

## C++ and Object Oriented Programming

CS 72

**Q.No-1**

**Ans**

Any data which is declared **private** inside a class is not accessible from outside the class. A function which is not a member or an external class can never access such private data.

But there may be some cases A class can allow non-member functions and other classes to access its own private data, by making them as **friends**. two important points.

- Once a non-member function is declared as a friend, it can access the private data of the class
- similarly when a class is declared as a friend, the friend class can have access to the private data of the class which made this a friend

**Friend function sample:**

```
#include <iostream.h>
//Declaration of the function to be made as friend for the C++ Tutorial sample
int AddToFriend(int x);
class CPP_Tutorial
{
    int private_data;
    friend int AddToFriend(int x);
public:
    CPP_Tutorial()
    {
        private_data = 5;
    }
};
int AddToFriend(int x)
{
    CPP_Tutorial var1;
    return var1.private_data + x;
}
int main()
{
    cout << "Added Result for this C++ tutorial: " << AddToFriend(4) << endl;
}
```

**friend class:**

```
#include <iostream.h>
class CPP_Tutorial
{
    int private_data;
    friend class friendclass;
public:
    CPP_Tutorial()
    {
        private_data = 5;
    }
}
```

```

};
class friendclass
{
public:
    int subtractfrom(int x)
    {
        CPP_Tutorial var2;
        return var2.private_data - x;
    }
};
int main()
{
    friendclass var3;
    cout << "Added Result for this C++ tutorial: "<< var3.subtractfrom(2)<<ENDL;
}

```

### Q.No-2

#### Ans

In object-oriented programming (OOP), a **virtual function** or **virtual method** is a function whose behavior, by virtue of being declared "virtual," is determined by the definition of a function with the same signature furthest in the inheritance lineage of the instantiated object on which it is called. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

```

class B {
    virtual void a_pure_virtual_function() = 0;
};
void Abstract::pure_virtual() {
    // do something
}

```

```

class Child : Abstract {
    virtual void pure_virtual(); // no longer abstract, this class may be
                                // instantiated.
};

```

```

void Child::pure_virtual() {
    Abstract::pure_virtual(); // the implementation in the abstract class
                                // is executed
}

```

The following is an example in C++:

```

#include <iostream>
using namespace std;
class Animal
{
public:
    virtual void eat() { cout << "I eat like a generic Animal.\n"; }
};

```

```

class Wolf : public Animal
{
public:
    void eat() { cout << "I eat like a wolf!\n"; }
};

```

```

class Fish : public Animal
{
public:
    void eat() { cout << "I eat like a fish!\n"; }
};

```

```

class OtherAnimal : public Animal
{
};

```

```

int main()
{
    Animal *anAnimal[4];
    anAnimal[0] = new Animal();
    anAnimal[1] = new Wolf();
    anAnimal[2] = new Fish();
    anAnimal[3] = new OtherAnimal();

    for(int i = 0; i < 4; i++)
        anAnimal[i]->eat();
}

```

Output with the virtual method eat:

I eat like a generic Animal.

I eat like a wolf!

I eat like a fish!

I eat like a generic Animal.

Output without the virtual method eat:

I eat like a generic Animal.

I eat like a generic Animal.

I eat like a generic Animal.

I eat like a generic Animal.

### Q.No-3

**Ans**

**Inline functions**      Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Inline functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an inline function must be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file.

Inline functions are safer and easier to use than macros if you

```
// need an ordinary function that would have been unacceptable for
// efficiency reasons.
// They are also easier to convert to ordinary functions later on.
```

```
#include <iostream.h>
int exforsys(int);
void main()
{
int x;
cout << "\n Enter the Input Value: ";
cin>>x;
cout<<"\n The Output is: " << exforsys(x);
}
inline int exforsys(int x1)
{
return 5*x1;
}
```

The output of the above program is:  
Enter the Input Value: 10  
The Output is: 50

#### **Q.No-4**

#### **Ans**

**Polymorphism**        The meaning of Polymorphism is something like one name many forms. Polymorphism enables one entity to be used as a general category for different types of actions. The specific action is determined by the exact nature of the situation. The concept of polymorphism can be explained as "one interface, multiple methods".

