INDEX

NAME: ... Reddy STD.: BTech SEC.: 17 ROLL NO.: 12B1T0077 SUB.: OOPS (C++)

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		25th Feb -		
		no class templates		

```
class student
{
    char name[3];
    int Rollno;
    int mark;

    void getinfo();
    void display();

};

Void student::getinfo()
{
        cin>>name;
        cin >> Rollno;
        cin >> mark;
}
void Student :: display()
{
        cout << name;
        cout << Rollno;
        cout << mark;
}
int main
{
        student S;
        S.getinfo();
        S.display();
}
```

scope resolution operator

datatype declaration

prototype declaration

function invocation

String array:

```
class student.
{
    char name[3];
    int rollno;
    int mark;
public:
    void getinfo();
public    void display(); };
void student::getinfo()
{   cin>>name>>rollno>>marks;
}
void stud::display()
{ cout<<name<<rollno<<marks; }

int main()
{
  int i;
  student s[5];
{for(i=0; i<5; i++)
{ s[i].getintfo(); }
 for(i=0; i<5; i++)
  { s[i].display(); }
  getch();
  return 0;
  }
```

* private
* public
* protected → inheritance

* Atleast one member function should be under public:

returntype classname :: functionname ();

```
class student
{
    private:
        char name;
        int rollno;
        int mark;
public:
        void display();
};
Student :: getinfo()
{ cin>>name >> marks >> Rollno
}
Student :: display()
{ getinfo();
  cout << " " << " ";
}
```

```
int main()
{
    student s1;
    s1.display();
}
```

* function defined in the class is called "Inline" function

```
class student
{
  private:
    int rollno;
    int mark;
    inline void getdata()
    { cin >> rollno >> mark; }
  public
    void display();
};
void student::display()
{
    getdata();
  cout << rollno << mark; }
int main()
{
  student s;
 s.display();
 getch();
 return 0;
}
```

macro substitution

should not have 'complex code'

are carried over using "stack" data structure. * no memory references in 'inline function'

→ Inline function
→ Function overloading
→ Default arguments
→ friend function.

program counter - PC

→ "I.F"

```
class student
{
    int Regno;
    int marks;
Public :
inline void getdata(){ cin >> Rollno >>mark ; }
    void displaydata();
}
```

* Function Overloading :-

```
char swap (char c, char D)
{
}
int swap( int A, int B)
{
return;
}
double swap( )
{
}

int main()
{
swap (int A, int B)
swap ( char c, char D)
swap (double t, double f)
}
```

"namemangling"
→ the process of modifying of compiler in identifying the actual function

Q WAP to collect the employee info.

- name
- emp.id
- Dept
- salary
- joindate
- DOB

} highest salary

Define the function outside the class

```
class employee
{
    char name[50];
    int empid;
    char Dept[50];
    int salary;
    int joindate;
    char DOB[10];
    int highsalary;
public:
    employee()
    {
        highsalary = 0;
    }
    void getdata();
    void display();
    void highsalary(int a);
};

void employee::getdata()
{
    cin >> name >> empid >> Dept >> salary >> joindate >>
        DOB
}
void employee::display()
{
    cout <<
}
void employee::highsalary(int a)
{
    if (a > highsalary)
        highsalary = a;
}
```

```
int main()
{
    employee E[10];
    for (int i=0 ; i<10 ; i++)
        E[i].getdata();
    for(i=0 ; i<10 ; i++)
        E[i].display();
    salary = E[0].highest_salary();
    for (i=0; i<10 ; i++)
    {
                Employee s;
        if (E[i].highest_salary > salary)
                s.highsalary(E[i].salary);
        salary = E[i].highest_salary();
    }
    cout << "Highest salary" << salary;
}
```

complex addition:

```
class complex
{
    int real;
    float imag;
public:
    void getdata();
    complex addition(complex);
    void display();
};

complex complex::addition(complex c2)
{
    complex temp;
    temp.real = real + c2.real;
    temp.img = img + c2.img;
    return temp;
}

int main()
{
    complex c1, c2, c3;
    c1.getdata();
    c2.getdata();
    c3 = c1.addition(c2);
    c3.display();
```

Friend Function:-

* can be declared either in private (or) public
* Friend fn is not a member of the class

```
class Complex
{ int real;
  float img;
void getdata ();
void display ();
friend void addition (Complex c1, Complex c2)
};
void addition (comp c1, comp c2)
{   real = c1.real + c2.real;
    img = c1.img + c2.img;
}
void complex :: getdata () { cin >> real >> img }
void Complex :: display () { cout << real << img }
int main ()
{
```

```
complex addition (comp c1, comp c2)

};
complex addition (comp c1, comp c2)
{
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.img = c1.img + c2.img;
    return temp;
}

void . —
void . —

int main()
{
    complex c1, c2, temp;

    c1.getdata();
    c2.getdata();

temp = addition (c1, c2);
}
```

Extern → global variable, can be accessed anywhere in the program

Static

Register → 6 registers,

Auto → stack

- A - accumulator
- B
- C
- D
- E

frequently used variables

- F register - flag

* Accessing speed is ~~low~~ high in Registers

STATIC } global
Heap } datastructure

↳ initialises a variable only once.

Constructor is a function

→ Default constructor
→ Parameterised "
→ copy "

new, delete } operators used for memory allocation

* constructor should be declared in the public because of accessing problems

* constructor is used for initialisation

```
class complex
{    int real;
     int img;
public:
   complex (){real=0;img=0}
   complex (int R, int I)
   {real=R ;img=I;}
   void getdata (); void display();
   void addition(c1,c2)
}
int main()
{   complex c1,&c2;
    complex c3(2,3);
     c1.getdata(); c2.getdata();
       c3.getdata;
       c1.addition(c3,c2)
```

Telnet 192.168.0.63

$ vi ex1.cpp

esc

i

esc :wq

$ g++ ex1.cpp

$ ./a.out

→

Ssh (secure software header)

ss claim

cp

move mv ex1.cpp ; ex2.cpp

```
class complex
{
    int real, img;
public:
    void getdata();
    void display();
    complex() { real = ; img = }
    complex(int R, int I) { real = R, img = I }
};
void complex :: getdata()
{ cin >> real >> img;
}
void complex :: display()
{   cout << real << img;
}
int r, i;
complex c3(2,5)
cin >> r >> i;
complex c4(r,i)
```

```
int main()
{
    complex c1, c2;
    c1.getdata();
    c2.getdata();
    complex *ptrobject;
    ptrobject = &c1;
    ptrobject -> getdata;
```

→ Properties related to friend function:

1. It is not in the scope of the class to which it has been declared as the friend.
2. Since, it is not in the scope of the class it cannot be called using the object of the class
3. It can be invoked like a normal function without the help of any object
4. unlike member functions it cannot access the members names directly and has to use an object name and dot membership

operator with each name.

5 It can be declared either in public or in private

6 usually has the object as the arguments

→ Properties of Constructor

* They should be declared in public section. They are invoked automatically when the objects are created. They do not have return types not even void and therefore they cannot return values

* They cannot be inherited through a derived class can call the base class constructor

* like other C++ functions they have default arguements

* Constructors cannot be virtual they cannot refer to their addresses and object with a constructor cannot be used as a member of union.

