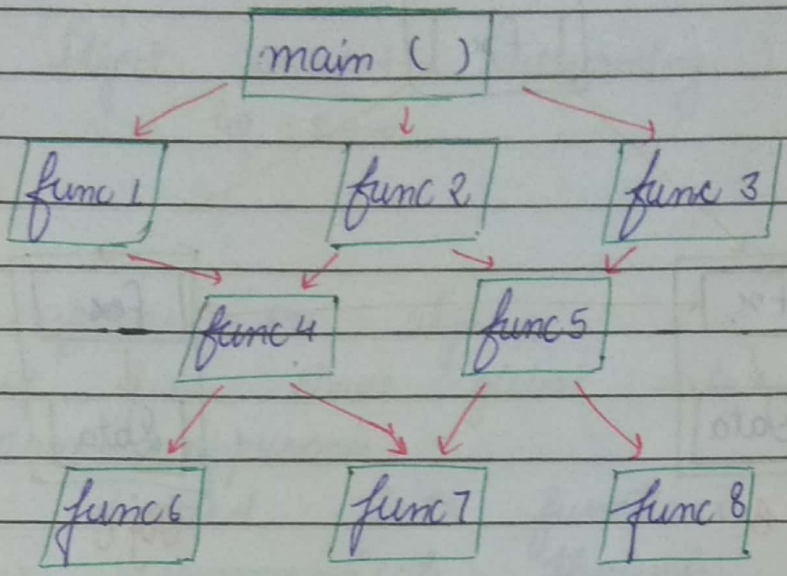
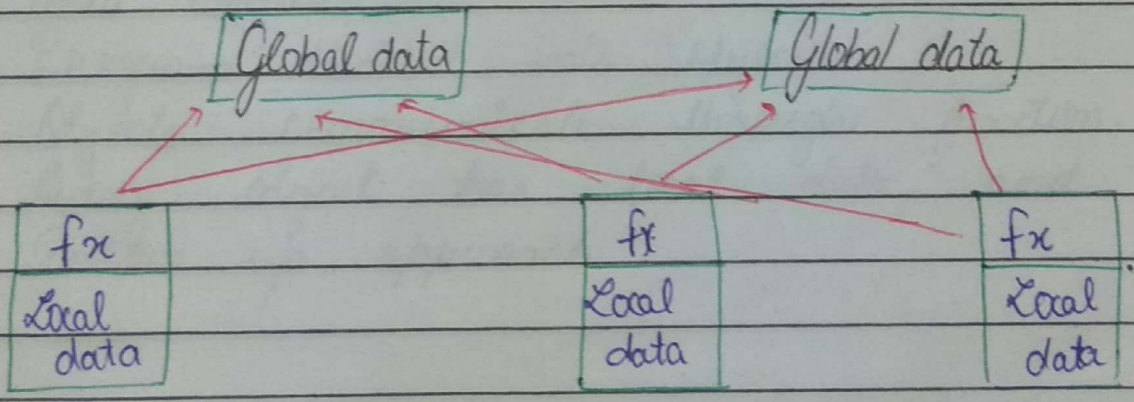
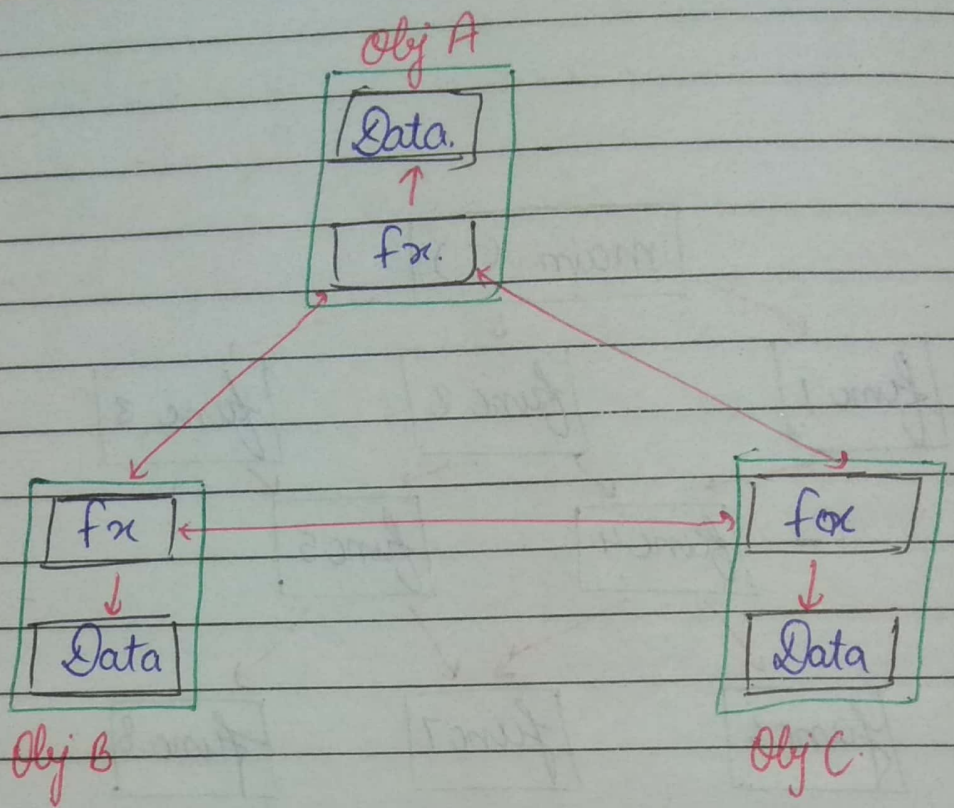
Date: 29/8/18 Page no:



Structure of POP



Data & function in POP



Data and function in OOP

POP → Procedure oriented programming
eg : C

OOP → Object oriented programming
eg : C++

POP.
Main emphasis on algo.
Most of func share global data.
Top bottom approach.
Program divided into functions.
Functions communicate through data transfer.