- C++ provides a simple mechanism for class<sup>□</sup>-based polymorphism<sup>□</sup>
- Member<sup>□</sup> functions can be marked as polymorphic using the **virtual** keyword
- Calls to such functions then get dispatched according to the actual type of the object which owns them, rather than the apparent type of the pointer (or reference<sup>□</sup>) through which they are invoked

#### **Virtual functions<sup>□</sup>**

- A member<sup>□</sup> function which is declared with the **virtual** keyword is called a *virtual function<sup>□</sup>*
- Once a function is declared virtual, it may be redefined in derived classes<sup>□□</sup> (and is virtual in those classes<sup>□</sup> too)
- When the compiler sees a call to a virtual function<sup>□</sup> via a pointer or reference<sup>□</sup>, it calls the correct version for the object in question (rather than the version indicated by the type of the pointer or reference<sup>□</sup>)

#### **An example**

```
class Vehicle
{
public:
```

```

        Vehicle(char* regnum)
        : myRegNum(strdup(regnum))
        {}

        ~Vehicle(void)
        { delete[] myRegNum; }

        virtual void Describe(void)
        {
            cout << "Unknown vehicle, registration "
                << myRegNum << endl;
        }

protected:
        char* myRegNum;
};

class Car : public Vehicle
{
public:
        Car(char* make, char* regnum)
        : Vehicle(regnum), myMake( strdup(make) )
        {}

        ~Car(void)
        { delete[] myMake; }

        virtual void Describe(void)
        {
            cout << "Car (" << myMake
                << "), registration "
                << myRegNum << endl;
        }

protected:
        char* myMake;
};

Vehicle* vp1 = new Car ("Jaguar", "XJS 012");
Vehicle* vp2 = new Vehicle ("SGI 987");
Vehicle* vp3 = new Vehicle ("ABC 123");

vp1->Describe();           // PRINTS "Car (Jaguar)....."
vp2->Describe();           // PRINTS "Unknown vehicle....."
vp3->Describe();           // PRINTS "Unknown vehicle....."

```

**Q.No-5**

**Ans**

- seekg()

Sets the position of the *get pointer*.

The *get pointer* determines the next location to be read in the source associated to the stream.

### Parameters

#### pos

The new position in the stream buffer. This parameter is an integral value of type streampos.

#### off

Integral value of type streamoff representing the offset to be applied relative to an absolute position specified in the *dir* parameter.

#### dir

Seeking direction. It is an object of type ios\_base::seekdir that specifies an absolute position from where the offset parameter *off* is applied. It can take any of the following member constant values:

### Example

```
// load a file into memory
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    int length;
    char * buffer;

    ifstream is;
    is.open ("test.txt", ios::binary );

    // get length of file:
    is.seekg (0, ios::end);
    length = is.tellg();
    is.seekg (0, ios::beg);

    // allocate memory:
    buffer = new char [length];

    // read data as a block:
    is.read (buffer,length);

    is.close();

    cout.write (buffer,length);

    delete[] buffer;
    return 0;
}
```

- seekp()

Sets the position of the *put pointer*.

The *put pointer* determines the location in the output sequence where the next output operation is going to take place.

#### Parameters

##### **pos**

The new position in the stream buffer. This parameter is an integral value of type `streampos`.

##### **off**

Integral value of type `streamoff` representing the offset to be applied relative to an absolute position specified in the *dir* parameter.

##### **dir**

Seeking direction. It is an object of type `ios_base::seekdir` that specifies an absolute position from where the offset parameter *off* is applied. It can take any of the following member constant values:

#### Example

```
// position of put pointer
#include <fstream>
using namespace std;

int main () {
    long pos;

    ofstream outfile;
    outfile.open ("test.txt");

    outfile.write ("This is an apple",16);
    pos=outfile.tellp();
    outfile.seekp (pos-7);
    outfile.write (" sam",4);

    outfile.close();

    return 0;
}
```

```
tellg ( );
```

#### **Get position of the get pointer.**

Returns the absolute position of the *get pointer*.

The *get pointer* determines the next location in the input sequence to be read by the next input operation.

#### Parameters

none

#### Example

```
// read a file into memory
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    int length;
    char * buffer;
```

```

ifstream is;
is.open ("test.txt", ios::binary );

// get length of file:
is.seekg (0, ios::end);
length = is.tellg();
is.seekg (0, ios::beg);

// allocate memory:
buffer = new char [length];

// read data as a block:
is.read (buffer,length);

is.close();

cout.write (buffer,length);

return 0;
}

```

- tellp()

### **Get position of put pointer**

Returns the absolute position of the *put pointer*.

