POINTERS AND ARRAYS

9.1 Void Pointer:

- Pointers can also be declared as void type. When declared void 2 bytes are allocated to it. Later using type casting actual number of bytes can be allocated or deallocated.
  Ex:

```
#include<iostream.h>
#include<conio.h>
int p; float d; char c;
void *pt=&p;
void main()
{
        clrscr();
        *(int *)pt=12;
        cout<<p;
        pt=&d;
        *(float *)pt=5.4;
        cout<<d;
        pt=&c;
        *(char *)pt='s';
        cout<<c;

}
```

9.2 Wild pointer:

- When a pointer points to an unallocated memory location or to data value whose memory is deallocated such a pointer is called as wild pointer.
- This pointer generates garbage memory location and pendent reference.
- A pointer can become a wild pointer due to :
  - ✓ Pointer is declared but not initialized.
  - ✓ Pointer alteration:
    It is the careless assignment of new memory location in a pointer. This happenes when other wild pointer access the location of a legal pointer. This wild pointer than converts a legal pointer to wild pointer.
  - ✓ Access destroyed data:
    This happens when the pointer attempts to access data that has no longer life.

9.3 Class Pointer:

- Class pointer are pointers that contains the starting address of member variables.

Ex:

```
#include<iostream.h>
```

```
#include<conio.h>
void main()
{
    class man
    {
        public:
            char nm[20];
            int age;
    };
    man m= {“Ravi”,15};
    man *ptr;
    ptr= &(man)m;
    clrscr();
    cout<<m.name<<m.age;
    cout<<ptr->name<<ptr->age;
}
```

9.4 Pointers to derived and base classes:

```
#include<iostream.h>
#include<conio.h>
class A
{
    public:
        int b;
        void display()
        {
            cout<<b;
        }
};
class B : public A
{
    public:
        int d;
        void display()
        {
            cout<<b<<d;
        }
};
void main()
{
    clrscr();
    B *cp;
    B b;
    cp=&b;
    cout<<”Enter values for b & d :”;
```

```
        cin>>cp->b>>cp->d;
        cp->display();
}
```

9.5 Array of classes:

- In this array every element is of class type.

Ex:

```
#include<iostream.h>
#include<conio.h>
class student
{
    public:
            char name[30];
            int rolln0;
            char branch[10];
};
class student st[10];
```

9.6 BINDING AND POLYMORPHISM

Binding means link between a function call and the real function that is executed when a function is called.

- There are two types of binding :-
  i) Compile-time or early or static binding
  ii) Run-time or late or dynamic binding
- In early binding, the information pertaining to various overloaded member function is to be given to the compiler while compiling.
- Deciding a function call at compile time is called static binding.
- Deciding a function call at run time is called dynamic binding.
- It permits suspension of the decision of choosing a suitable member function until run time.
- There are two types of polymorphism:
  i) Run-time polymorphism (Virtual Function)
  ii) Compile-time polymorphism (Function Overloading & Operator Overloading)

9.7 Compile time (Virtual Function):

```
class first
{
        int d;
        first() {d=5;}
        public:
                void display()
                {
                        cout<<d;
                }
};
```

```
class second : public first
{
        int k;
        public:
                second() {k=10;}
                void display()
                {
                        cout<<k;
                }
};
void main()
{
        second r;
        r.display();
}
```

- The virtual function of base classes must be redefined in the derive classes.
- The programmer can define a virtual function in a base class and can use the same function name in any derived class even if the number and type of arguments are matching.
- The matching function overwrites the base class function of the same name. This is called as function overriding.
- The base Class Version is available to derived class objects via scope overriding.

Rules For Virtual Functions:

- The virtual functions should not be static and must be member of a class.
- A virtual function may be declared as friend of another class.
- Constructors can't be declared as virtual but destructors can be virtual.
- The virtual function must be defined in the public section of the class.
- The prototype of virtual functions in base and derived classes should be exactly the same.
- In case of mismatch the compiler neglects the virtual function mechanism and treate them as overloaded function.
- If a base class contain virtual function and if the same function is not redefined in the derived classes in that case the base class function is invoked.
- The keyword virtual prevents the compiler to perform early binding. Binding is postponed until run time.
- The operator keyword used for operator overloading also supports virtual mechanism.

```
#include<iostream.h>
#include<conio.h>
class first
{
        int b;
        public:
                first()
```

```
        {
            b=10;
        }
        virtual void display()
        {
            cout<<"b="<<b;
        }
};
class second : public first
{
    int d;
    public:
        second()
        {
            d=20;
        }
        void display()
        {
            cout<<"d="<<d;
        }
};
void main()
{
    clrscr();
    first f, *p;
    second s;
    p=&f;
    p->display();
    p=&s;
    p->display();
}
```

