29/7/2019

Programming Fundamentals & OOP concepts.

SKS sir.

Low Level Language

⇒ Machine Language ⇒ 0s & 1s.

- Error prone
- Debugging is difficult
- Machine Dependent (Not portable).
- Tedious

⇒ Assembly Level prog. → Not portable but Easier.
- Pseudo-code
- Mnemonics (Alphanumeric)

Assembly Level
Assembler ← Translator
Machine Lang.

High Level Language → Machine-dependent object code (Source code can be copied to any machine)

⇓ → Portable

FORTRAN, BASIC, C, .., etc ⇒ English Like.

GRAMMAR ⇒ SET OF RULES

Translator
- M/C code
- Interpretor
- Compiler

Source Program
↓ Translator
Object code

Interpretor → Takes a Line at a time from source code and translates it & execute it; stops if no line is left or encounters any error.

Error → Rectify & start again from first line.

Compiler → Translates the whole program in a single shot. If no error, creates object code else displays a list of error.

Programming Approach

⇓

Paradigm ⇓

1. Data
2. Algorithm

System is developed

Modify | ⇓ Changes to be made → Organization of the data.

→ Finding efficient algorithm to work on data

Problem

⇓

Sub-tasks → Write the procedure/fn for the sub-problem

⇓

to focus on the algorithm

→ MODULAR PROGRAM ⇓

→ RELATED PROCEDUREs and data on which they work are put together to form a module.

→ Organisation of data is ignored

Data Abstraction

⇓

Access control

Object Oriented Approach

30/7/2019

Programming Fundamentals

SKS sir.

int x ;
↑
datatype → Interpretation
→ Space requirements
→ ~~Observation~~ operations allowed
→ Range of values

int x = 65 ;
↓
binary

char C = 'A' ; → Stream of 0/1 depends on datatype.
↓
ASCII value → binary

COBOL → Common Business oriented Language.

(Assume 2 bytes).
Byte Addressable memory.

p is pointing to x.

Space allocation is same.

int x ;
int *p ;

When we don't know the value in garbage

can hold the address of another integer variable.

Segments
Core Data Stack

Segmentation Fault
Accessing memory not allocated.

p = &x ;
↑
address of x.

scanf("%d", &x);
↑
if & is not provided, it will not give any error.

*p = (content of location to which p points).

*p = x

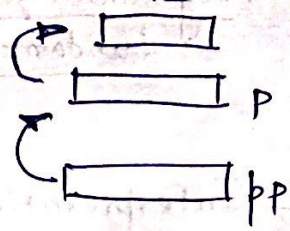
int x; int *p ;
↓
type of variable to which p will point.

int * *pp ;
↑ Type to which it is pointing

pp = &p

$p = \&x$

$pp = \&p$



$\Rightarrow p \equiv \&x$

$\Rightarrow *p \equiv x.$

$\Rightarrow pp \equiv \&p$

$\Rightarrow *pp \equiv *p \equiv \&x.$

$\Rightarrow *(*pp) \equiv *p \equiv x.$

~~Avg~~ ~~Sum~~ Avg of n numbers.

↳ while loop.

```
char ch = 'y'; int sum = 0; int cnt = 0;
  while (ch == 'y' || ch == 'Y').
  {
      scanf("%d", &x);
      sum = sum + x;
      cnt = cnt + 1;
  }.
~~double~~ float avg = float(sum) / float(cnt);
                      (either one).
```

Mixed type
↓
More General type.

→ Elements in a single variable & be able to refer them → variable name

```
int x[100];
```

int → type of the elements.

100 → size.

x → integer pointer.

#define LENGTH 100

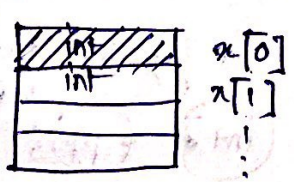
↑ Preprocessor directive.

Collection of homogeneous elements

↳ Array

A block of memory of contiguous space will be allocated to hold 100 integers.

name points to 1st element



```
char s1[30], s2[80];
scanf("%s", s1);
```

strcpy(s2, s1).

s2 = s1

⇓

assigning one pointer to another

however they are constant pointer

int const *x ;

- int → type to which the pointer is pointing to.
- const * x → constant pointer

```
for (i=0; i<n; i++)
{
    scanf("%d", &x[i]);
}
for (i=0; i<n; i++)
{
    printf("%d", x[i]);
}
```

x → x[0]

x + i → x[i]

⇒ &x[i] can be written as x + i

x ⟶ x[0].

p ⇒ pointer.

p + 1 ⟶ 500 → 502.

int pointer.

char pointer ⟶ 500 → 501

p + K → pointer to Kth variable.

⇓

p + (K * size of T).

T *p

* (x + i) ≡ x[i]

x[i] ≡ * (x + i).