* They make implicit calls to the operator functions "new" and "delete"

* Destructors cannot be overloaded.

Operators

15

Function overloading:

```
#include <iostream>
using namespace std;
void swap (char c, char d)
{
    char t;
    t = c;
    c = d;
    d = t;
  cout << c << " " << d;
}
void swap (int c, int d)
{
    c = c+d;
    d = c-d;
    c = c-d;
  cout << c << " " << d;
}
void swap (float c, float d)
{
    c = c+d;
    d = c-d;
    c = c-d;
  cout << c << " " << d;
}
main()
{
  int a, b;
  char c, d;
  float e, f;
```

```
cout << "enter the integers";
cin >> a >> b;
swap(a,b);
cout << "\n enter the chars:";
cin >> c >> d;
swap(c,d);
 cout << "\n enter the floating point decimals:";
 cin >> e >> f;
  swap(e,f);
  }
```

Friend function:

```
#include <iostream>
#include <conio.h>
using namespace std;
class complex
{ float real, img;
public:
  void getdata();
  void display();
friend void addition(complex c1, complex c2,
        complex c3);
};
```

```
void addition (Complex c1, Complex c2, Complex c3)
{
    c3.real = c1.real + c2.real;
    c3.img  = c2.img + c2.img;
    c3.display();
}
void complex::getdata()
{
    cin >>real >>img;
}
void complex::display()
{
    cout <<real<<img;
}
int main()
{
    complex c1, c2, c3;
    c1.getdata();
    c2.getdata();
    addition (c1, c2, c3);
  getch();
}
```

Employee using static variable:

```
#include <iostream>
using namespace std;
class emp
{
    static float max;
    char name[50];
    int empid;
    char dept[20];
    float sal;
public:
    void getinfo(void);
    void compare(void);
{   if(sal>max)
    max = sal;
}
    void display(void);
};
void emp::getinfo(void)
{ cin >> max >> name >> empid >> dpt >> sal;
}
void emp::display(void)
{
    cout << max << name << empid << dpt << sal;
}
```

```
int main()
{
    int n,i;
    emp e1,e2;
    e1.getinfo();
    e2.getinfo();
    e1.compare();
    e2.compare();
    e1.display();
    e2.display();
getch();
}
```

Complex no. addition::

```
#include <iostream>
using namespace std;
class complex
{   int real;
    int img;
public:
    void getdata(void);
    void addition(complex, complex);
    void display(void);
};
void complex::getdata(void)
{   cin>>real>>img;
}
```

```
void Complex :: addition (Complex c1, Complex c2)
{
    real = c1.real + c2.real;
    img = c1.img + c2.img;
}
void complex :: display(void)
{
    cout << real << img;
}
int main()
{
    Complex c1, c2, c3;
    c1.getdata();
    c2.getdata();
    c3.addition (c1, c2);
    c3.display();
}
```

Complex c4;

Constructors & Destructors

```
#include <iostream>
#include <conio.h>
using namespace std;
class Complex
{   int real;
    int img;
public:
    Complex()      //default constructor
{ real = 0;
   img = 3;
}
Complex(int R, int I)      //parameterised
{  real = R; img = I;
}
~Complex() { cout << " destructor "; } };
main()
{  Complex c1, c2;
   int a, b;
    cin >> a >> b;
   Complex c3(a, b);          /* parameterised function call
   Complex c4(5, 6);          /*  "   "
    getch();
}
```

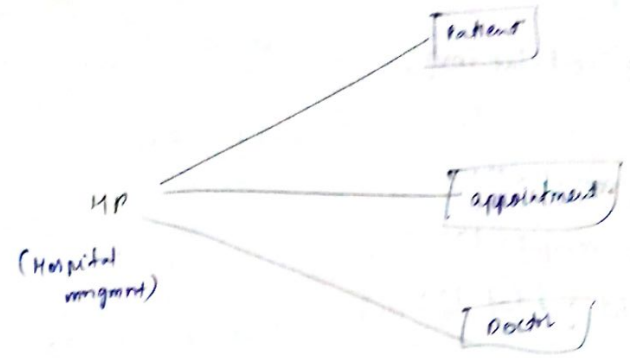
```
class staticexample
{
    int i;
    static int count;
public:
    static objectcount () { count ++ }
    void getdata () { cin >> i; count = count + 1;

    void display () {

};
int staticexample :: count = 0;
int main ()
{
    Staticexample s1, s2;
    staticexample :: objectcount ();
    s1.getdata ();
    s2.getdata ();
    s2.display ()
}
```

for accessing global variables & heap data structure

static → either in public or private

→ Heap

→



→ class

{ find fn;
getdata(); putdata(); find fn; initialize fns;
default arguments;
constructor;
}

accessing
val variables
& heap
data structure

* Dynamic allocation — before / during execution.

Default Arguements

Complex *objptr = new Complex[5]; ← returns starting address

= NULL; ⇒ no memory allocated (check for mem. allocation)

delete name;
↓
variable

```
class complex
{
    int real; int img;
public:
    complex();
    ~complex();
    void getdata();
    void display();
};
void complex::getdata()
{
    cin>>
}
complex::complex()
{
    cout<<"default constructor invoked";
}
```

complex::~complex()
{ cout<<"destructor invoked";
}

```
int main()
{
    complex c1, c2, c3(4,5);
    c1 = complex();
    c2 = complex(2,3);
    ~complex();
```

Default Arguements

```
class complex
{
    int real, img;
public:
    complex();
    void addition (10, 15, 2);
    ~complex()
    void getData();
    void display();
};
```

default arguements
↓
parameters are compulsory

```
int main()
{
    complex c1, c2, c3(4,5)
    c1 = complex();
    c2 = complex(2,3)
    ~complex();
    int a, b, c;
    cin >> a >> b >> c;
    addition (a,b);
    addition (a). !5, 2);
    addition (a,b,c);
```

in (a, b, c) → 'a' cannot be omitted
+
(b, c) can be omitted.

addition (, b, c) → error

Q Define a class string that would work as a user defined string type. include constructors, that will enable us to create an uninitialised string, and also initialise an object with a string constant at the time of creation. copy the same string to one more variable

Ans:

```
char *name;
name = new char[l+1];
```

↑ for null character also memory to be calculated.