OOP.
Main emphasis on data member (variable) and data member.
Program divided into object.
Objects communicate through function.
Every object has local data and function.
Bottom up approach

Characteristics of OOP

1. Class
2. Object
3. Data abstraction
4. Data Encapsulation
5. ~~Data Hiding~~ Inheritance
6. Data Binding
7. Polymorphism
8. Message passing

(16) Class : It is a collection of data member and member function

Object : It is a basic ~~write~~ run time entity used to access data member and member function.

(4) Data abstraction : Act of representing essential features without showing background details and features. (explaination)

Data encapsulation : Wraping up of data member and member function into a single unit.

(24) Inheritance : It is a process in which properties of one class is acquired by another class. Used for code reusability.

Polymorphism : Ability to take more than one form.

- Compile time
  - → func. overloading (2)
  - → operation overloading (3)
- Runtime → virtual fn. (2)

Date: __/__/____ Page no: ____

Data binding : It is a process of binding object to function.

Late binding (Dynamic)

Early binding (Static)

Message passing : When one object needs to communicate to another object it takes help of message passing.

Data Types

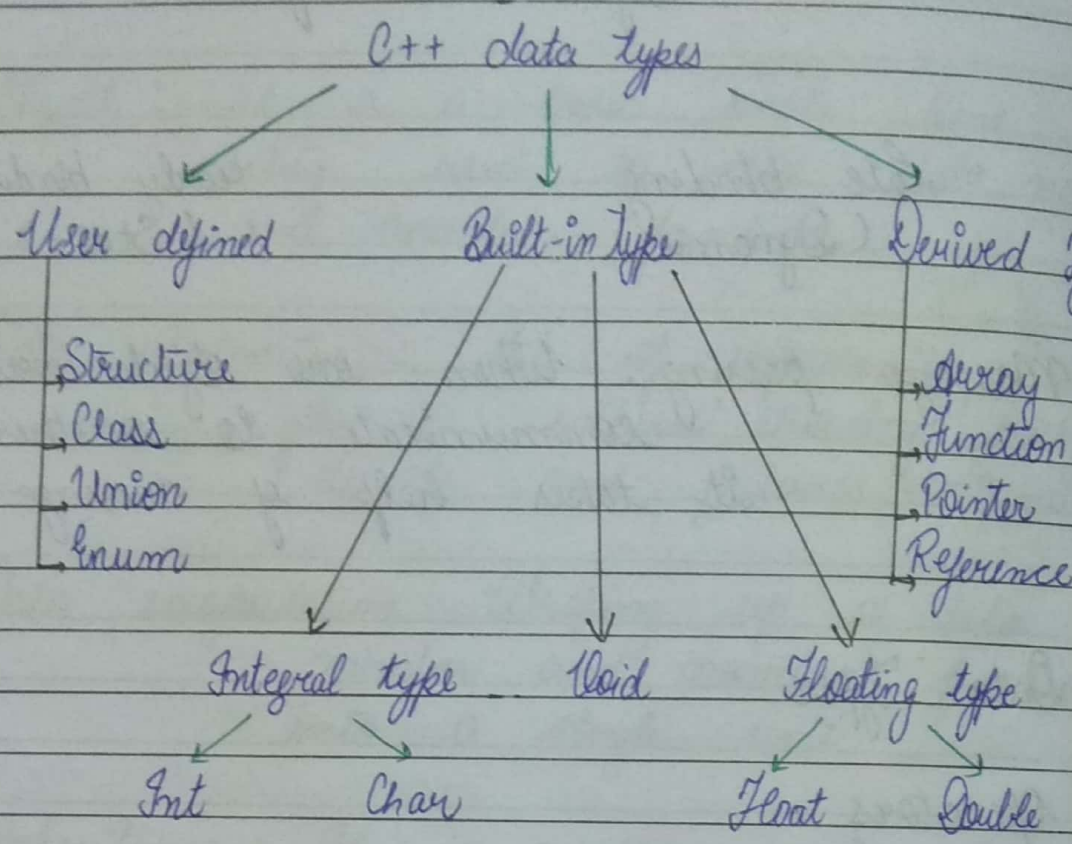
Operators

Keywords

Functions

Date: 18/18 Page no:

Data Types in C++



Inline function

```
inline float mul(float x, float y);
{
    return x*y;
}

inline double div(double x, double y)
{
    return x/y;
}
```

```
main ()
{
    float a = 5;
    float b = 3;

    cout << "Mul =" << mul (a,b);
    cout << "Div = " << div (a,b);
}
```

NOTE:
endl : To go to nextline.
<< : insertion operation.
>> : extraction operator.

Eg :-
```
cout << "Hello";
cout << "\n";
cout << "Hi";
cout << endl;
cout << "How are you";
```

or

```
cout << "Hi" << "Hello";
```

Program to add two numbers

```
#include <iostream.h>
main()
{
    int a, b, c;
    cout << "Enter two numbers";
    cin >> a;
    cin >> b;
    c = a + b;
    cout << c;
}
```

or

```
cin >> a >> b;
```

4 marks short note.

Inline Function.

Inline function can include only those functions which are single lined.
If function consists of loop, goto statement or static variable, function can not be declared inline.
It is used to reduce the time taken after function calling to jump to that function defination and to return back to main function.
When inline functions are called they instead of jumping to the defination open up the codes to be executed by that function, and execute them at the time of calling itself.

