Input - output

printf (" Hello ");. → cout << "Hello."

printf (" %d ", a); cout << a

printf (" sum is %d ", a); cout << "sum is" << a

scanf (" %d %d", &a, &b) → cin >> a >> b.

scanf (" %d %f", &a, &b) cin >> a >> b.

header file for cout << & cin >> is

iostream.h

\n → endl

Program

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

void main ()
{
    clrscr ();
    int x;
    cout << " Enter a number " << endl;
    cin >> x;
    int s = x * x.
    cout << " square of " << x << "is" << s;
    getch ()

}
```

DATE

Reference Variables.

int &y = x ; (it is NOT address var; it is reference var).

- It is necessary to initialize at the time of declaration.
- It is actually an internal pointer
- It is as if y is another name of x (just to understand)
- It can be initialized with already declared variables.
- Reference variable can not be updated.

Functions.

```
# include <iostream.h>

Void main ()
{
    Void fun() ;          ⟶ Declaration
    cout << "you are in main";
    fun ();               ⟶ Call
}
 Void fun ()
{
   Cout << "You are in fun" ;
}
```

Defination.

DATE

- Inline function
syntax:

inline void fun (); ⟶ (at the time of declaration)

- Function overloading
  - More than one functions can have the same name in program

```
# include <conioh>
# include <iostream>
int area (int, int);
float area (int);
void main ()
{ void
  clrscr ();
  int r.
  cout << " Enter radius of circle" ;
  cin >> r ;
  float A = area (r);
  cout << " \n Area of circle is " << A ;
  int l, b, a ;
  cout << " Enter length & breadht of rectangle " ;
  cin >> l . >> b ;
  a = area ( l, b);
  cout << \n Area of rect is " << a ;

}
float area (int x)
{ return ( π*x*x) ;
}
```

DATE

```
int area (int x, int y)
{
    return (x * y);
}
```

Note : Both functions have same job for diff parameter that is why they can have same names.

Structures.

ex:

```
struct book
{
    int bookid;
    char title[20];
    float price;
};

void main ()
{
    book b1, b2, b3;
    b1.bookid = 101;
    b1.title = "c++" ← X wrong
    strcpy (b1.title, "c++");
    b1.price = 300.0;

                OR

    book b1 = { 101, "c++", 450.0 };
```

No need to add struct before in c++

Declaring

```
b2.bookid = b1.bookid
b2.title  = b1.title
b2.price  = b1.price
```

copying data from b1 to b2

OR

```
b2 = b1;
}
```

- taking input from user. (prev book example).

```
void main ()
{
   book b1;
   cout << " Enter bookid, title & price " ;
   cin >> b1.bookid >> b1.title >> b1.price;
}
```

OR

```
void main ()
{
   book b1;
   b1 = input ();
}

book input ()
{
   book b;
   (same code as above)
   return(b);
}
```

DATE

- encapsulation in struct.

→ In C++ we can define functions inside structure

e.g

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

struct book
{
    int bookid ;
    char title[20];
    float price ;
    void input ()
    {
        cout << " Enter bookid, title & price " ;
        cin >> bookid >> title >> price ;
    }
} ;

void main ()
{
  book b1 ;
  b1.input () ;
}
```

DATE

Classes and objects.

- In structure all members are by default public where as in classes all members are by default private.

- Class is same as struct. Just members are private & variables are called as objects

Static Members

```
Void fun ()
{
    Static int x;   → by default 0
                    → memory throughout the program
    int y;          → memory when fun is called only
                    → Garbage value.
}
```

Static local variable

- Static member variable.
- Declared inside class body.
- Known as class member variable.
- Defined outside the class.
- They do not belong to any object but to whole class
- They can also be used with class name.

DATE

Example

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

class account
{
 private :
   int balance;          instance member variable
   static float roi;     static member variable / class member
 public :
   void setbalance (int b)
   { balance = b ; }

} ;
```

member label

```
float account :: roi = 3.5f ;    Declaration of 'roi'

void main ()
{
  account a1, a2 ;
}
```

Both a_1 & a_2 will have 'balance' variable but not 'roi'

'roi' is defined same for all variables

DATE

case 1: If 'roi' is not private i.e. it is public.

1) ~~Variable~~ Objects a_1 & a_2 are declared.

→ we can access 'roi' as.

```
a1.roi = 3.5;
```

2) No objects.

→

```
account :: roi = 4.5;
```

case 2: If private.

1) with objects.

make a function inside the class (public) to access roi.

i.e.

