OBJECT ORIENTED PROGRAMMING WITH C++

SUB CODE :18BIT23C

UNIT IV: Pointers: Declaration – Pointer to Class – Object – THIS Pointer – Pointer to Derived Classes and Base Classes – Arrays: Characteristics – Arrays of Classes – Memory Models – New and Delete Operators – Dynamic Object– Binding – Polymorphisms and Virtual Functions.

TEXT BOOK

1. Ashok N Kamthane, “Object Oriented Programming with ANSI and Turbo C++”, Pearson Education Publications, 2006

Prepared By: Dr. M.Soranamageswari

Pointer and arrays:

- C++ variables are used to hold data during program execution.
- Every variable can occupy some memory location to hold the data.
- Memory is arranged in a sequence of byte.
- The number specification to each byte is called memory address.
- The sequence starts from 0 onwords.
- It is possible to access and display the address of memory location using the pointer variable.

Pointer:

- Pointer is a memory variable that store a memory address.
- Pointer can have any name that is legal to other variable.
- It can be declared like normal variable it is always denoted by star(*)operator.

Feature of pointer:

- Pointer save memory space.
- Execution time with pointer is faster.
- The memory is accessed efficiently with the pointer.
- Pointer are used with data structure and they are useful for representing two dimensional array.
- A pointer declared to a base class can access the object of derived class.

Pointer Declaration:

- Pointer can be declared as follows:
- Syntax:

  Data type * pointer variable name.

Example:

- int *a;
- int *b;
- float*c;

- It inform to the compiler that hold the address of any variable.
- The in-direction(*)operator is also called de-reference operator.
- Where as a pointer indirectly access to their own values.
- The (&) is the address operator represented the address of a variable.
- The address of any variable is a whole number.

Example:

- Display the value and address of the variable using pointer.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int *p;
int x=10;
p=&x;
cout<<x<<&x;
cout<<*p<<&p;
}
```

Void pointer

- Pointer can be also be declared as void type.
- Void pointer cannot be reference without explicit type conversion.
- A void pointer can pointer any type of variable with proper type casting.

Example:

```
#include<iostream.h>
#include<conio.h>
int p;
float d;
char c;
void *pt=&p;
```

```
void main()
 {
*(int*)pt=12;
cout<<p;
pt=&d;
*(float*)pt=5.4;
cout<<d;
pt=&c;
*(char*)pt=’x’;
cout<<c;
}
```

Pointer to class:

- The pointer is a variable that hold the address of another data variable.
- The variable may be of int,float and double in the same way we can define pointer to class.
- This type of pointer are called class pointers.

Syntax:

Class name *pointer variable name

Example:

Student *str;

Example:

```
#include<iostream.h>
#include<conio.h>
Classman
{
public:
char name[20];
int age;
voidgetdata(char s[10],int a)
{
name =s;
age=a;
}
```

```
Voidputdata()
{
cout<<name<<age;
}
};

void main()
{
Man *ptr,m;
ptr=&m;
ptr->getdata(“xyz”,m);
ptr->putdata();
}
```

Pointer to object:

- Like variable object can also have an address.
- The pointer can point to a specified object.
- Using the pointer it is possible to access different type of classes.

Syntax:

Class name*ptr variable name;

Eg:

Student*str;

Example

```
#include<iostream.h>
#include<conio.h>
class bill
{
intqty;
float price;
float amount;
public:
{
Voidgetdata(inta,floatb,float c)
{
qty=a;
price=b;
amount=c;
}
```