Default arguments

1. int mul (int a, int b = 2, int c); // Error
2. int mul (int a = 5, int b, int c); // Error
3. int mul (int a, int b=3, int c = 7);
4. int mul (int a, int b, int c=5);
5. int mul (int a=2, int b=5, int c=7);
6. int mul (int a=3, int b=3, int c); // Error.

Default argument can not be set at intermediate position, similarly, default can't be set at first position.

Structure

Syntax:

```
struct structurename
{
    Data element 1 ;
    ---"---
}---"---;
```

2 How to create struct variable

```
struct structname variablename;
```

3 How to access

```
varname.dataelement;
```

```
struct student
{
  char name[30];
  int age;
  float marks;
};

main()
{
  struct student S;
  cout << "Enter name, age and marks";
  cin >> S.name >> S.age >> S.marks;
  cout << "The values are:";
} cout << S.name << S.age << S.marks;
```

Date: / / Page no:

Classes And Objects

Syntax.

```
class classname
{
    data member ;
    member function ;
    access specifier :
        data member ;
        member function ;
        :
        :
};
```

Creating object

```
classname objectname ;
```

Accessing class.

```
objname . datamember / member function ;
```

Class are by default private.

Access Specifier

- Private
- Public
- Protected

```
class Person
{
    char name[30];
    int age;
public:
    void getdata()
    {
      cout << "Enter name & age";
      cin >> name >> age;
    }

    void putdata()
    {
      cout << name << age;
    }
};

main()
{
  Person P;
  P.getdata();
  P.putdata();
}
```

Date: 5,9,18 Page no:

Inside Class Defination

```
class person
{
    char name[30];
    int age;

    public:
        void getdata()
        {
            cin >> name >> age;
        }

        void putdata()
        {
            cout << name << age;
        }
};

main()
{
    person p;
    p.getdata();
    p.putdata();
}
```

Outside Class Defination

```
returntype classname :: fn name (arg list)
{
  // fn body

}
```

Example:

```
class person
{
    char name[30];
    int age;

    public:
        void getdata()
        {
            cin >> name >> age;
        }

        void putdata();
};

void person :: putdata()
{
    cout << name << age;
}
```

Array of Object

```
main ( )
{
    person p[5];
    cout << " Enter details of five persons ";
    for ( i=0; i<5 ; i++ )
      {
          p[i].getdata( );
      }
    cout >>" The details are :";
    for ( i=0 ; i<5 ; i++ )
      {
          p[i]. putdata ( );
      }
}
```

(8) Static Data Member.

```
class item
{
    static int count;
    int no;

    public :
        void getdata (int a)
        {
            no = a;
            count ++;
        }

        void getcount ( )
        {
            cout << count;
        }
};

int item :: count ; // Define static data
                       member.

main ( )
{
    int item a, b, c;
    a.getdata a.getcount ( );
    b.getcount ( );
    c.getcount ( );
    a.getdata (100);
    b.getdata (200);
```

```
c.getdata(300);
a.getcount();
b.getcount();
c.getcount();
}
```

Output: 0 0 0 3 3 3

Default value of static data member is 0
Static data member is shared by all objects, ie, it is independent of objects.
It is visible within the class only but it is executable through out the program.

Static Member Function

```
class Test
{
    int code;
    static int count;
    public:
    void setcode()
    {
        code = ++count;
    }
    void showcode()
    {
        cout << code;
```

++count
code = count

```
    static void showcount ()
    {
      cout << count;
    }
};

int Test :: count;

main ()
{
  Test t1, t2;
  t1.setcode ();  → 1
  t2.setcode ();  → 2

2 ← Test :: showcount (); // Access static function
  Test t3;
  t3.showcode ();  → 3

3 ← Test :: showcount ();
  t1.showcode ();  1
  t2.showcode ();  2
  t3.showcode ();  3
}
```

→ Static member functions are called by class name.
Syntax : classname :: member f^n name

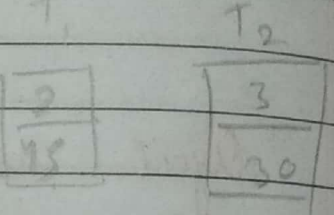
→ Static member functions can access only static data members.

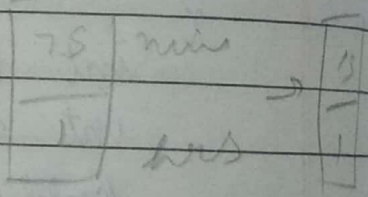
8 Sept '2018

Object as function arguments

```
class time
{
  int hrs ;
  int mins;
  public :
    void gettime (int h, int m)
    {
      hours = h ;
      mins = m ;
    }
    void puttime (void)
    {
      cout << hrs << mins;
    }
    void sum (time, time);
};
```

```
void time :: sum (time t1, time t2)
{
    mins = t1.mins + t2.mins
    hrs = mins/60;
    mins = mins %. 60
    hrs = hrs + t1.hrs + t2.hrs.
}

main ()
{
    time T1, T2, T3;
    T1.gettime (2,45);
    T2.gettime (3,30);
    T3.gettime Sum (T1, T2);

    T1.puttime ();
    T2.puttime ();
    T3.puttime ();
}
```





Friend Function

```
class sample
  {
    int a, b;
    public:
      void setvalue ()
        {
          a = 45;
          b = 20;
        }
      friend float mean (sample);
  };

float mean (sample x)
  {
    return float (x.a + x.b)/20;
  }


main ()
  {
    sample S;
    S.setvalue ();
    cout << mean (S);
  }
```