```
void setroi (float r)
{ roi = r; }
```

call in the program as

```
a1.setroi(4.5);
```

2) without objects

make the above function static.

DATE

Constructor

- member function of class.
- name is same as class.
- No return type.
- It can never be static

• Calling constructor

- we don't call, it gets called when obj is created
- Used to solve problem of initialisation

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

class complex
{
private :
   int a, b;
public :
   complex (int x, int y)                    ①
   { a=x ; b=y ; }
   complex (int k)                           ②
   { a=k }
   complex ()                                ③
   { a=0 ; b=0 }

};
void main ()
{
   complex c1(3,4), c2(5), c3 ;

}
```

① ② } parameterized const

③ default const
↳ complex ()
{ }

① ② ③

DATE

Imp cases

1) You have introduced NO CONSTRUCTOR

→ Compiler will make Default + Copy constr.

2) You have introduced Simple Constr.

→ Compiler will make only copy const

3) You have introduced Simple + Copy

→ Compiler will make No constructor

• eg of copy const.

```
complex (complex &c)
{
    a = c.a;
    b = c.b;
}
```

DATE

Destructors

- instance member function
- name is same as class with ~ symbol
- Never static
- No return type
- No argument. (No overloading)

```
# include <iostream.h>

class complex
{   private :
        int a, b ;
    public :
        ~complex ()
        { }
};
```

- It should be defined to release resources allocated to an object.

DATE

Friend function

- It is not a member function of a class to which it is a friend.
- Declared in the class with friend keyword.
- Defined outside the class
- Can access any member of friend class.
- Can not access directly →
- No caller object
- It should not be defined with membership label

Ex:

```
# include <iostream.h>

class complex
{
  private:
    int a, b;
  public:
    void setdata (int x, int y)
    { a = x; b = y; }
    void showdata ()
    { cout << a << b; }

    friend void fun (complex);      - Declaration
};

void fun (complex c)                — Definition
{
  cout << "sum is" << c.a + c.b;
}
```

DATE

```
void main ()
{
    complex c1, c2, c3;
    c1.setdata (3,4);
    fun (c1);
}
```

calling

- Friend function can become friend to more than on class.

- Operator overloading in friend function. (example below)

```
# include <iostream.h>

class complex
{ private:
    int a, b;
  public:
    void setdata (int x, int y)
    { a = x ; b = y;
    void showdata ()
    { cout << a << b ; }

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

complex operator + (complex X, complex Y)
{ complex temp;
    temp.a = X.a + Y.a;
    temp.b = X.b + Y.b;
```

DATE

```
    return (temp);
}

void main()
{
 clrscr()
 compler c1,c2,c3;
 c1.setdata(3,4);
 c2.setdata(5,6);
 c3 = c1+c2;          // c3 = operator + (c1,c2)
 c3.showdata();
 getch();
}
```

- over loading of insertion and extraction ~~function~~ operators

```
# include <iostream>
# include <conio.h>
 class complex
{
  private:
    int a,b;
  public:
    void setdata(int x, int y)
    { a=x; b=y; }
    void showdata ~~(void)~~ ()
    { cout << a << b; }
  friend ostream & operator << (ostream &, complex);
  friend istream & operator >> (istrem &, complex &);
};
```


PAGE

DATE

```
ostream& operator << ( ostream &dout , complex c)
{
   cout << c.a << c.b ;
   return dout ;
}
 istream & operator >> ( istream &din , complex &c)
{
   cin >>  c.a >> c.b ;
   return din ;
}

void main ()
{
   clrscr () ;
   complex c1 ;
   cin >> c1 ;          // operator >> ( cin , c1 )
   cout << c1 ;         // operator << ( cout , c1 )
   getch() ;

}.
```

- Member function of one class can become friend to another class.

DATE

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

class A
{
      void fun()
       {. . . . .}
      Void foo()
       {. . . . . .}
      Void boo()
       {       }
}
Class B
{
      friend void A::fun();
      friend void A::foo();
}