```
void show()
{
cout<<”QUANTITY:”<<qty<<”\n”;
cout<<”PRICE”<<price<<”\n”;
cout<<”AMOUNT”<<amount<<”\n”;
}
};
int main()
{
clrscr();
bill s;
bill*ptr;
ptr=&s;
ptr->getdata(45,10.25,45*10.25);
(*ptr).show();
return 0;
}
```

This pointer:

- The objects are used to invoke non static member function of the class.
- The pointer this is transferred as an unseen parameters to call non static member function.
- The keyword “this” is a local variable always present in the body of non-static member function.
- The keyword this does not need to be declared.
- The pointer is rarely referred explicitly in a member function.
- It is used implicitly with in the function for member reference.
- Using this pointer it is possible to access the individual member variables of the object.

Example:

- Program to use this pointer and return pointer reference(minimum or smallest value of two numbers)

```
#include<iostream.h>
#include<conio.h>
Class number
{
Public:
intnum;
void input()
{
cin>>num;
}
```

```
void show()
{
cout<<num;
}
number min(number t)
{
if(t.num<num)
return t;
else
return *this;
}
};
```

```
void main()
{
number n,n1,n2;
n1.input();
n2.input();
n=n1.min(n2);
n.show;
}
```

Pointer to derived classes and base class :-

- It is possible to declare a pointer , which points to the base class as well as the derived class.
- One pointer can point different classes.

Example

- Write a program to declare a pointer to the base class and access the member variable of base and derived class.

```
# include<iostream.h>
#include<conio.h>
class A
{
```

```
Public:
int b;
void display()
{
cout<<”b=”<<b<<”\n”;
}
};
class B:public A
{
public:
int d;
void display()
{
cout<<”b=”<<”\n’<<”d=”<<d<<”\n”;
}
};
```

```
void main()
{
clrscr();
A*CP;
A base;
CP=&base;
CP->b=100;
// CP->d=200;Not accessible
cout<<"\n CP points to the base object\n"
 CP->display();
B b;
Cout<<"\n CP points to the derived class\n"
 CP=&b;
CP->b=150;
// CP->d=300; Not accessible
CP->display();
return 0;
}
```

Pointer to the derived class

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
int b;
void display()
{
cout<<”b=”<<b<<”\n”;
}
};
```

```
class B : public A
{
public:
int d;
void display()
{
cout<<”\t b=”<<”\n”<<”\t d=”<<d<<”\n”;
}
};
```

```
void main()
{
clrscr();
B*CP;
B b;
CP=&b;
CP->b=100;
CP->d=350;
cout<<"\n CP points to the derived object \n";
CP->display();
return 0;
}
```

Array

- Array is the collection of elements of similar data types in which each element in unique and located in separate memory locations.
- **Array declaration and initialization:-**
- The arrays can be declared for different data type are described as follows;
- int a[10];
- char b[20];
- double c[5];
- The array initialization can be declared as follows:-

Example

- int a[5]={1,2,3,4,5};
- The range of elements begins from zero(0) memory location.

Characteristics of arrays :-

- All the elements of an array of the same name and they are differentiated from one another with the help of elements number.
- The element number in an array is the most important factor in calling each element.
- Any particular element of an array can be modified without disturbing the other element.
- Any element of an array can be assigned to another ordinary variable (or) array variable of its type.
- The array elements are stored in continuous memory locations, the amount of storage required for the elements depends on its type and size
- Total bytes=sizeof(data type)*sizeof(array)

Example

```
#include<iostream.h>
#include<conio.h>
class arraytest
{
int a[10];
public:
void get();
void display();
};
```

```
void arraytest::get()
{
for(int i=1;i<=10;i++)
{
cin>>a[i];
}
}
void arraytest::display()
{
for(int i=1;i<=10;i++)
{
cout<<a[i];
}
}
```

```
void main()
{
arraytest a1;
a1.get();
a1.display();
}
```

Array of classes :-

- An array is a collection of similar data type in the same way it is also possible to define array of classes.
- Here array is a class type array of class object can be declared as follows:

```
class student
{
char sname[20];
student s[10];
int sno[10];
char sadr[20];
}
```