→ Doesn't belong to any class.
→ Can be declared in either private or public.
→ Generally take object as an argument.
→ It is defined as normal function outside class.
→ It is called as a normal function

```
class ABC ; // forward declaration
class XYZ
 {
    int x;
    public:
      void setvalue (int value)
        {
          x = value;
        }
    friend void max (XYZ, ABC);
 };

class ABC
 {
   int a;
   public:
     void setvalue (int value)
       {
         a = value;
       }
```

```
    friend void max (XYZ, ABC);
};

void max (XYZ m, ABC n)
{
    if (m.x >= n.a):
        cout << m.x;
    else:
        cout << n.a;
}

main ()
{
    XYZ obj1;
    ABC obj2;
    obj1.setvalue (10);
    obj2.setvalue (20);
    max(obj1, obj2);
}
```

Returning Object

```
class complex
  {
     double real;
     double img;
     public:
        void getdata (float double x, double y)
          {
             real = x;
             img = y;
          }

        void putdata ( )
        {
          cout << real << img;
        }

  friend complex sum (complex, complex);
  };

complex complex :: sum (complex c1, complex c2)
  {
     complex c3;
       c3.real = c1.real + c2.real;
       c3.img = c1.img + c2.img;
  } return c3;
```

```
main ( )
{
  complex A, B, C;
  A.getdata (2.5, 3.6);
  B.getdata (1.6, 2.1);
  C = sum (A, B);
  A.putdata ( );
  B.putdata ( );
  C.putdata ( );
}
```

Constructor

```
class test
{
  ---
  ---
  ---
  public:
    Test ( )
    {
      ---
      ---
    }
  ---
}  ---
```

Types of Constructor
1. Default
2. Parameterized
3. Copy

If name of class and function is same then function is called constructor.
As soon as object of class is created constructors are called.
Constructors do not have return type.
Constructor can not be inherited.
It can't be virtual and is always declared in public.

Constructors in which no parameters is passed is called default constructor.

Constructors in which parameters are passed, they are parameterized constructor.

```
class Test
{
    int x, y;
    public:
        Test ( )
        {
            x = 0;
            y = 0;
        }

        Test (int m, int n)
        {
            x = m;
            y = n;
        }

        void display ( )
        {
            cout << x << y;
        }
};
```

```
main ( )
{
  Test T ;
  Test T1 (10, 20);
  Test T2 (20, 30);
  Test T3 ( );

  T1 . display ( );
  T2 . display ( );
  T3 . display ( );
  T3 . display ( );
}
```

```
class complex
{
  double real, img ;
  public :
    complex ( )
    {
      real = img = 0;
    }
    complex (double);
    complex (double, double);
    friend complex sum (complex, complex)
    void display ( );
};
```

```
complex :: complex (double x, double y)
{
    real = x;
    img = y;
}

complex :: complex (double m)
{
    real = img = m;
}

complex sum (complex C1, complex C2)
{
    complex C3;
    C3.real = C1.real + C2.real;
    C3.img = C1.img + C2.img;
    return C3;
}

void complex :: display ( )
{
    cout << real << img << endl;
}
```

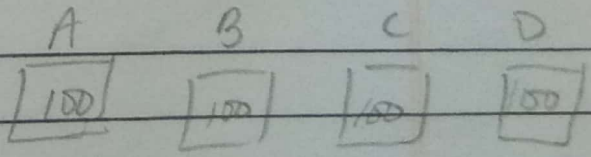
PTO

```
main()
{
  complex A(1.6, 2.4)        // Implicit calling
  complex B = complex(3.9)   // Explicit calling

  complex C;

  C = sum(A, B)

  A.display();
  B.display();
  C.display();
}
```

21/9/18

Copy Constructor

```
class code
{
    int id;
    public:
        code ()                          // Default constructor
        {
        }
        code (int a)                     // Parameterized constructor
        {
            id = a;
        }
        code (code &x)                   // Copy constructor
        {
            id = x.id;
        }
        void show ()
        {
            cout << id;
        }
};

main ()
{
    code A(100);
    code B = (A);
    code C = A;
    code D;
```



```
D = A
A.show()
B.show()
C.show()
D.show()
}
```

Deconstructor

```
int count = 0;
class test
{
  public:
    test ( )
    {
      count ++;
      cout << "No of objects created "<<count;
    }
    ~test ()
    {
      cout <<"No of digits destroyed"< count;
      count --;
    }
}

main ( )
{
  test t1, t2, t3, t4
    {
      test t5;
    }
    { test t6;
    }
}
```

Operator Overloading

1. Unary
2. Binary

```
returntype classname :: operator op (arglist)
{
   fn" body
    —"—
}
```

```
class space
{
int x, y, z;
public:
   void getdata (int a, int b, int c)
   {
      x = a;
      y = b;
      z = c;
   }
   void putdata ()
   {
      cout << x << y << z;
   }
} void operator - ();
```

```
void space :: operator -()
{
    x = -x;
    y = -y;
    z = -z;
}

main()
{
    space s;
    s.getdata(10, -20, 30);
    s.putdata();
    -s;          // activate operator -()
    s.putdata();
}
```

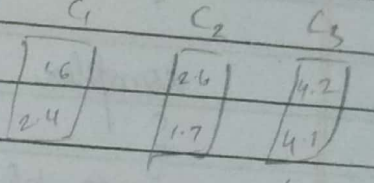
If f" is non-friend (normal) then no of arguments req^d are actual no. of arguments -1

Unary friend = 1 arg
Unary nonfriend = 0 arg
Binary friend = 2 arg
Binary nonfriend = 1 arg