9.8 Pure Virtual Function :

- In practical applications the member functions of base classes are rarely used for doing any operation such functions are called as do-nothing functions or dummy functions or pure virtual functions.
- They are defined with null body. So that the derived classes should be able to over write them.
- After declaration of a pure virtual function in a class the class becomes an abstract class. It can't be used to declare any object.

syntax:

```
virtual void functionname () =0;
```

Ex:

```
virtual void display () =0;
```

- Any attempt to declare an object will result in an error that is can't create an instance of abstract class.

- Here the assignment operator is used just to instruct the compiler that the function is a pure virtual function and it will not have a definition.
- The classes derived from pure abstract classes are required to redeclare the pure virtual function.
- All these derived classes which redefine the pure virtual function are called as concrete classes.
- These classes can be used to declare objects.

Ex:

```
#include<iostream.h>
#include<conio.h>
class first
{
        protected:
                int b;
        public:
                first()
                {
                b=10;
                }
        virtual void display()  = 0;
};
class second : public first
{
        int d;
        public:
                second()
                {
                        d=20;
                }
                void display()
                {
                        cout<<b<<d;
                }
};
void main()
{
        clrscr();
        first *p;
        second s;
        p=&s;
        p->display();
}
```

9.9 Object Slicing:

- A virtual function can be invoke using pointer or reference. If we do so object slicing takes place.

```
#include<iostream.h>
#include<conio.h>
class A
{
	public:
		int a;
	A()
	{
		a=10;
	}
};
class B : public A
{
	public:
		int b;
	B()
	{
		a=40;b=30;
	}
};
void main()
{
	clrscr();
	A x;
	B y;
	cout<<x.a;
	x=y;
	cout<<x.a;
}
```

9.10 VTABLE & VPTR:

- To perform late binding the compiler establishes a virtual table (VTABLE) for every class and its derived classes having virtual function.
- The VTABLE contains addresses of the virtual functions.
- The compiler puts each virtual function address in the VTABLE.
- If no function is redefined in the derived class ,i.e defined as virtual in the base class the compiler take address of the base class function.
- If it is redefined in the derived class, the compiler takes the address of derived class function.
- When objects of base or derived classes are created a void pointer is inserted in the VTABLE called VPTR (virtual pointer).
- This VPTR points to VTABLE.

9.11 new & delete operator:

- The new operator not only creates the object but also allocates memory.
- It allocates correct amount of memory from the heap that is also called as a free store.
- The delete operator not only destroys the object but also releases allocated memory.
- The object created and memory allocated by using new operator should be deleted by delete operator . Otherwise such mismatch operations may corrupt the heap or may crush the system.
- The compiler should have routines to handle such errors.
- The object created by new operator remains in memory until it is released by delete operator.
- Don't apply C functions such as malloc (), realloc () & free () with new and delete operators. These functions are unfit to object oriented techniques.
- Don't destroy the pointer repetitively. The statement delete x does not destroy the pointer x but destroys the object associated with it.
- If object is created but not deleted it occupies unnecessary memory. So it is a good habit to destroy the object & release memory.

Ex:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int *p;
        p=new int[3];
        cout<<"Enter 3 integers :";
        cin>>*p>>*(p+1)>>*(p+2);
        for(int i=0; i<3; i++)
        cout<<*(p+i)<< (unsigned) (p+i);
        delete [] p;
}
```

9.12 Dynamic Object:

- An object that is created at run time is called as a dynamic object.
- The construction & destruction of dynamic object is explicitly done by the programmer using new and delete operators.
- A dynamic object can be created using new operator.

  ptr = new classname;
- Where the variable ptr is a pointer object of the same class the new operator returns the object created and it is stored in the pointer ptr.
- delete object syntax is : delete ptr;

Ex:

```
#include<iostream.h>
#include<conio.h>
class data
{
        int x,y;
        public:
                data()
                {
                        cout<<"Enter the values for x & y :";
                        cin>>x>>y;
                }
                ~data()
                {
                        cout<<"Destructor";
                }
                void display()
                {
                        cout<<x<<y;
                }
};
void main()
{
        clrscr();
        data *d;
        d=new data;
        d->display();
        delete d;
}
```

/* Write a program to define virtual and non-virtual functions and determine the size of objects.*/