Copy Constructor:

→ deep copy → using copy constructor → pts data to be copied → complex c1, c2; deep copy

Complex c3 = c2;

→ shallow copy → c1 = c2

copy const is automatically defined by compiler if it is not defined

Copy Constructor:

```
complex :: complex ( const & c2)
{
    char * name
    name  = new char [c2.size]
    for (          )
}
```

Complex c1 = c2
copy constructor

c1 = c2
↓
operator = overloading

```
class string
{
    int size;
    char * name;
public:
    string ();
    String ( const string &);
    String ( char * getname);
```

Copy Constructor :-

deep - byte byte ← dynamic allocation

shallow - assignment

```
class string
{
    char *name;
    int mark;
    int size;
public:
    string ();
    string (char *str, int mark, int size);
    string (string &s2);
};
string :: string (char *str, int mark, int size)
{
    name = new char[30];
    strcpy (name, str);
    mark = mark;
    size = size;
}
```

ic allocation

size);

t size);

Friend class:

```
class first
{
    int f1;
public:
    void getdata () { cin >> f1; }
    void display()
    friend second;
};
class second
{   int S2;
public:
    void getdata();
    void display();
};
int main()
{
    first F;
    seconds;
→ S.getdata();
  F.getdata();
  F.display ();
}
```

26/09/2013

```
Second :: getdata()
{
    cin >> f1 >> s2;
        first :: display();
}
first :: display()
{   cout << f1 << s2;
}
```

26/02/2013

OPERATOR OVERLOADING:-

* To enhance the properties of an existing operator

eg: +
*
-
==
new ✓
delete ✓

* To enhance the properties of an operator not only to add built in data types but also user defined data types

* void operator +( )

keyword

eg: Complex Return type

operator getting overloaded

{

}

Complex operator ++( )

{

}

For Addition:

```
class complex
{
    int real;
    int img;
public:
        void getdata()
        {
          cin>>real>>img; }
    void display()
    {
       cout<<real<<img;
    }
};

complex operator +(complex c2);

};

complex complex:: operator +(complex c2)
{
      real : c2.real + real;
      img : c2.img + img;

       return c1;

}
```

```
int main()
{
    Complex c1, c2, c3;
    c1.getdata();
    c2.getdata();
    c3 = c1 + c2;
    c3.display();
}
```

(⇒ c1.+(c2) ; explicitly passed; explicitly passed)

For Postfix & Prefix increment:

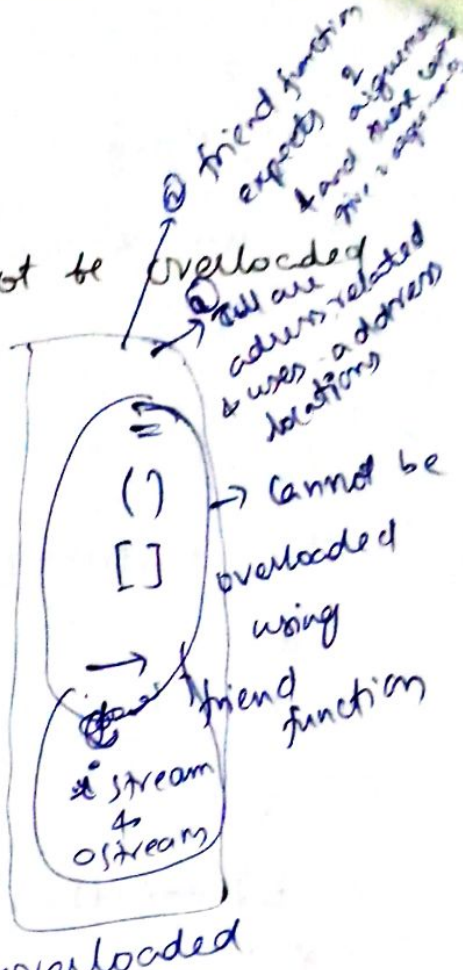
```
class complex
{
    int real;
    int img;
public:
    void getdata();
    void display();
    void operator ++(int c);   // postfix
    void operator ++();        // prefix
};
```

```
void complex :: operator ++(int ());
{
    real ++;
    img ++;
}

complex .complex :: operator + (complex (2)
{
    real = (2.real + real;
    img = (2.img + img;
    return (1;
}

void complex :: operator ++()
{
    ++ real;
    ++ img;
}

int main()
{
    complex (1, (2, (3;
    (1.getdata();
    (2.getdata(),
    (1++;
    (3 = (1 + (2;
    ++(2;
```

```
    c3 display();
}
```

* The operators which cannot be overloaded

1) : :
2) ?:
3) * pointer access
4) . (member access)
5) sizeof().



* new and delete → operators can be overloaded

STRING OPERATIONS :-

* Concatenation of 2 strings.

```
class string
{
    char *name1;
    char *name2;
public:
```

```
class string
{
  char *name
  .
  .
};
```

→ operator overloading should be declared

```
string :: string (char *str)
{
    ____ = new
}
int main()
{
    String S1, S2, S3;
  char str2[30];
  S1("Hai");
  cin >>
S2(str2);
S2. getdata();
S3 = S1 + S2;
}
```

```
class concatenate
{
    char *str;
public:
    concatenate(char s[])
    {
        str = new char[100];
        strcpy(str, s);
    }
    char * void operator +(complex s2);
};
char * concatenate :: operator +(complex s2)
{
    strcat(str, s2.str);
    return str;
}
void main()
{
    concatenate s1("Hai");
    concatenate s2("Bye");
    s3 = s1 + s2;
}
```

```
class complex
{
    int real;
    int img;
Public:
friend returntype operator +(complex c1, complex c2)
void getdata();
void display();
};
```

```
Complex operator +(complex c1, complex c2)
{
    Complex temp;
  temp.real = c1.real + c2.real;
  temp.img = c1.img + c2.img;
    return temp;
  }
  int main()
{
    Complex c1, c2, c3;
    c1.getdata();
    c2.getdata();
```

```
c3 =
c3
}.
```

→

```
class
{
public:
friend
friend
friend
void
void
}
Com
{
```

c3 = c1+c2;
c3.display();
}

→

1). Plus
2) Negation (a, -a)

complex (2)

class complex
{
int real;
int img;
public:
friend complex operator + (complex c1, complex c2);
friend complex operator * (complex c1, complex c2);
friend complex operator - (complex c1, complex c2);
void getdata();
void display();
};

complex operator *(complex c1, complex c2)
{
complex temp;
temp.real = c1.real * c2.real;
temp.img = c1.img * c2.img;
return temp;
}
complex operator *-(complex c1, complex c2)
{
complex temp;

```
temp.real = c1.real - c2.real;
temp.img  = c1.img - c2.img;
return temp;
}

int main()
{
  Complex c1, c2, c3;
  int n;
  c1.getdata();
  c2.getdata();
  cout<<" enter your choice"<<'1. multiplication"
<<'2. Substraction";
  cin>>n;
  Switch (n)
{
  Case 1 :
    c3 = c1*c2;
    break;
  case 2:
    c3 = c1-c2;
    break;
  Default:
  Cout<<"entered wrong choice";
  }
```

c3 display();

}

temp(real) = temp.real;

→ Date) Using ++(post fix) & ++(prefix) operator increment the date & check whether the given year is leap year (or) not.

class Date

06/03/13

iostream operator Overloading:-

cin ✓ istream ; cout - ostream.

cin } objects
cout }

```
class complex
{
        int real;
        int imag;
public:
    friend istream & operator >>( istream & input,
                                  complex & c1);
friend ostream & operator << (ostream & output,
                              complex & c2);
};
istream & operator >> (istream & input ; complex &
                                          c1 );
{
        input >> c1.real;
        input >> c2.imag;
        return input ;
}
int main()
{       complex c1, c2;
        cin >> c1
          cin >> c2
        cout << c1
        cout << c2;
}
```