Binary Operator Overloading

```
Class complex
{
   float real, img;
   public:
      complex ( )
      {
      }
      complex (float x, float y)
      {
       real = x;
       img = y;
      }
      void display ( )
      {
        cout << real << img;
      }
      complex operator + (complex);
}

complex complex :: operator + (complex c)
{
    complex temp;
    temp.real = real + c.real;
    temp.img  = img + c.img;
    return temp;
}
```

```
main ()
{
  complex C1 (1.6, 2.4);
  complex C2 (2.6, 1.7);
  complex C3;
  C3 = C1 + C2;
  C1.display ()
  C2.display ()
  C3.display ()
}
```



$C_3 = C_1 + C_2$

$C_3 = C_1 \cdot operator+(C_2)$

$\Downarrow$

temp.real = real + C.real

$C_1 \cdot operator+(C_2)$

```
class complex
{
  float real, img;
  public:
    complex ()
    {
    }
    complex (float x, float y)
    {
      real = x;
      img = y;
    }
    void display ()
    {
      cout << real << img;
    }
```

```
    friend complex operator + (complex, complex);
};

complex operator + (complex a, complex b)
{
    complex temp;
    temp.real = a.real + b.real
    temp.img = a.img + b.img
    return temp;
}

main()
{
    complex C1(1.6, 2.4);
    complex C2(2.6, 1.7);
    complex C3;
    C3 = C1 + C2;
    C1.display();
    C2.display();
    C3.display();
}
```

```
const int size = 3;
class vector
{
    int v[size];
  public:
    vector();
    vector(int *x);
    friend vector operator *(int a, vector b);
    friend istream & operator >> (istream &, vector &);
    friend ostream & operator << (ostream &, vector &);
};

vector :: vector()
{
    for(int i=0; i<size; i++)
        v[i] = 0;
}

vector :: vector(int *x)
{
    for(int i=0; i<size; i++)
        v[i] = x[i]
}

vector operator *(int a, vector b)
{
    vector c;
```

```
    for (int i=0; i<size; i++)
        c.v[i] = a * b.v[i];
    return c;
}

istream & operator >> (istream & din, vector &b)
{
    for (int i=0; i<size; i++)
        din >> b.v[i];
    return din;
}

ostream & operator << (ostream & dout, vector &b)
{
    for (int i=0; i<size; i++)
        dout << b.v[i];
    return dout;
}

int x[size] = {2,4,6};
main()
{
    vector m;
    vector n = x;

    cout << "Enter elements of vector m:";
    cin >> m;  // invoke operator >> function
    cout << m;  // invoke operator << function
    vector p;
    p = 2 * m;  // invoke operator * function
    cout << p;  // invoke operator << function
}
```

Data Conversion

```
class invent2;
class invent1;
{
    int code;
    int items;
    float price;
    public:
        invent1 (int a, int b, float c)
        {
            code = a;
            items = b;
            price = c;
        }

        void putdata ( )
        {
            cout << code << items << price;
        }

        int getcode ( )
        {
            return code;
        }

        int getitems ( )
        {
            return items;
        }
```

```
float getprice ( )
{
return price;
}

operator float ( )
{
return (items * price);
}

};

class invent 2
{
int code;
float value;
public:
invent 2 ( )
{
code = value = 0;
}

invent 2 (int x, float y)
{
code = x;
value = y;
}
```

```
void putdata ()
{
    cout << code << value;
}
invent2 (invent1 p)
{
    code = p.getcode ();
    value = p.getitems * p.getprice;
}
};

main ()
{
    invent1 s1 (100, 5, 140.0);
    invent2 d1;
    float total;

    total = s1;    // invent1 to float:
                   class to basic
    d1 = s1;       // invent1 to invent2:
                   class to class.
    s1.putdata ();
    cout << total;
    d1.putdata ();
}
```

Types of Conversion

1. Basic to Class
2. Class to Basic
3. Class to Class

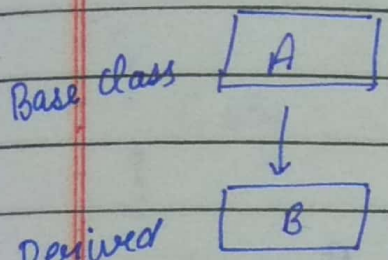
:: SRO
(Scope Resolution Operator)

Date: / / Page no:

Inheritance

Syntax :

```
class derived class : visibility mode base class
{
    class body ;
}
```

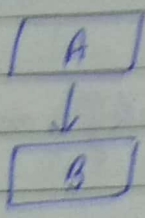



```
Class A
{
};

Class B : public A
{
};
```

Types of Inheritance

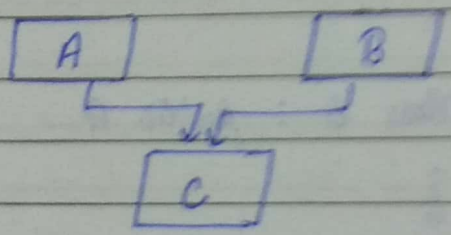
- Single Level inheritance
- Multiple inheritance
- Multi level inheritance
- Heirarchical inheritance
- Hybrid inheritance

1. Simple inheritance



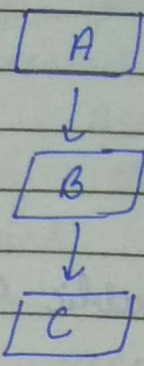

```
class A
{
};

class B : public A
{
};

main()
{
  B obj;
}
```

2. Multiple inheritance



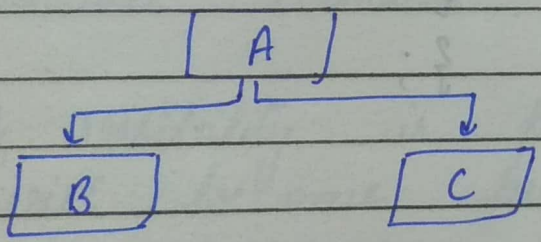

```
class A
{
};

class B
{
};

main()
{
  C obj;
}

class C : public A, B
{
};
```