```
#include<iostream.h>
#include<conio.h>
class A
{
        private:
                int j;
        public:
                virtual void show()
                {
                        cout<<"In class A";
                }
```

```
};
class B
{
        private:
                int j;
        public:
                void show()
                {
                cout<<"In class B";
                }
};
class C
{
        public:
                void show()
                {
                        cout<<"In class C";
                }
};
void main()
{
        clrscr();
        A x;
        B y;
        C z;
        cout<<sizeof(x);
        cout<<sizeof(y);
        cout<<sizeof(z);
}
```

9.13 Heap:

- It is a huge section of memory in which large number of memory locations is placed sequentially.
- It is used to allocate memory during program execution or runtime.
- Local variables are stored in the stack and code in code space.
- Local variables are destroyed when a function returns.
- Global variables are stored in the data area. They are accessible by all functions.
- The heap is not declared until program execution completes. It is a user's task to free the memory.
- Memory allocated from heap remains available until the user explicitly deallocates it.

9.14 Virtual Destructors:

- The constructor can't be virtual since it requires information about the accurate type of object in order to construct it properly, but destructor can be declared as virtual and implemented like virtual functions.
- A derived class object is constructed using new operator.
- The base class pointer object force the address of the derived object.
- When this base class pointer is destroyed using delete operator the destructor of base and derived classes is executed.

Ex:

```
#include<iostream.h>
#include<conio.h>
class B
{
        public:
                B()
                {
                        cout<<"Class B constructor";
                }
                virtual ~B()
                {
                        cout<<"In class B destructor";
                }
};
class D : public B
{
        public:
                D()
                {
                        cout<<"Class D constructor ";
                }
                ~D()
                {
                        cout<<"In class D destructor";
                }
};
void main()
{
        clrscr();
        B *p;
        p=new D;
        delete p;
        getch();
}
```

Advantage:

- Virtual destructors are useful when a derived class object is pointed by the base class pointer object in order to invoke the base class destructor.

EXCEPTION HANDLING

10.1 Multiple catch statement

```
 #include<iostream.h>
 #include<conio.h>
 void num(int k)
 {
 try
 {
 if(k==0)throw k;
 else
 if(k>0)throw 'p';
 else
 if(k<0)throw .0;
 }
 catch(char g)
 {
        cout<<"\n caught a positive value";
 }
 catch(int j)
 {
       cout<<"\n caught a null value";
 }
 catch(double f)
 {
         cout<<"\n caught a negative value";
 }
}
int main()
{
 num(0);
 num(-5);
 num(-1);
 getch();
}
```

10.2 Catching Multiple Exception\ Generic catch block

```
#include<iostream.h>
#include<conio.h>
void num(int k)
{
 try
 {
 if(k==0)throw k;
```

```
    else
    if(k>0)throw 'p';
    else
    if(k<0)throw .0;
    cout<<"\n ---------TRY BLOCK-----------";
    }
    catch(...)
    {
            cout<<"\n caught an exception";
    }

}
int main()
{
   num(0);
   num(5);
   num(-1);
   getch();
}
```

10.3 Rethrowing an Exception

```
#include<iostream.h>
#include<conio.h>
void sub(int j,int k)
{
    try
    {
          if(j==0)
          throw j;
          else
          cout<<"\n subtraction:"<<j-k;
    }
    catch(int)
    {
           cout<<"caught a null value";
           throw;
    }
}
int main()
{
   try
   {
       sub(8,5);
       sub(0,8);
   }
   catch(int)
```

```
    {
            cout<<"caught a null value inside main";
    }
    getch();
}
```

10.4 Specifying Exception

```
#include<iostream.h>
#include<conio.h>
class num
{
     public:
          float a;
          void get()
          {
              cout<<"enter a number:";
              cin>>a;
          }
          friend void result(num &,num &);
};
void result(num &n,num &m)
{
     float r;
     r=n.a/m.a;
     cout<<"division is"<<r;
}
int main()
{
    num o1,o2;
    o1.get();
    o2.get();
    try
    {
          if(o2.a==0)
          throw (o2.a);
          else
          result(o1,o2);
    }
    catch(float k)
    {
             cout<<"exception caught:"<<k;
     }
     getch();
}
```