Example

```
#include<iostream.h>
#include<conio.h>
Class student
{
Char sname[20];
Char grade;
int sno;
Public:
void get();
void display()
};
```

```
void student::get()
{
cin>>sname;
cin>>sno;
cin>>grade;
}
void student::display()
{
cout<<sname;
cout<<sno;
cout<<grade;
}
```

```
void main()
    {
    student S[10];
    int n,i;
    cout<<”enter the no. of students “;
    cin>>n;

for(i=1;i<=n;i++)
    {
    S[i].get();
    }
    for(i=1;i<=n;i++)
    {
    S[i] display();
    }
    }
```

C++ and Memory models :-

- The memory models sets the supportable size of code and data.
- Before computing and linking the source code we have to specify the appropriate memory model.
- C++ always use different segments for code and data.

The different types of memory models used in C++ are as follows :

1. Tiny :-

- The tiny model is absolutely premium all four segment registers(CS,DS,SS,ES) are initialized with same address and all addressing is accomplished using 16-bits total memory capacity is 64 kb.
- The code data and stack must fit with in the same 64kb segment
- Programme are executed quickly is this model is selected.

2. Small :-

- All code should fit in a single 64kb segment.
- All pointers are 16bit in length execution speed is same as single model.
- This model is used for average size programs near pointers are always used.

3. Medium :-

- All data fit in a single 64kb segment.
- The code is allowed to use multiple segments.
- All jumps and calls require 32bits address.
- In this model access to data is passed program execution is slowwith their model.
- Far pointers are used for code.

4. Compact :-

- All code fit in 64kb segment to the data can use multiple segment all pointers to data and 32bits.
- But jumps and calls used 16bits slow action to data and quick code execution will be observed this model.
- For pointers are preferred for data.

5. Large :-

- Both code and data allowed to multiple segment.
- All pointers are 32bits code execute is slow.
- This model is preferred for very big programs.
- For pointers are used for both code and data.

6. Huge :-

- Both code and data are allowed to use multiple segments
- Every pointers is 32bits in length cod execution is the slowest.
- The huge model permits multiple data segments 1MB code segments and 64kb for stack.

The pointers and code pointers are far

S.no	**Memory model**	**Secments**			**Type of pointers**	
		Code	**Data**	**Stack**	**Code**	**Data**
1	Tiny	64kb	64kb		Near	Near
2	Small	64kb	64kb		Near	Near
3	medium	1mb	64kb		Far	Near
4	Compact	64kb	1mb		Near	Far

5	Large	1mb	1mb		Far	Far
6	Huge	1mb	64kb each	64kb	Far	Far

7. **Segment and offset address**

- Every address has to parts segment and offset these parts are defined using two macrosdefind in <dos.h> header file.
- FP-SEG()-this macros is used to obtained segment address.
- FP-OFF() – this macros is used to obtained offset address.

The new and delete operators

- C++ provide new and delete operators can be used for crating and destroying the object.

New operator :-

- The new operator allocates memory of specified type and returns the starting address to the pointers memory variable.
- If it’s fails to allocate its returns null value.

Syntax

Pointer memory variable=new data type

Example

a= new int;

int *a = new int[50];

*a = new int[3];

Delete operator :-

- The delete operator frees the memory location allocated the operators.

Syntax

Delete pointer variable name;

Delete [element size] pointer variable name;

Example

Delete a;

Delete[50] a;

Size of operator

- The sizeof operator is used to returns the size of your variable bytes.

Syntax

- sizeof (variable name);

Example

- Cout<<sizeof(a);

Few points regarding : new and delete operators are :-

- The new operator not only creates an object but also allocates memory.
- The new operator allocates correct amount of memory from the heap that also called a free store.
- The object created and memory allocated using new operator should be deleted by the delete operator.
- The delete operator not only destroys the object but also returns the allocated memory.
- The new operator creates an object and its remains in the memory until it is released using delete operator.
- If we sent a null pointers to delete operator it is secure using delete to zero(0) has no result.