* string

* string

* Date

class

{

publ

};

void

{

}

* string concatenation

s1+s2

* string compare

s1 < s2

(or)

s1 > s2

* Date increment:

```
class date
{
    int day;
    int month;
    int year;
public:
    void getdata();
    void display();
    void operator ++();
};
void date :: getdata()
{
    cout << "enter date (eg: 14 3 2013): ";
    cin >> day >> month >> year;
}
```

```
void operator++()
{
    int months[] = {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30
                    , 31};
    if (day <= months[month-1])
    {
        if (month == 2 && day == 28)
    {
        if (year%4 != 0 || year%100 != 0)
        {
            day = 1;
            month = 3;
        }
    else
    {
        day++;
        if (day > months[month-1])
        {
            day = 1;
            if (month == 12)
            {
                month = 1;
                year++;
            }
        }
```

```
main()
{
    date d1;
    d1.getdata();
    ++d1;
```

String Operations:

```
class string
{
    char *str1;
    char *str2;
public:
  friend istream & operator >> (istream & in,
                                string &s);
{   string
    in>> str;
}
friend ostream & operator << (ostream & out,
                              string &s);
{
    out << str;
}
```

```
class s
{
    char str[50];
public:
    friend istream &operator>>(istream &input, s &s1);
    friend ostream &operator<<(ostream &output, s &s1);
    friend s operator+(s1, s2);
};
istream &operator>>(istream &input, s &s1)
{
    input >> s1.str;
};
ostream &operator<<(ostream &output, s &s1)
{
    output << s1.str;
};
int operator+(char s1, char s2)
{
    int a;
    a = strcmp(s1.str, s2.str);
    return a;
}
```

```
int main()
{
    int i;
    string s1, s2;
    cin >> s1;
    cin >> s2;
    i = s1+s2;
    if (i == 0)
    cout << "strings are equal";
    if (i<0)
    cout << "s1 is longer";
    else
    cout << "s2 is longer";
    cout << s1;
    cout << s2;
}
```

12/03/2013

Conversion from one class to other class:-

class Radian ⟷ class degree

* Using operator
* Using constructor

rad = deg * (π/180);

deg = rad * (180/π);

Symantic - meaning
Syntax - Rule

* we should not change symantics of the operator

Important

R1(D1) = R1 = D1

class d

Return

*

Return by Reference:

* pass by reference

* whenever we pass by reference no copy will be present.

rvalue space location

a[3] = 6

memory location

→ Ex: student s → object, class has array marks ⇒ marks[]

s[4] = 90 → array 4th location stored = 90

so, '4' is used

```
int & operator [] ( int x )
{
    return marks[x];
}
```

marks[position]

returns memory location

*

13/3/13

- Basic data type to User defined obj
- Conversion b/w different defined types

```
float length;
meter m1
```

length = m1 → operator float()
{
cm = meter * 100;
}

m1 = length → from wes
⇓
m1(length) → parameterised constructor

*Q: metre to cm:

```
class metre
{
    meter m1;
public:
    m1.getdata();
operator float();
{
    cm = m * 100;
    return cm;
}
```

```
int main()
{
    float length;
    metre m1;
    cin >> length
    length =
```

15/03/2013

Subscript

operato
{

float setwidth()
{
float length;
cout << "enter ml";
cin >> length;
length = ml
}

18/02/2018

Subscript operator overloading: this can be done also by constructor

operator [] (int x, int y)
{
sal = x;
id = y;
return *this;
}

*this; → returns which object has invoked that obj only
return → returns current object

Inheritance:

```
Employee
{
private:
    sal;
    empid;
public:
    getdata();
    display();
};
```

```
class student : public Employee
{
    regno;
    name;
public:
    input();
    output();
};
```

visibility

```
Employee
{
protected:
    sa
    em
public:
    ge
    D
};
```

✓ Employee E1

E1.getdata();

E1

```
sal;
empid;
getdata();
display();
```

✓ student S1

S1.input();

S1

```
private E1
    sal
    empid
private S1
    regno;
    name;
input();
output();
getdata();
display();
```

input()

{ cin >> sal >> engno;

}

→ public visibility mode

```
E1
priva
  e
  e
pub
  g
  d
```

A

private

pro

visibility

public Employee

ity mode

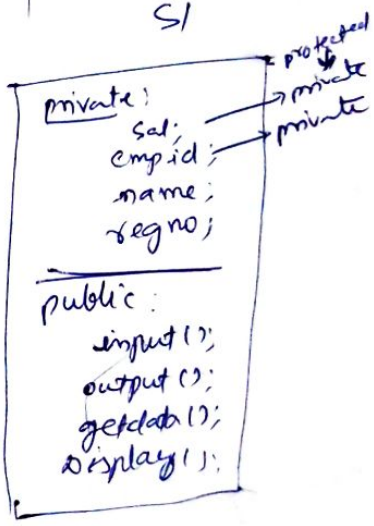
* Eg using protected.

```
Employee
{
  protected:
      sal;
      empid;
  public:
      getdata();
      Display();
};
```

```
Student : public Employee
{
    regno;
    name;
  input();
  output();
};
```

E)

S)



Accessing:

private:

```
input()
{   cin >> sal >> empno;   ✗
}
```

protected:

```
input()
{   cin >> sal ; empno;
}
```

* If visibility is protected

input() → getdata() → empid, sal.

empid, sal:

date increment program:

```
#include <iostream>
#include <conio.h>
using namespace std;
class date
{
    int day;
    int month;
    int year;
  public:
    void getdata()
  { cin >> day >> month >> year;
   }
   void display()
   { cout << day << "\t" << month << "\t" << year;
  }
  int leapyear_check()
  {
  if((year%4==0 && year%100!=0)||(year%400==0))
  return 1;
  else
  return 0;
  }
  int thismnthmaxday()
{
    int m[12]={31,28,31,30,31,30,31,31,30,31,30,31};
  if(month==2 && leapyear_check())
  { return 29;
   } else
```

```
{ cout<<"\n"
  return m[month-1];
  } }
void operator ++()
{
   day++;
  if (day > thismonthmax day())
{
   day=1
  month = ++months;
  }
if (month>12)
{ month =1;
  year = ++year;
}
} };
void nextday (date &d)
{    ++d;
  cout << "date after increment is :";
  d.display();
  }
int main()
{   date today;
cout<<"enter the date (dd mm yyyy)";
today .getdata();
nextday (today);
getch();
return 0;
}
```

Concat. of string using ptr (by passing arguments)

```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
class str
{   char *name;
public:
 void getdata()
{    cin>>name;
  }
void display()
{
    cout<<name;
  }
  str(){
{  name = new char[20];
 }
 str(char *namein)
{  name = new char[20];
  strcpy(name, namein);
}
str operator +(str s2)
{
   str s3;
   strcpy(s3.name, name);
   strcat(s3.name, s2.name);
   return s3;
  }
  };
```

```
int main()
{
   Str S1 ("welcome to ");
   Str S2 ("operator overloading");
   Str S3;
S3 = S1 + S2;
   cout<<" after concatenation";
   S3.display();
   getch();
   return 0;
   }
```