3. Multilevel inheritance



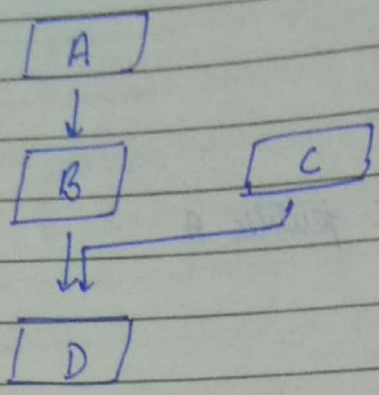

```
Class A
{
};
Class B : public A
{
};
Class C ; public B
{
};
main ( )
{
  C obj ;
}
```

4. Heirarchical inheritance




```
Class A
{
};
Class B : public A
{
};
Class C : public A
{
};
main ( )
{
  B obj ;
  C obj1 ;
}
```

5. Hybrid Inheritance




```
Class A
{
};

Class B : public A
{
};

class C
{
};

class D ; public B, C
{
};

main ()
{
  D obj;
}
```

Visibility Mode
- → Public
- → Private
- → Protected

If visibility of base class is public then all the variables of Base class become public for derived class, (except private variables).

Sir

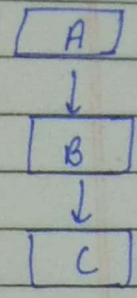
Case Study

Access Specifier

If a datamember is in A class and is of public access then it can be accessed anywhere in the program

If access is private then only the particular class in which the variable is declared can access it.

If access is protected only the child class along with the parent class can access the variable. B can access variables of A but C can't.



Similarly, if visibility is protected then all variables of base class, except private variable will become protected.

If visibility is private, then all variables will become private for derived class.

```
class A
{
};
class B: private A
{
};
class C : protected B.
{
}
```

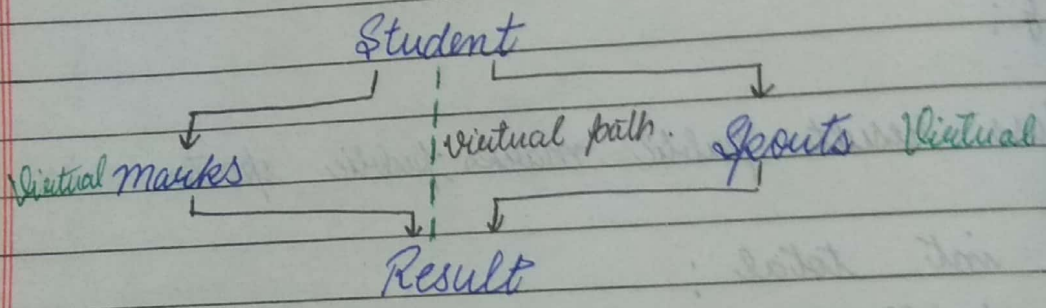
Source Code

```
class student
{
  int roll;
  public:
    void getroll()
    { cin >> roll;
    }

    void putroll()
    {
      cout << roll;
    }
};

class marks : public virtual student
{
  protected:
    int s1, s2;
  public:
    void getmarks()
    {
      cin >> s1 >> s2;
    }
    void putmarks()
    {
      cout << s1 << s2;
    }
}
```

```
class sports : public virtual student
{
  protected:
    int score;
  public:
    void getscore()
    {
      cin >> score;
    }
    void putscore()
    {
      cout << score;
    }
};

class result : public marks, public sports
{
  int total;
  public:
    void display()
    {
      putrolls();
      putmarks();
      putscore();
      total = S1 + S2 + score;
      cout << total;
    }
};
```

```
main()
{
  result R;
  R.getroll();
  R.getmarks();
  R.getscore();
  R.display();
}
```

S_1, S_2, score are protected as they are accessed in result class, ie, sub-derived class.



Class student can directly switch to result class through virtual function.

overriding :- Same f" name & arguments.
overloading :- Same f" name & diff arguments.

virtual base class for every class is student.

Result has 2 paths from result → student from sports & marks, so to avoid ambiguity we use a keyword virtual.

Class & object

```
class person
{
    char name[30];
    int age;
    public:
        void getdata()
        {
            cin >> name >> age;
        }
        void putdata()
        {
            cout << name << age;
        }
};
```

Obj = normal

Obj = array
array of obj

Obj = pointer
pointer to obj

Virtual Function

```
class base
{
  public:
virtual void display ()
    {
      cout << "Display base";
    }

    void show ()
    {
      cout << "Show base";
    }
};

class derived : public base
{
  public:
    void show()
    {
      cout << "Show derived";
    }

    void display ()
    {
      cout << "display derived";
    }
};
```

Pointer always points to its class. even if it contains the reference of any other class. If we want to call f" of class whose reference is in pointer then we will use virtual function.