- The statement delete x does not destroy the pointer x.
- It destroy the object.
- If the object created is not deleted it occupy the unnecessarily.
- It is a good habit to destroy the object and release the system resources.

Difference between new and malloc()

New	Malloc()
• Creates objects • returns pointers of relevant type. • It is possible to overload new operators.	• Allocates memory • returns void pointer. • Malloc() cannot be overloaded.

Write a program to allocate memory to store 3 integers. Use new and delete operator.

```
#include<iostream.h>
#include<conio.h>
int main()
{
Clrscr();
inti,*p;
p=&i;
```

```
p=new int[3];
*p=2;
*p(p+1)=3;
*p(p+2)=4;
 cout<<”value address”;
for(int x=0;x<3;x++)
cout<<endl<<*(p+x)<<”\t”<<(unsigned)(p+x);
delete[]p;
return 0;
}
```

Dynamic object:-

- C++ supports dynamic memory allocation .C++ allocates memory and initializes the member variable.
- An object can be created at run time.
- such object is called dynamic.
- The construction and destruction are dynamic object is explicitly done by the programmer.
- The operator new and delete are used to allocate and deallocate to such objects.
- A dynamic objects can be create using new operator.

Syntax:

- Ptr=new class name;

- The new operator returns the address of the object created and it is stored in the pointer(ptr).
- Ptr is a pointer variable is the same class.
- The member variables of object can be accessed using operator
- The dynamic object can be destroy using delete operator.

Syntax:

- delete ptr;
- It destroy the object pointed by pointer ptr.

Example

```
#include<iostream.h>
#include<conio.h>
class data
{
int x,y;
public:
data()
{
cout<<"\n construction";
x=10;
y=50;
}
~data(){cout<<"\n destructor";}
```

```
void display()
{
cout<<”\n*x=”<<x;
cout<<”\n y=”<<y;
}
};
void main()
{
clrscr();
data*d;
d=new data;
d→display();
delete d;
}
```

Binding ,polymorphism and virtual functions

- **Binding:-**
- In c++ functions can be bound either at complied time (or) at run time designing the function call at compile time is called compile time (or)early(or)static binding.
- Designing function call at run time is called as run time(or)late(or)dynamic binding .
- Dynamic binding permits the decision of choosing suitable function until run time.

Static or early binding:-

- Similar function name are used at many places but during their references their position is indicates explicitly .
- Their ambiguities are fixed at compile time.

Example:

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

```
int k;
public:
void display()
{
cout<<k;
}
}
```

2. Dynamic (or) late binding :

- Dynamic binding of member function in c++ can be done using the keyword VIRTUAL
- The member function followed by virtual keyword is called as virtual function.
- The virtual function must be defined in public section.
- If the function is declared virtual,the system will use dynamic binding
- Which is carried out at runtime

Example:

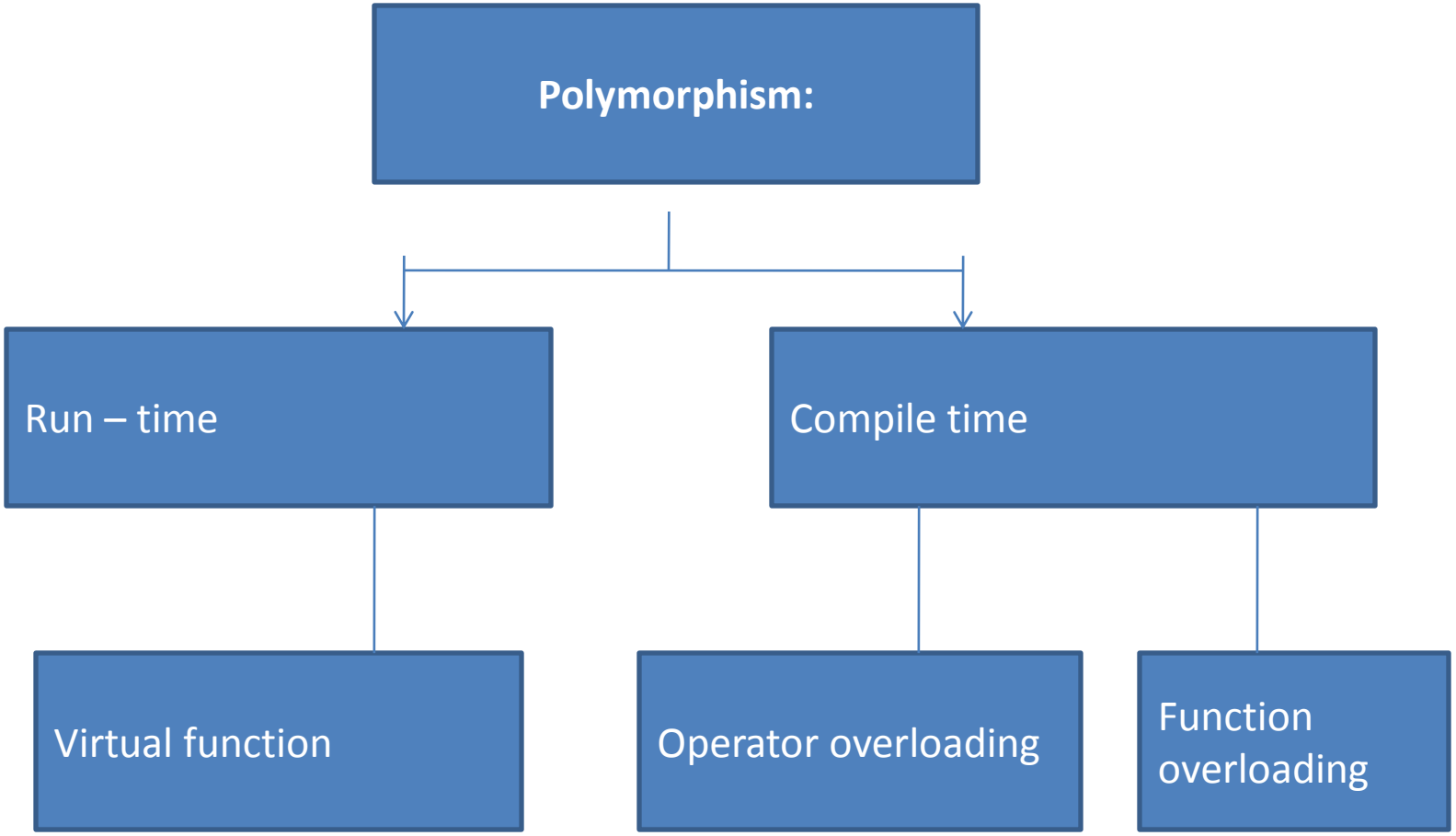
```
#include<iostream.h>
#include<conio.h>
class data
{
int x,y;
public:
data()
{
cout<<"\n constructor";
x=10;
y=50;
}
```

```
~data()
{
cout<<”\n destructor”;
}
void display()
{
cout<<”\n*X=”<<x;
cout<<”\n Y=”<<y;
}
};
```

```
void main()
{
clrscr();
data*d;
d=new data;
d→display();
delete d;
}
```

Polymorphism:

- It is a technique in which various forms of single function can be defined and shared by various object to perform the operation polymorphism


Polymorphism:
Run – time
Compile time
Virtual function
Operator overloading
Function overloading

Virtual Function

- Run time polymorphism can be achieved using virtual functions
- Virtual functions of the base class must be re-defined in the derived class
-
- A virtual function in a base class can use the same function name in any derived class even if the number and type of argument of same Virtual functions can be number functions

Rules For Virtual Function:

- The virtual function should not be static and must be a number of class
- A virtual function may be declared as friend for another class
- Constructors cannot declare as virtual but destructors can be declared as virtual
- The virtual function must be defined in public section of the class
- The virtual keyword is used in the declaration and not in the member declaration
- It is also possible to return a value from virtual function
- The prototype of virtual function in base and derived should be the same
- Arithmetic operation cannot be used with base class pointer
- If a base class contains Virtual function and if the same function is not redefined then the base class function is invoked
- The operator keyword is used for operator overloading also supports virtual mechanism

Example:

```
#include<iostream.h>
#include<conio.h>
class first
{
int b;
public:
first() { b=10; }
virtual void display()
{
  cout<<b;
}
};
class second: public first
{
int d;
public:
second() { d=20; }
```

```
virtual void display()
{
cout<<d;
}
};
void main()
{
first f,*p;
second s;
p=&f;
p→display();
p=&s;
p→display();
}
```

Array of pointers

- Polymorphism refer to late (or) dynamic binding.
- Dynamic binding is associated with object pointers.
- Address of different object can be stored in an array to invoke the function dynamically.

Example

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
```

```
virtual void show()
{
cout<<”A”;
}
};
class B : public A
{
public:
void show()
{
cout<<”B”;
}
};
```

```
class C : public A
{
public:
void show()
{
cout<<”C”;
}
};
class D : public A
{
```

```
public:
void show()
{
cout<<”D”;
}
};
class E : public A
{
public:
void show()
{
cout<<”E”;
}
};
```

```
void main()
{
A a;
B b;
C c;
D d;
E e;
A*P[]- { &a,&b,&c,&d,&e};
for(int i=0;i<5;i++)
{
P[i] ->show();
}
}
```

Pure virtual function :-

- The member function of the class is rarely used for doing any operation.
- Such function are called do-nothing functions, dummy function or pure virtual function.
- The pure virtual function are always virtual function.
- There are defined with null body after declaration of pure virtual function in a class. The becomes abstract class.
- It cannot be used for declare any object.

The pure function can be declared as follows:

Syntax

- virtual void display()=0;
- A pure virtual function declared in base class cannot be used for any operations.
- The class containing pure virtual function cannot be used declared object.
- Such class are known as abstract class (or) pure abstract class.
- The class is derived from pure abstract classes are required to redeclared pure virtual function.
- All other derived classes witout pure virtual function are called concreate classes.
- These classes can be used to create objects.

```
#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<<”\nb=”<<b<<”d=”<<\n”
}
```

```
int main()
{
clrscr();
first*P;
second *S;
p=&S;
p->display();
return 0;
 }
```

Constructor and virtual function

- It is possible to define a virtual function using constructor.
- Constructor makes the virtual mechanism illegal.
- When a virtual function is invoked throw a constructor the base class virtual function will not be called and instead of this member function of the class is invoked.

Example

```
#include<iostream.h>
#include<conio.h>
class B
{
```

```
int k;
public:
B(int l)
{
K=1;
}
virtual void show()
{
cout<<k;
}
};
class D : public B
{
int h;
```

```
public :
D(int m,int n):B(m)
{
h=n;
B=*b;//base class ptr in derived class constructor
B this;//it contains the address of the object calling the member fun.
b->show();
}
```

```
void show()
{
cout<<h;
}
};
void main()
{
B b(4);
D d(5,2);
}
```

Virtual destructors

- Destructors can also declared as virtual.
- Constructors cannot be virtual since it requires about the accurate type of object.
- Destructors of derived and base class are called when a derived class object is addressed by the base class pointer is deleted.
- When the base class pointer is destroy using delete operation the destructors of base and derived class is executed.

Example

```
#include<iostream.h>
#include<conio.h>
class B
{
public:
B();
{
cout<<”constructor B”;
}
```

```
virtual ~B()
}
cout<<”destructor B”;
};
class D:public B
{
public:
D();
{
cout<<”constructor D”;
}
 ~D()
{
cout<<”destructor D”;
};
```

```
void main()
{
B*P;
P=new D;
delete P;
}
```