String comparison , iostream operator overloading:

```
#include <iostream>
#include <conio.h>
#include< string.h>
using namespace std;
class Str
{
    char name[20];
public:
   friend istream & operator >>(istream &in,
                                Str & S1)
{
   in >>S1.name;
} return in;
}
```

```
friend
{
   out<
}
int op
{  int
   a =
   ret
}
};
int main
{
   St
   co
   c
   co
```

```
friend ostream & operator <<( ostream &out,
                              str &s1)
{
  cout << s1.name;
}
int operator > (str s2)
{  int a;
   a = strcmp( name, s2.name);
   return a;
}
};
int main()
{  int a;
   str s1, s2;
cout<<" enter the first name:";
cin >> s1;
cout<<" enter the second name:";
cin >>s2;
cout << " string 1 is :";
cout<<s1<<"\n";
cout << " string 2 is :";
cout<< s2;
a = s1>s2;
if (a==0)
{ cout << " name 1 is equal to name 2";
}
```

```
else if ( a == 1)
{ cout << name 1 is greater than name 2";
}
else
{ cout << "name 2 is greater than name 1";
}
getch();
return 0;
}
```

Data Conversion:

* Conversion b/w basic data types:

type casting

weight = (float) age ⇒ int

float

weight = float(age);

* Conversion b/w objects and Basic types:

* basic to userdefined:

```
constructor (Basic type)
{
    // steps for converting
    // Basic type to object attributes
};
```

syntax for invocation:

M1 = length (or) M1 (length)

name 2":

name 1":

* user defined to basic:

```
operator Basictype ()
{
   // Steps for converting
   // object attributes to Basic type
}
```

Syntax for invocation:

length = M1
↓ float ↓ metre class object.

Example:

* converting the units from metre to cm & cm → metre

Program:

```
#include <iostream>
#include <conio.h>
using namespace std;
class metre
{
   float metre;
public:
   void getdata()
   { cin >> metre;
   }
   void display()
   {
     cout << metre;
   }
   metre()
   {
     metre =0;
   }
```

```
    metre(float cm)
    {
     lmetre = cm/100.0;
    }
    operator float()
    {
     float cm;
     cm = lmetre * 100;
     return cm;
    }
};
int main()
{
    metre m1, m2;
    float lcm, lcm1;
    cout<<" enter the length in metre:";
    m1.getdata();
    lcm = m1;
    cout<<" length in cm:";
    cout<<lcm<<"\n";
    cout<<" enter the length in cm";
    cin>>lcm1;
    m2 = lcm1;
    cout<<" length in metre:";
    m2.display();
    getch();
    return 0;
}
```

26/03/2013

Inheritance

26/03/2013

Inheritance:-

```
class Base2;
class Base1;

class Derived : public Base1, public Base2;
```

 → avoids ambiguity

```
void getdata()
{
   Base1 :: getdata();
   cin >> a >> b >> c;
}

class Base1
{
   getdata();
}
class derived
{
   getdata();
}
int main()
{
   Derived D1;
   D1. getdata();
   D1. Base1 :: getdata();
```

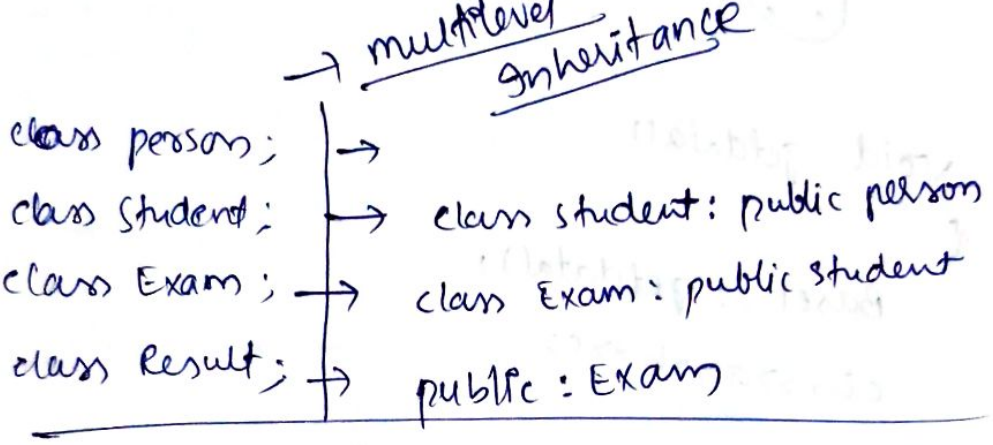
Base B1;
B1.getdata();

** distinguishing property.

forward declaration :-

```
class A;
class B;
class C;
class D
{
};
```

Ex:




```
class person
{
    char name[20];
    int id;
    char dob[15];
public:
void getdata();
};

class student : public person
{
    int regno;
    int no-of-subjects;
    char name[20];
public:
void getdata();
};
```

```
class Exam : public student
{
        int regno;
        int no_of_subjects;
        int marks [no. of subjects];
        int rank;
        int totalmarks;
public: void getdata();
```

```
class person
{
protected:
        name;
        address;
        dob;
    getdata();
    display();
    }

class student: public person
{
    private:
        regno;
protected:
        sub;
public:
        getdata();
        display();
};
```

```
class Exam : public student
{
    protected:
        m1;
        m2;
    public:
        getdata();
        Display();
};
class Result : public Exam
{
    public:
        avg();
        Rank();
        display();
        getdata();
};
int main()
{
    Result R1;
    Exam E1;
    Student S1;
    Person P1;
}
```

09/04/2013

Construct

Base

Derived

Derived

Differ

*) No

Cla

a)

3)

4)

5)

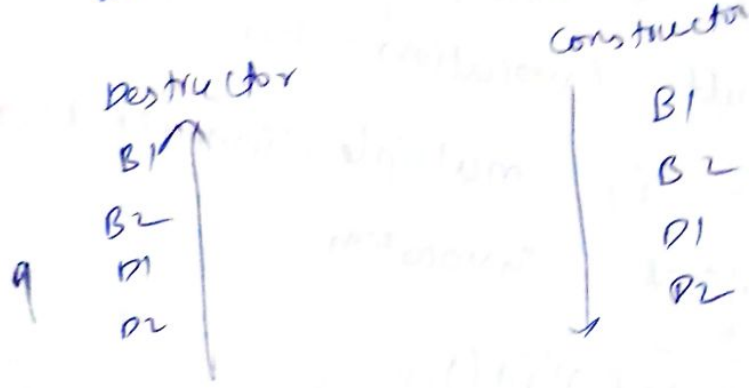
09/04/2013

Constructors in Base and Derived class

Base1

Derived1 : public B1

Derived2 : public D1, virtual B1 ,



Different combinations of constructors:

*) No constructors in Base class and Derived class

a) Constructors only in base class

b) Constructors only in derived class

c) Constructors in both the derived the Base classes

5) Multiple constructor in Base class
single constructor in derived class

c) constructors in base and derived classes without default constructor

7) Explicit invocation in the absence of default constructor

```
D(int a) : B(a)
```