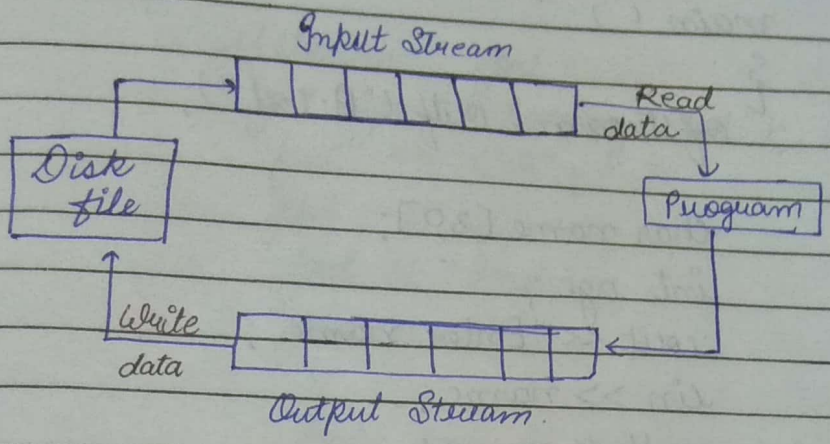
For f" overriding, inheritance is required. and virtual f" is also needed.

```
main ()
{
    base *bptr;
    base B;
    bptr = &B;
    bptr → show ();
    bptr → display ();

    derived D;
    bptr = &D;
    bptr → show ();
    bptr → display ();
}
```

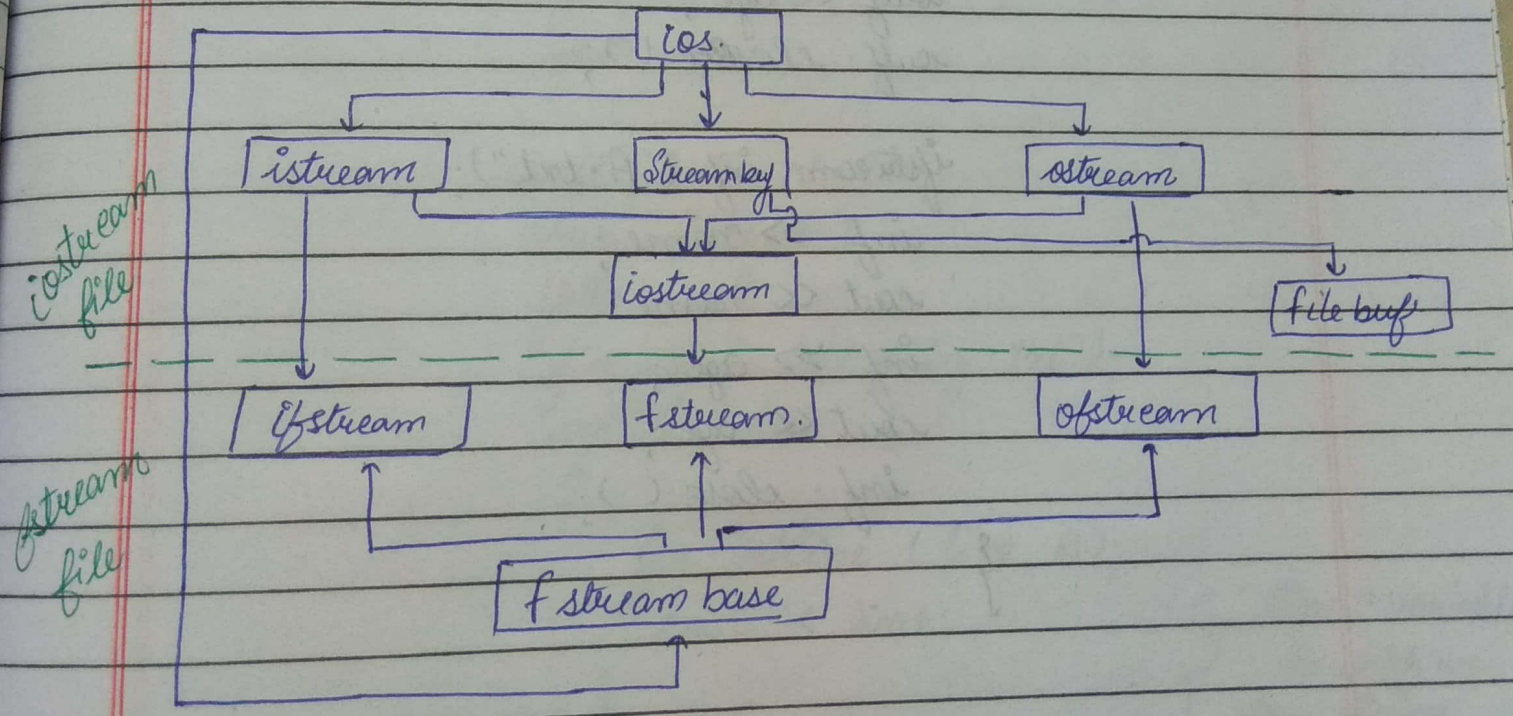
20/11/18

Date: / / Page no:

FILE HANDLING.



File input / output Stream



Stream classes for file operations

```
#include <iostream.h>
#include <fstream.h>
main()
{
ofstream outf("A.txt");

char name[30];
int age;
cout << "Enter name";
cin >> name;
outf << name;
cout << "Enter age";
cin >> age;
outf << age;
outf.close();

ifstream inf("A.txt");
inf >> name;
cout << name
inf >> age;
cout << age;
inf.close();
}
```

```
main ()
{
    ofstream fout;
    fout.open ("country.txt");
    fout << "USA";
    fout << "UK";
    fout << "India";
    fout.close ();

    fout.open ("Capital.txt");
    fout << "Washington";
    fout << "London";
    fout << "Delhi";
    fout.close ();

const int N=80;
char line[N]
ifstream fin;
fin.open ("country.txt");
while (fin)
{
    fin.getline (line, N);
    cout << line;
}
fin.close ();
fin.open ("Capital");
while (fin)
{
    fin.getline (line, N);
} cout << line;
fin.close ();
```

line → variable in which we store string
N → length of string.