The put pointer determines the location in the output sequence where the next output operation is going to take place.

#### **Parameters**

none

#### **Example**

```

// position of put pointer
#include <fstream>
using namespace std;

int main () {
    long pos;

    ofstream outfile;
    outfile.open ("test.txt");

    outfile.write ("This is an apple",16);
    pos=outfile.tellp();
    outfile.seekp (pos-7);
    outfile.write (" sam",4);

    outfile.close();

    return 0;
}

```

```
*****/
```

## Q.No-6

Ans

```
#ifndef I4_BINARY_TREE_HH
#define I4_BINARY_TREE_HH
```

```
/*
```

```
pretty simple, uses linear_allocator for quick allocation
two methods, ::insert & ::find so far, you need to define the > & <
operators for the type if they don't exist.
```

example usage:

```
-----
```

```
i4_binary_tree<int> tree;
tree.insert(6);
tree.insert(10);
if (tree.find(12)) x=5;
```

or for user defined structs :

```
struct complex
```

```
{
    float a;
    float b;
```

```
    i4_bool operator>(const complex c) const { return (i4_bool)(a>c.b); }
    i4_bool operator<(const complex b) const { return (i4_bool)(a<c.b); }
};
```

```
i4_binary_tree<complex> tree2;
tree2.insert(complex(3.1, 4.5));
tree2.find(complex(3.1,0));
*/
```

```
#include "memory/lalloc.hh"
```

```
template <class T>
class i4_binary_tree
{
    void recursive_delete(T * node)
    {
        if (node->right)
            recursive_delete(node->right);
```

```

    if (node->left)
        recursive_delete(node->left);
    delete node;
}

```

```

public:
    T *root;

```

```

i4_binary_tree() { root=0; }

```

```

// insert return a previous instance if already in tree

```

```

T *insert(T *x)

```

```

{
    x->left=x->right=0;

    if (!root)
        root=x;
    else
    {
        T *p=root;

        while (1)
        {
            int cmp=x->compare(p);
            if (cmp<0)
            {
                if (!p->left)
                {
                    p->left=x;
                    return x;
                }
                else p=p->left;
            }
            else if (cmp>0)
            {
                if (!p->right)
                {
                    p->right=x;
                    return x;
                }
                else p=p->right;
            }
            else return p;
        }
    }

    return x;
}

```

```

T *find(const T *x)
{
    T *p=root;
    while (1)
    {
        if (!p) return 0;

        int cmp=x->compare(p);
        if (cmp<0)
            p=p->left;
        else if (cmp>0)
            p=p->right;
        else return p;
    }
}

void delete_tree()
{
    if (root)
        recursive_delete(root);
}

};

#endif

```

### **Q.No-7**

#### **Ans**

```

class student
{
private:
int age;
char name[20];
public:
student()
{
age=0;
}

void getdata(char nm[], int ag)
{
strcpy(name,nm);
age=ag;
}
void print()
{
cout<<"name"<<"\t"<<"age"<<endl
cout<<name<<"\t"<<age

```

```

}
};
void main()
{
char nm[44];
int ag;
cout<<"enter name :---";
cin>>nm;
cout<<"enter age:---";
cin>>ag;
student obj;
obj.getdata(nm,ag);
obj.print();
getch();
}

```

### **Q.No-8**

**Ans**

```

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <values.h>
#include <time.h>
#include <string.h>
#include <conio.h>
class student
{
protected:
int roll_no;
public:
void getnum()
{
cout<<"Enter Rollno:---";
cin>>roll_no;
}
void putnum()
{
cout<<roll_no<<"\t";
}
};
//*****
class test:public student
{
protected:
int marks;
public:
void getmarks()
{
getnum();
}
}

```

```

cout<<"Enter marks:---";
cin>>marks;
}
void putmarks()
{
putnum();
cout<<marks<<"\t";
}
};
//*****
class sport :public student
{
protected:
int scor;
public:
void getscore()
{

cout<<"Enter getscore:---";
cin>>scor;
}
void putscore()
{
cout<<scor<<"\t";
}
};
class result:public test,public sport,public student
{
int total;
public:
void display()
{

total=marks+scor;
cout<<"\t"<<marks<<"\t"<<scor<<"\t"<<total;
}
};

void main()
{
result obj;
clrscr();
obj.getmarks();
obj.getscore();

obj.putmarks();
obj.putscore();
obj.display();

getch();

```

}