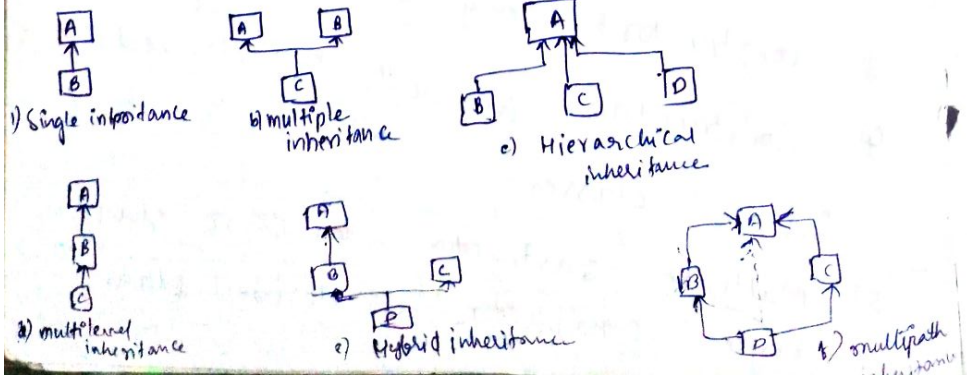
8. Constructors in multiple inherited class with default invocation

9. constructors in multiple inherited class with explicit invocation

```
D() : B2(), B1()
```

10. constructors in Base and derived class in multiple inheritance (virtual)

11. Multilevel inheritance.



derived classes

absence of

d class.

ted class

class in

multipath
inheritance

omsairam

```
class employee
{
    employee ()
    { }
};
```

FIFO
first in first out

```
class student : public employee
{
    student ()
    { }
};
```

*) order

```
class Result : public student, public B2, public B1,
               public B3;
```

order of const. invocation:

student, B2, B1, B3

If B1 is virtual:

virtual class const will be invoked

If two are virtual:

superMost virtual will be invoked first

10/04/2015

class derived

{

private:

public:

D(a,b) : B(x,y);

};

class Base

{

private:

b1;

b2;

public:

B(a,b) { b1 = a;

b2 = b;

}

(or)

B(a,b) : b1(a), b2(b);

};

int main()

{

derived D1(2,3);

Base B1(3,4);

}

Employee

ptr =

virtual

Emp

pu

de

};

Stud

{

Employee *Eptr;

Eptr = &E1;

Virtual display();
↳ avoids the ambiguity

virtual
Employee
public:
display();

D1.B1::getdata();

};

Student
:
display();
};

Eptr → base
Eptr → display()

Eptr = &S1

Eptr → display()
↳ ~~derived~~ base (if virtual)
↳ derived

* Run time polymorphism
namemangling operation } → compiler

* ~~Standing~~ static polymorphism
↳ During compilation
↳ * operator overloading
* function overloading

Objects ∧→ user defined data types

* virtual function
↳ Run time polymorphism.

* R.T. polymorphism because we are using pointers.

```
Employee *Eptr;
Eptr = &E1
Eptr -> display();
delete Eptr;
Eptr = &S1;
Eptr -> display();
```

* Derive ptr → Base ptr ✗
* Base ptr → Derived ptr ✓

static polymorphism → operator overloading, function overloading

Runtime polymorphism → virtual function

we are using

* pure virtual function
  * no arguements
  * no function definition
* later can be defined

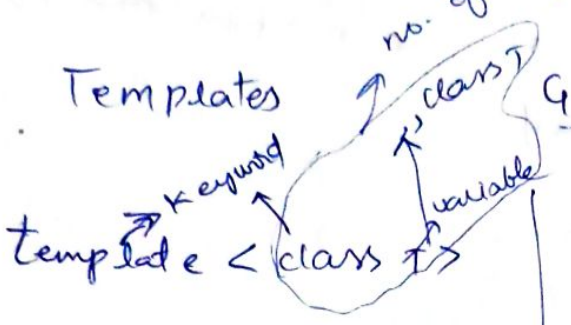
```
class employee
{
    data members
public:
virtual void process() = 0;
};
class student
{
    data members
public:
    void getdata();
    void display();
};
main()
{
    student s1;

    return 0;
}
```

→ pure virtual function

erloading
loading

12/04/2013

Templates — Generic data type




```
T sum (T a, T b)
{


return ___
}
```

```
main()
{
  sum (3, 4)
  sum (5.4, 3.5)
}
```

```
template < class T>
T bubblesort (T array)
{

}
```

```
main()
{
  bubblesort (int array);
  bubblesort (char array);
  bubblesort (float array);
}
```

```
template<class T>
```

```
T Integerclass <T> :: process ()
```

Home wo

Q. 1

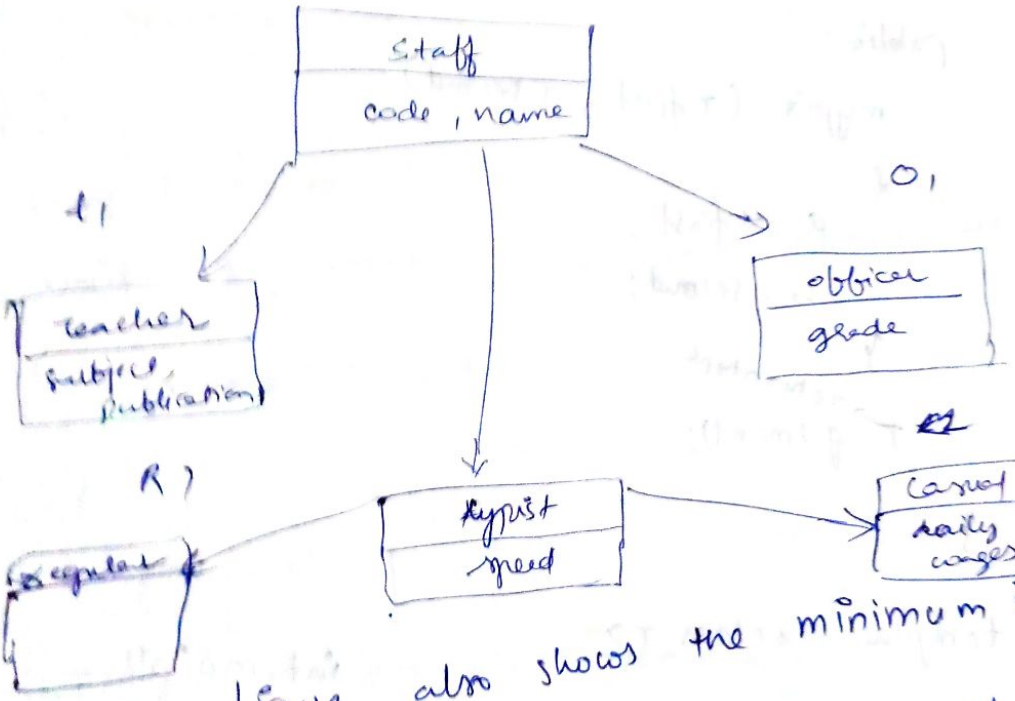
dat

div

h

Home work:

8. An educational institution wishes to maintain database of its employees. The database is divided into no. of classes whose hierarchial relationships are as shown



The figure also shows the minimum information required for each class. Specify the classes and define functions to create a database and retrieve individual information as and when required.