File modes

i) ios :: app → Append to end of file
ii) ios :: ate → Go to end of file on opening
iii) ios :: binary → Binary file
iv) ios :: in → open file for reading only
v) ios :: out → open file for writing only
vi) ios :: nocreate → open fails if file does not exist
vii) ios :: noreplace → open fails if file already exists
viii) ios :: trunc → Delete content of file if it exists

File pointers

i) seekg () → moves get pointer (input) to specified location
ii) seekp () → moves put pointer (output) to specified location.
iii) tellg () → Gives current position of get pointer
iv) tellp () → Gives current position of put pointer.

```
main( )
{
  char string [80];
  cout << "Enter string";
  cin >> string;
  int len = strlen (string);
  fstream file;

  file.open ("Test.txt", ios::in | ios::out);
```

to verify data is

```
for(i=0; i< len ; i++)
{
    file.put (string[i]);
}

file.seekg(0);
char ch;
while (file)
{
  file.get (ch);
  cout << ch;
  }
  file.close();
}
```

23/11/18

I/O Operation on Binary Files

```
main( )
{
    float height[4] = {175.5, 192.6, 187.6, 139.5};
    ofstream outfile;
    outfile.open ("A.txt");
    outfile.write ((char *) &height, sizeof(height));
    outfile.close( );

    for (int i=0; i<4; i++)
        height[i]=0;
```

to verify that data is fetched from file

```
ifstream infile;
infile.open("A.txt");
infile.read((char*)&height, sizeof(height));

for(i=0; i<4; i++)
cout<<height[i];

infile.close();
}
```

NOTE

infile.read((char*)&v, sizeof(v));
outfile.write((char*)&v, sizeof(v));

compulsory: Type casting variable add to char pointer

```
class inventory
{
  char name[10];
  int code;
  float cost;

public:
  void readdata()
  {
    cin>>name>>code>>cost;
  }
```

```
void write_data()
{
    cout << name << code << cost;
};

main()
{
    inventory items[3];
    fstream file;

    file.open("stock", ios::in | ios::out);

    for(i=0; i<3; i++)
    { items[i].readdata();
      file.write((char*)&items[i], sizeof(item[i]));
    }

    file.seekg(0);

    for(i=0; i<3; i++)
    {
      file.read((char*)& items[i], sizeof(items[i]));
      items[i].writedata();
    }

    file.close();
}
```

copy content from source to dest file.

27/11/18

Templates

Class Template

```
template <class T>
class xyz
{
  Data members
  with type T
  ---------
};
```

Templates are used for generic programming, ie datatype of variable is decided during execution.

```
template <class T1, class T2>
class sum
{
  T1 x;
  T2 y;
public:
    sum(T1 a, T2 b)
    {
      x = a;
      y = b;
    }
    void display()
    {
      cout << x << y;
    }
};
```

```
main ( )
{
    sum < int , double > obj (10, 2.5);
    sum < double , double > obj1 (2.4, 3.6);
    sum < int , int > obj2 (1, 2);
     obj1 . display ( );
     obj2 . display ( );
     obj3 . display ( );
}
```

Function Template

```
template < class T >
void swap ( T &x, T &y)
{
    T temp = x;
     x = y;
     y = temp;
}

void fun (int m, int n, float a, float b)
{
    swap (m, n);
    cout << m << n;
    swap (a, b);
    cout << a << b;
}

main ( )
{
    fun (100, 200, 2.4, 5.6);
}
```

```
template <class T1, class T2>
void display (T1 x, T2 y)
{
  cout << x << y;
}

main()
{
  display ("ABC", 199);
  display (1000, 9.4);
}

template <class T>
void display (T x)
{
   cout << x;
}

template <class T1, class T2>
display (T1 x, T2 y)
{
  cout << x << y;
}

void display (int x)
{
  cout << x;
}
```

```
main ()
{
    display (100);
    display (12.34);
    display (100, 12.34);
    display ('A');
    display (1000);
}
```

Exception Handling

try block:
→ detects & throws an exception.

Exception object

catch block
→ catches & ~~throws~~ handles an exception.

Exception Handling Mechanism

```
try
{
  —
  —
  throw exception;
  —
}—
catch ( type arg)
{ —
  —
}—
```

```
main()
{
  int a, b;
  cin >> a, b;
  int x = a-b;
  try
  {
    if (x != 0)
      cout << a/x;
    else
      throw (x);
  }

  catch (int i)
  {
    cout << "exception caught" << i
  }
}

void divide (int x, int y, int z)
{
  if ((x-y) != 0)
  { int R = Z/x-y;
  } cout << R;

  else
    throw (x-y);
}
```

Eg2:

```
main()
{
    try
    {
        divide(10,20,30);
        divide(10,10,20);
    }
    catch(int i)
    {
        cout << "Exception caught";
    }
}
```

Multiple Catch Statements

```
void test(int x)
{
    try
    {
        if(x==1)
            throw x;
        else if(x==0);
            throw 'x';
        else if(x==-1);
            throw 1.0;
    }
    catch(char a)
    {
        cout << "Char caught";
    }
```

```
catch (int b)
  {
    cout << "Integer caught";
  }

catch (double c)
  {
    cout << "double caught";
  }
}

main()
{
  test(1);
  test(0);
  test(-1);
  test(-2);
```

- To use single catch for multiple try or throw

```
catch (...)
{
  cout << "Exception caught";
}
```

Rethrowing an Exception

```
void divide (double x, double y)
{
    try
    {
        if (y==0.0)
            throw 'y';
        else
            cout << x/y;
    }

    catch (double a)
    {
        cout << "double caught in fn.";
    } throw;

}

main ()
{
  try
  {
    divide (10.0, 5.0);
    divide (10.0, 0.0);
  }

  Catch (double)
  {
    cout << "caught double in main";
}}
```