Lab Sheet 4
Additional Components of a Class

Constructor

A constructor is basically a function used for initialization of the class data members, allocation of memory, and so on. It is convenient if an object can initialize itself when it is first created, without the need to make a separate call to a member function. It has the same name as the class in which they are members. No return type is used for constructors. Since the constructor is called automatically by the system there is no reason for it to return anything. Compiler knows that they are constructors by looking at their name and return type. Let us take an example that each time an object of the date class is created, it is to be initialized with the date 2/02/2011.

Examples:

```
class date{
 private:
 int dd,yy;
 char mm[3];
 public:
    date()     // constructor
    {   dd=2,mm[0]='0',mm[1]='2';
        mm[2]='\0', yy=2011;
    }
    showdate()  //display
    {
        cout<<"\n"<<dd<<"/"<<mm<<"/"<<yy;
```

```
    }
};
int main()
{
   date d1; //define and initialize
   clrscr();
   d1.showdate();
   return 0;
```

```
 }
```

Here as soon as an object d1 is created, the constructor will be involved and the values dd, mm, and yy initialized to the required date- 2/02/2011. Note that the constructor here neither takes any parameters nor does it return values. So it is called default constructor. Data members of object can be initialized by passing arguments to the constructor when the object is created. The constructor that takes arguments is called parameterized constructor. Following example adds parameterized constructor in the above example program.

```
class date{

 //....

public:

     date()    // constructor
     {dd=2,mm[0]='0',mm[1]='2';

         mm[2]='\0', yy=2011;

     }

     date(int d, char m[], int y) //parameterized
     constructor
     {dd=d,mm[0]=m[0],mm[1]=m[1];

         mm[2]='\0', yy=y;

     }

   //......

 };

int main()

{

    date d1(2,"02",2011); //define and initialize
```

```
    d1.showdate();
    return 0;
}
```

Copy Constructor

We already know that no argument constructor can initialize data members to constant values, and parameterized constructor can initialize data members to values passed as arguments. There is also

another way to initialize an object with another object of the same type. It is called the *default copy constructor*. It is a one-argument constructor whose argument is an object of same class.
Example:

```
class date

{

 private:

    int dd,yy;

    int mm;

 public:

   date()

   { dd=26, mm=1,yy=2004; }

   date(int ee,int nn,int zz):
dd(ee),mm(nn),yy(zz) //parameterized Constructor

     { }

   showdat

   e()

   { cout<<"\n"<<dd<<"/"<<mm<<"/"<<yy;}

};

int main()

{

       d1(26,2,20
  date d2(d1 04);

  date );   //copy constructor call: it can also be
 writt as   d2=d1
   en date     ;
```

```
  d1.showdate();
  d2.showdate();
  return 0;
}
```

Destructor

Constructors serve to automatically initialize data members of an object at the point of creation. Destructors are complimentary to constructors. They serve to cleanup objects when they are destroyed.

A destructor may be called either when the object goes out of scope or when it is destroyed explicitly using the delete operator. A destructor, like a constructor has the same name as that of the class, but is prefixed by the tilde ('~') symbol. A class cannot have more than one destructor. And destructor can't take arguments or specify a return value. The most common use of destructor is to de-allocate memory that was allocated for the object by the constructor. Destructor for above program is

```
~date() //destructor
{   }
```

Constant Member Function

A function is made a constant function by placing the keyword const after the function header before function body. Member function that do not change the value of its object but acquire data from their object are obvious candidates for constant function.

General syntax:

```
return_type func_name(argument list) const; //const
function declaration
//.....
return_type func_name(argument     //const function
list) const definition
{
   //Function body;
```

}

Constant Object

When an object is declared as constant, we can't modify it. The constant objects can only use constant member functions with it, because they are the only ones that guarantee not to modify its value.

General syntax:

```
const class_name object_name; //creation of constant object
```

Constant Reference Argument

When we don't want to modify the arguments passed to the function, the reference parameters are made const in the function declaration and definition.

General syntax:

```
return_type func_name(const int&, float&); const member function arguments
//......

return_type func_name(const int&, float&) const member function arguments
{

   //function body; //here int type can't be modify
   but float can be.
}
```

//function declaration with

//function definition with

Static Data Members

If a data item in class is declared as static, then only one copy of that item is created for the entire class, no matter how many objects there are. All the objects share a common item of information. Once the first object is created its value is initialized to zero. And separate definition for static data member is necessary outside the class

General syntax:

```
class class_name
```

```
{
   //......
   static data_type                    //declaration of
variable_name; data                    static
member inside the
class //.....
};
data_type class_name :: //definition variable_name;
of
static data member outside the class and here value
can
```

//be assign to variable.