13/04/2013

```
#include<iostream>
using namespace std;

template <class T>
class mypair
{
    T a, b;
public:
    mypair (T first, T second)
    {
        a = first;
        b = second;
    }
    T getmax();
};

template <class T>
T mypair <T> :: getmax()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}
```

```
int main()
{
    mypair <int> myobject
            (100, 75);
    cout << myobject.getmax();
    return 0;
}
```

program II:

```
#include <ios
using nam
template <c
class mys
{
    T mem
public:
    void s
    T get
};
templ
void
{
    m
}
ter
T
{
```

program II:

```
#include <iostream>
using namespace std;
template <class T, int N>          // T: generic data type
class mysequence
{
  T memblock[N];
public:
void setmember(int x, T value);
 T getmember(int x);
};
template <class T, int N>
void mysequence<T, N>:: setmember(int x, T value)
{
  memblock[x] = value;
}
template <class T, int N>
T mysequence <T, N>:: getmember(int x)
{
   return memblock[x];
}
int main()
{
   mysequence <int, 5> myints;      // objname
   mysequence <double, 5> myfloats;
   myints.setmember(0, 100);
```

> myobject
(100, 75);
t.getmax();

```
myfloats.setmember(3, 3.1416);
cout << myints.getmember(0) << "\n";
cout << myfloats.getmember(3) << "\n";
return 0;
}
```

16/04/2013

1Q. Write a function template for finding the minimum value contained in an array. ~~with a class templa~~

2Q Write a class template to represent a generic vector. include member functions to perform the following tasks

i) To create a vector
2) To modify the value of a given element
3) To multiply by a scalar value
4) To display the vector in the form of

10, 20, 30.

1) template <class T>

```
void minimum (T a[], T l)
{
    T min, i;
    min = a[0];
  for(i=0; i<l; i++)
   if (a[i] <min)
     min = a[i];
  cout<<" minimum value is :"<<min;
}
main()
{
   int a[20], l, i;
  cout<<" Enter the length of the array";
   cin>>l;
  for (i=0; i<l; i++)
    cin >> a[i];
  minimum .(a, l);
}
```

2. template < class T >

```
class vector
{
    T x, y, z;
public:
void getvector (T a, T b, T c)
{
    x = a;
    y = b;
    z = c;
}
void modifyvector ()
{
cout << " enter new values for the vector ";
int a, b, c;
cin >> a >> b >> c;
x = a; y = b; z = c;
}
void multiply ( T a)
{
    x = a * x;
    y = a * y
    z = a * z;
}
};
int main ()
{
    vector <int> v;
v. getvector (5, 4, 3);
v. modifyvector ();
v. multiply (4);
}
```

```
vector <float> v1;
v1. getvector (5.1, 4.1, 3.1);
v1. modifyvector ();
v1. multiply (4.32);
```

17/04/13

FILES

iostream - Keyboard

fstream - file

istream - to get data from K-b

ostream - to display data

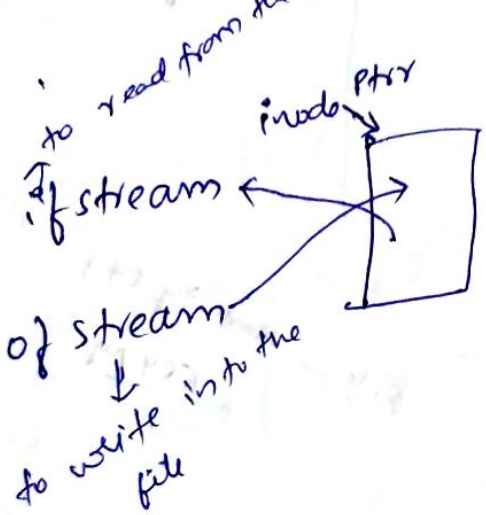
ifstream - input file operation

ofstream - output file operation

inode → starting pointer

file name ⇒ from which file has to be created.

↳ keyword




```
ofstream out ("student file")
char name [30];
cin >> name;
out << name;
out.close();
```

```
ifstream in ("student file")
char studentname[40];
cin >> name.studentname;
cout << studentname;
while (!(read.eof()))
{
  read operation;
}
```

```
out.open();
cin.getline(buffer,
cin.getch();
```

19/4/13

2Q Binary files

ASCII files

input output stream.

Refer mastering C++ 684 pg

ios::bin → opens file in binary format

Binary file (NO) → binary conversion → storage

ASCII (M) → storage

file. fin.open ("studentfile", ios::app → append mode)
(filename, mode)

of stream . fout ("student");

class employee
{
    name;
    eid;
    getdata();
    display();
};

of stream fout;

fout.open ("empfile", ios::out)

fout.write (char *& (employee object), sizeof (empobject))

fout.close()

if stream . fis;

fis.read (pointer, sizeof());

finout.open (w)

finout.close()

finout.open()

finout.read()

finout.close()

20/4/13

fstream object

fs.open(filename, ios::in | ios::out | ios::bin | ios::app

moving file ptr.

+ → forward
- → backward

fs.read();
fs.write
fs.eof()

fs.seekg(0, ios::beg
cur
end)

seekg(-2, ios::end)
↳ error

fs.tellg → to know the position of file ptr.

get pointer - while reading
put pointer - while writing

If we want to write whole object:

fstream finout ("", ios::in|ios::out)

finout.write((char *)&s1, sizeof(s1));

finout.close();

finout.read((char *)&s2, sizeof(s2));

s2.display();

23/4/13

Comma

int
{

23/4/13

Command line arguments.

```
int main ( int argc , char * argv[] )
{

}
```

```
$ g++ exp.cpp
$ ./a.out I → second argument value
          first argument value
```

```
fout.open ( argv[i] , ios::out )
```

24/4/18.

Exception handling:

```
try
{
}
catch()
{
}
throw
```

```
class Positive { };
class negative { };
class zero { };
class number
{
  int num;
  public:
  check()
  {
    if num<0
      throw negative;
    if num>0
      throw positive;
    else
      throw zero;
  }
```

```
int ma
{
  Num
try
{
  No. ch
}
catch
{
  cout
}
catch
{
  cou
}
cat
{
  C
}
}
```

```
int main()
{
    Number N1
try
{
 N1.check();
  }
 catch (number::positive)
{
 cout << " number is +ve";
}
catch (number::negative)
{
 cout << " number is negative";
}
catch (number::zero)
 {
 cout << " number is zero";
 }
};
```

Static Allocation :- Allocation of memory space at compile time

Dynamic Allocation :- Allocation of memory space at run time / execution time.