Class C
{
      friend class A;
}
```

All functions in class A

DATE

Inheritance in C++.

- It is a process of inheriting properties and behaviours of existing class into a new class.

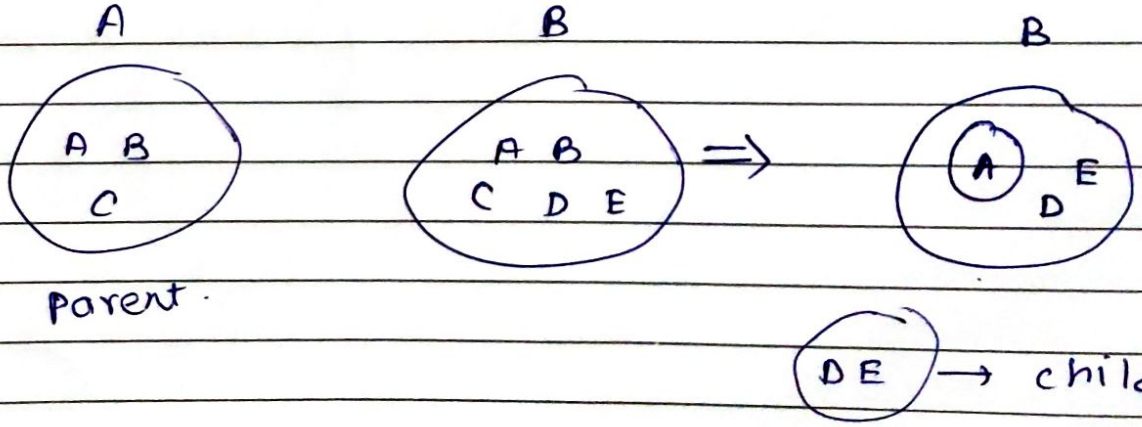
- Old class → parent / base class
  New class → child / derived class

- Syntax.

```
Class A
{
};
Class B : visibility mode A.
{
};
```

A → Parent B → Child

Visibility mode → private / public / protected.



Parent.

DE → child.

DATE

Types of inheritance

1) single $A \rightarrow B$

```
Class A
{ };
Class B : public A
{ };
```

2) multilevel $A \rightarrow B \rightarrow C$

```
Class A
{ };
Class B : public A
{ };
Class C : public B
{ };
```

3) multiple $A_1, A_2 \rightarrow B$

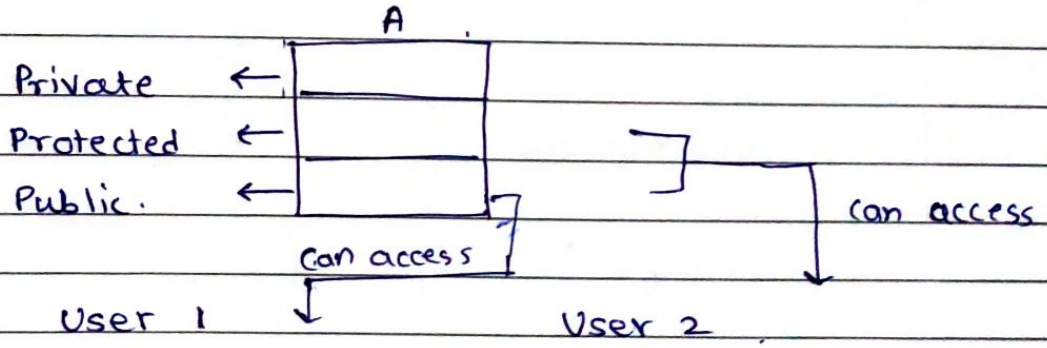
```
Class A1
{ };
Class A2
{ };
Class B : public A1, public A2
{ };
```

4) Hirarchial $A \rightarrow B_1, B_2$