Static Member Function

A static member function can have access to only other static members declared in the same class and it can be called using the class name

General Syntax:

static return_type func_name (argument list); static member function //.... **//declaration of**

class_name:: fun_name(argument passed) static member function call **//**

Exercises

1. **Write a program that has a class to represent time. The class should have constructors to initialize data members hour, minute and second to 0 and to initialize them to values passed as arguments. The class should have member function to add time objects and return the result as time object. There should be another function to display the result in 24 hour format.**

```
#include <iostream>
using namespace std;
class time {
int h,m,s;
public:
    time()
    {
        h=0;m=0;s=0;
    }
    time(int x,int y, int z)
    {
        h=x;m=y;s=z;
    }
    void get_time()
    {
        cout<<"Enter hour: ";
        cin>>h;
        cout<<"\nEnter minutes: ";
        cin>>m;
        cout<<"\nEnter second: ";
        cin>>s;
    }
    void add_time(time t2)
    {
        h+=t2.h;
        m+=t2.m;
        s+=t2.s;
        h+=m/60+s/3600;
        m+=s/60;
        h%=24;
        m%=60;
        s%=60;
    }
```

```
    void display()
    {
        cout<<endl<<h<<":"<<m<<":"<<s;
    }
};
int main()
{
    time c1,c2(1,2,3);
    c1.get_time();
    c2.get_time();
    c1.add_time(c2);
    c1.display();
    return 0;
}
```

2. Write a program that has a class with a dynamically allocated character array as its data member. One object should contain "Engineers are" and another should contain " Creatures of logic". Member function join() should concatenate two strings by passing two objects as arguments. Display the concatenated string through a member function. Use constructors to allocate and initialize the data member. Also, write a destructor to free the allocated memory for the character array. Make your own function for concatenation of two strings.

```
#include<iostream>
using namespace std;
class character
{
private:
    char *p;
public:
    character()
    {
        p = new char[100];
    }
    void setdata( char a[])
    {
        char *w ;
        w=p;
        while(*a!= '\0')
        {
```

```
            *w=*a;
            w++;
            a++;
        }
        *w='\0';
    }
    void join(character b)
    {
        char *w ;
        w=p;
        while (*w != '\0')
        {
            w++;
        }
         while (*(b.p) != '\0')
         {
             *w=*(b.p);
              w++;
             (b.p)++;
         }
         *w= '\0';
    }
    void display()
    {
            cout<<"the concatenated string is: "<<p;
    }
    ~character()
    {
        delete [] p;
    }

};
int main()
{
    char a[]="engineers are ", b[]="creatures of
logic";
    character j1,j2;
    j1.setdata(a);
    j2.setdata(b);
    j1.join(j2);
    j1.display();
}
```

3. Write a class that can store Department ID and Department Name with constructors to initialize its members. Write destructor member in the same class and display the message "Object *n* goes out of the scope". Your program should be made such that it should show the order of constructor and destructor invocation.

```
#include<iostream>
#include<cstring>
using namespace std;
class department
{
    int id;
    string name;
public:
    department(int i,string n)
    {
        id=i;
        name=n;
        cout<<"Object "<<name<<" is
created.\n"<<endl;
    }
    ~department()
    {
        cout<<"Object "<<name<<" is
destroyed.\n"<<endl;
    }
};
int main()
{
    department d1(1,"BCT"),d2(2,"BEX"),d3(2,"BME");
    return 0;
  }
```

4. Assume that one constructor initializes data member say num_vehicle, hour and rate. There should be 10% discount if num_vehicle exceeds 10. Display the total charge. Use two objects and show bit-by-bit copy of one object to another (make your own copy constructor).

```
#include<iostream>
using namespace std;
class vehicle
{
```

```
   int n,h,r;
   float am;
public:
    /*vehicle(vehicle& d)
    {
        n=d.n;
        h=d.h;
        r=d.r;
    }*/
    void setdata()
    {
     cout<<"enter no. of vehicles, time in hours and
rate:\t";
     cin>>n>>h>>r;
    }
    void calc()
    {
        am= h*r;
        if(n>9)
        {
                am*=0.9;
        }
    }
    void display()
    {
        cout<<"\nThe charge is:"<<am;
    }
};
int main()
{
    vehicle a;
    a.setdata();
    vehicle b(a);
    b.calc();
    b.display();
}
```

5.Write a program that illustrate the following relationship and comment the relationships.

i)const_object.non_const_mem_function
ii) const_object.const_mem_function

iii) non_const_object.non_const_mem_function

iv) non_const_object.const_mem_function

```
#include<iostream>
using namespace std;
class example
{
    int num;
public:
    example ()
    {
        cout<<"Enter a number :\t";
        cin>>num;
    }
    void const_display()const
    {
        cout<<"The number is : "<<num<<"\n\n";
    }
    void display()
    {
        cout<<"The number is : "<<num+4<<"\n\n";
    }
};
int main()
{
    const example e1; /*Constant object*/
    /*Constant object cannot call non_constant member
function*/
    e1.const_display();/*Constant object calling
constant member function*/
    example e2;/*Non constant object*/
    e2.display();/*Non constant object calling
non_constant member function*/
    e2.const_display();/*Non constant object calling
constant member function*/
    return 0;
    }
```

6. Create a class with a data member to hold "serial number" for each object created from the class. That is, the first object created will be numbered 1, the second 2 and so on by using the basic concept of static data members. Use static

```
    member function if it is useful in any of the
    member functions declared in the program.
    Otherwise make separate program that demonstrate
    the use of static member function.
#include <iostream>
using namespace std;
class serial_no
{   int s;
    static int n;
    public:
    int getserial_no()
    {
        cout<<"Enter serial number: ";
        cin>>s;
    }
    static void display1()
    {
        n++;
        cout<<"Number "<<n<<endl;
        cout<<"Serial number"<<endl;
    }
    void display2()
    {
        cout<<s<<endl; /*static function can only
access static values.*/
    }

}create[4]; //Creating 4 objects
int serial_no::n=0;
int main()
{
    int i;
    for(i=0;i<4;i++)
    {
      create[i].getserial_no();
    }
    for(i=0;i<4;i++)
    {
      create[i].display1();
      create[i].display2();
    }
    return 0;
}
```