"malloc()" and "calloc()" are used to "allocate" memory dynamically at run time. The function "free()" to free dynamically the allocated memory } C & C++

new → create an object
delete → destroy an object } C++

pointer variable = new datatype;
↳ allocates memory
↳ holds address of the memory space allocated

Pass by value:-

a "copy" of the entire object is passed to the function

Pass by reference:-

only the "address" of the object is passed "implicitly" to the function

Pass by pointer:-

The address of the object is passed "explicitly" to the function.

COPY CONSTRUCTOR:-

A constructor having a reference to an instance of its own class as an argument is known as "copy constructor"

Type definition (typedef):-

* It is used to create a new data type which is equivalent to the existing data type.

* from then, this new data type can be used to declare variables

example: typedef int age;

age x, y;

Typecasting :-

The compiler can be instructed explicitly to perform type conversion using the type conversion operators known as "typecast operators"

Ex:- int to float

weight = (float) age (or) in C++ weight = float(age)

* Type casting is done by user and we may lose data . from wider (float) to smaller (int).

* Type conversion is done by compiler and we will not lose date * from smaller (int) to wider (float)

for 1-D array

```
int * grades = new int [no of grades];

delete [] grades;
```

for 2-D

```
for(i=0; i<rows; i++)
delete [] matrix[i];

delete [] matrix
```

3-D:

```
for(int i=0; i<xsize; i++)
{
  for( int j=0; j<ysize; j++)
  {
    delete [] volume [i][j]
  }
  delete [] volume [i] ;
```

Prime numbers generation:

```
#include <iostream>
#include <conio.h>
using namespace std;
void primes (int n)
{
    int i, j, c=0;
    cout<<" prime numbers:";
    for(i=n; i>=0; i--)
    {
    for(j=1; j<i; j++)
    {
     if((i%j)==0)
    c++;
    }
    if (c==1)
    cout<<i<<" ";
    c=0;
    }
    }
int main()
{
  int n;
cout<<" enter the number:";
cin>>n;
primes(n);
getch();
return 0;
}
```

Prime number check:

```
#include <iostream>
#include <conio.h>
using namespace std;
void primecheck(int n)
{
  int i, j;
  int c=0;
for(i=1; i<n; i++)
{ if((n%i)==0)
 c++;
 }
if (c==1)
cout<<" the given number is
    prime number";
else
cout<<" the given number is
not a prime number";
}
int main()
{
  int n;
cout<<"enter the number:";
 cin>>n;
primecheck(n);
getch();
return0;
 }
```

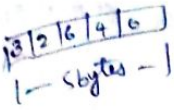
dynamic memory allocation for 2-D array

```
int **p;
p = new int *[x];
for (i=0; i<y; i++)
p[i]=new int [y];
```

3-D - array expectation

```
int ***p;
p = new int **[x]
for (i=0; i<y; i++)
{
p[i] = new int *[y];
for (j=0; j<z; j++)
p[i][j] = new int [z];
}
```

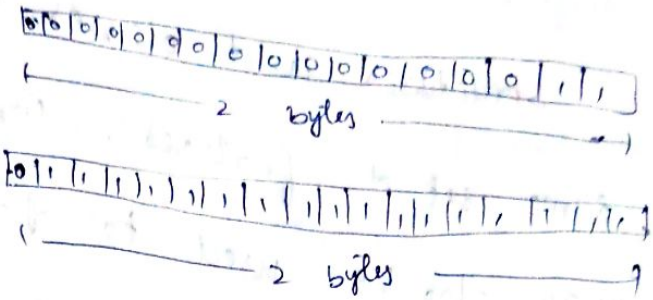
ASCII format



* Representation size varies according to magnitude

* File I/O requires data conversion, Binary to ASCII while writing to file and ASCII to binary while reading a file

Binary format



* Representation size remains constant irrespective of magnitude

* File I/O requires no conversion of data and hence fast access to a file

Difference between structures and classes :-

Structures and classes in C++ are given the same set of features. For example, structures may also be used to group data as well as functions.

In C++, the difference between structures and classes is that by default structure members have "public accessibility", whereas class members have "private access control" unless otherwise explicity stated. The declaration for a structure in C++ is similar to a class specification.

Data Abstraction and Encapsulation.

Encapsulation:

Encapsulation is a mechanism that associates the code and the data it manipulates and keeps them safe from external interference and misuse

Data Abstraction:

Creating new datatypes using encapsulated items that are well suited to an application to be programmed is called data Abstraction

Abstract data types:

The data types created by the data abstraction process are known as Abstract data types (ADTs)

* The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.

* these functions provide the interface between the object's data and the program. The insulation of the data from direct access by the program is called data hiding

* Information hiding

UNIT-I

a) Class:

Object oriented programming constructs models out of data types called classes

A class is a way to bind the data and its associated functions together

2) It allows the data (and functions) to be hidden if necessary, from external use

class specification has two parts

1) class declaration → describes the type & scope of its members

2) class function definitions → describes how the class functions are implemented

C.D:

class classname

{

AS

—

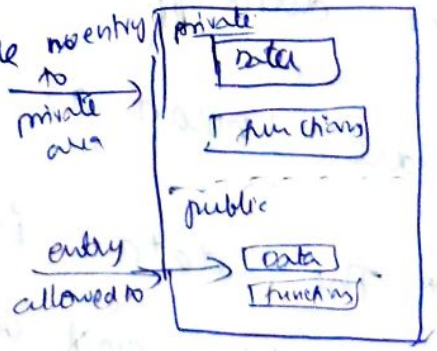
—

AS

—();

—();

};



Object:

Defining variables of a class data type is known as a 'class instantiation' and such variables are called objects

* object is a smallest unit of a program
* It may represent a person, a place, or a thing with which the computer must deal

objects can be

1) Concrete (eg. file system)
2) Conceptual (eg scheduling)

* "empty classes" ⇒ "stubs" significant use can be found with "exception handling" during initial stages of the project.

⑥ deg to Radian using constructor in destination object

```
#include<iostream>
using namespace std;
class degree
{   float deg;
public:
    void getdata()
{   cin>>deg;
 }void display()
 {  cout<<deg;
    }
    float getdegree()
    {  return(deg);
    }
 };
class radian*
{
   float rad;
public:
   void getdata()
   {  cin>>rad;
    }
   void display()
  {  cout<<rad;
   }
   radian()
 {  rad=0;
  }
radian(degree d1)
{
rad=(d1.getdegree()*(3.14/180);
}
};
int main()
{
  radian r1;
  degree d1;
  d1.getdata();
  r1=d1;
cout<<"angle in terms of
      radian is =";
 r1.display();
 getch();
 return 0;
  }
```

Sample program of friend class:

```
#include<iostream>
using namespace std;
class first
{    int x1,x2;
public:
    void getdata()
   {   cin>>x1>>x2;
    }
    void display()
    {  cout<<x1&<x2;
     }
    friend class second;
};
class second
{    int y1,y2;
public:
    void getdata()
    {  cin>>y1>>y2;
     }
   void display()
   {cout<<f.x1<<f.x2<<y1<<y2;
   } };
int main()
{   first f;
    Second s;
    f.getdata();
    s.getdata();
    s.display(f);
                ...getch(); }
```