```
Class A
{ };
Class B1 : public A
{ };
Class B2 : public A
{ };
```

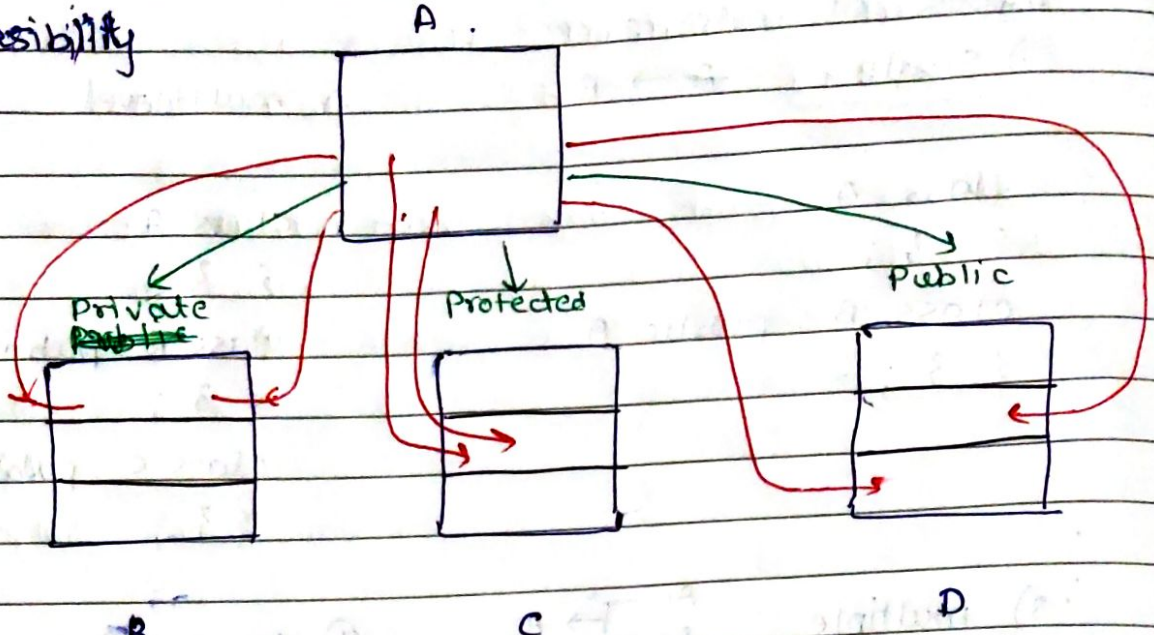
5) Hybrid - (in next lectures)

visibility modes



DATE

Accessibility



is-a relation ship.

Child ← Banana is a fruit → Parent class

two wheeler is a vehical

- "is a" relationship is always implemented as a public inheritance

DATE

Constructors in inheritance.

In inheritance, when we create object of derived / child class, what will happen ?

Ans: Constructors from both classes will get called.

object $\xrightarrow{\text{calls}}$ child constr. $\xrightarrow{\text{calls}}$ parent constr.

Calling order of constructor : B → A

execution " : A → B

Syntax :

```
class A
{ public :
    4A()                    } Default
    5{ }
};

Class B : public A
{
  public :
  B() : A()                 } Default
    2 3
  { }
};

Void main()
{  B obj ;
}
```

DATE

Note: If we create constructor in A but not in B → ERROR.

In that case you have create contructor in B and also have to call A.

example

```
# include <iostream>

 Class A
{    int a;
   public:
     A (int k)
     { a = k; }
};

 Class B
{     int b;
   public
     B(int x, int y) : A (x)
     { b=y; }
};

 Void  main ()
{
   B obj (2, 3);
}
```

Output a = 2 b = 3.

DATE

Destructor in inheritance

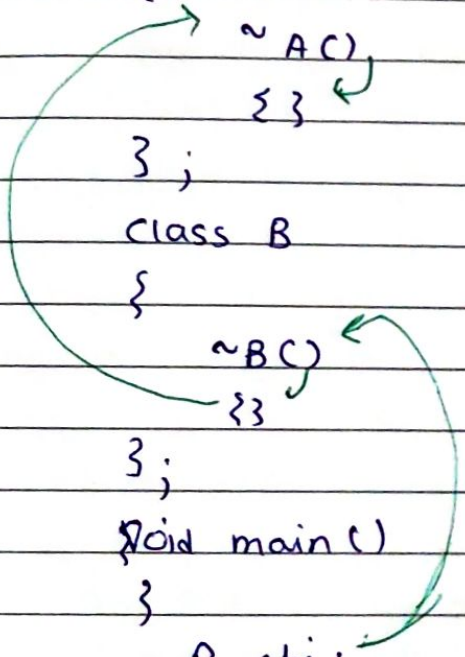
Obj $\xrightarrow{\text{calls}}$ child $\xrightarrow{\text{calls}}$ parent

calling : B → A

*execution : B → A

Syntax :

```
class A
{
    ~A()
    {}
};
class B
{
    ~B()
    {}
};
void main()
{
    B obj;
}
```



No need to call parent in child class. Compiler will do it automatically.

DATE

Function overriding & function hiding.

```
#include <iostream.h>
Class A
{   public:
        void   f1 ()     { }
        void   f2 ()     { }
}

Class  B
{     public
        void   f1 ()     {}        overriding
        void   f2 (int x) {}       hiding.
}

int  main ()
{
    B  obj ;
    obj. f1 () ;      // class B wala call hoga
    obj. f2 () ;      // Error ∵ Same name so won't go to A
    Obj. f2 (4) ;     // class B

}
```

arg B need ✗