new's special version placement new allows you to do it.

```
class Widget
{
   public :
      Widget(int widgetsize);
       ...
       Widget* Construct_widget_int_buffer(void *buffer,int widgetsize)
        {
            return new(buffer) Widget(widgetsize);
```

```
        }
};
```

This function returns a pointer to a Widget object that's constructed within the buffer passed to the function. Such a function might be useful for applications using shared memory or memory-mapped I/O, because objects in such applications must be placed at specific addresses or in memory allocated by special routines.