Max 1000 integers

#define LENGTH 1000

Mostly ⇒ working ~~with 100 numbers~~ with 100 numbers.

int x[LENGTH] ;

↳ leads to space wastage.

Dynamic Memory Allocation

x → numbers.

int *x

x →

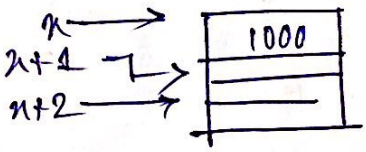
int *x;

x = (int *) malloc (sizeof(int) * n) ;

(void *)

No. of bytes we need → check for Null.

Tries to allocate specified amount of memory, successful- first address. OR null.



Prof Sanjay Kumar Saha

1D Array & Pointer

```
int *p;   p = ___ malloc ___ ;
```

~~p = (int *)~~ ⇓ dynamically allocated memory has to be released once it is not required

Variable

- Scope (part of the code where it is accessible)
- Lifetime (How long the variable exists?)

Releasing :-

```
free(p);
```

frees memory pointer pointing to the allocated memory

2-D Array

```
int x[10][5];
```

Rows, Column

In memory

Stored - 1D

x[0][0] x[0][1] .... x[1][0] ...

2D Array

⇓

A one dimensional array where elements are again 1D array.

x → 1st element

↑ => x → x[0] → x[0][0]

int **

x+1 → x[1]

x[0] + 1 → x[0][1]

x[0][0]

x+i → x[i]

*(x+i) = &x[i][0]

*(x[i]+j) = &x[i][j]

*(x+i) = x[i]

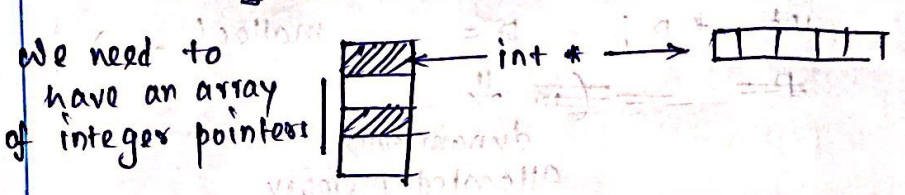
General form

*(x+i)+j = &x[i][j]

[(x+i)+j] = x[i][j]

during
size specified ↑ Execution

No. of ROWS ⟶ (known)
~~No. of column~~




```
int *p[10];
```

{ Memory is saved instead p[10][5]

INPUT r, c
r → no. of rows
c → no. of columns.

```
p[i] = (int *) malloc (c * sizeof(int));
```

```
int **p;
INPUT r, c;
p = (int **) malloc (r * sizeof(int));
for (int i = 0; i < r; i++)
    p[i] = (int *) malloc (c * sizeof(int));
```

Freeing up memory

```
for (int i = 0; i < r; i++)
    free(p[i]);
free(p);
```

PTR to Array

```
int x[10];
int (*p)[10];
```

2byte

p is a pointer to an array of 10 integers.

p = &x

p ⟶ x
1000

int *t = x

t++ ⟶ 1002

p++ ⟶ 1020.

```
int *p[5]
```

5 => rows

```
p = (int**) malloc (r * 5 * sizeof(int));
```

p+i ⟶ i th row.

=> p+1 ⟶ 1st row

Function

large task => smaller tasks.

calling function

Present in stack

```
main()
{
.
.
.
}
```

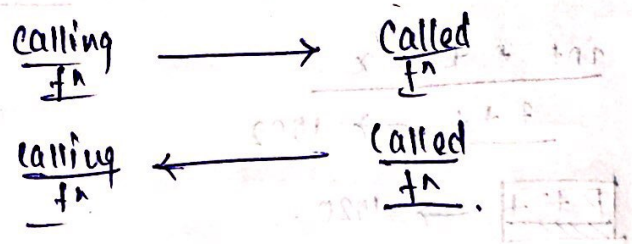
called fn.

1. When a fn is called, execution of a calling fn is temporarily suspended.
2. Control of execution goes to called fn. once execution is over, the control comes back to the calling fn.

④

Function
creates more stress on the system, however modularity is preferred more.

Call by value & Call by Reference




```
int main()
{
    int a, b;
    int area, perimeter;
    calc-parameter(a,b);
}
```

```
void calc-parameter
        (int a, int b)
{
    ...
}
```

→ Use of global variables is not advised

parameter passing

by value — by reference/address.

```
calc-parameter(a, b, &area, &perimeter).
{
    ...
    *area = a * b
    *perimeter = (a+b) * 2
}
```

```
char *c = "abcd" ;
f(c)
....          void f(char *p)
```

Variable where change is to be reflected, send its address.

07/08/2019 Programming Practice SKS sir

Passing array to function

1D Array

```
#define SIZE —

int x[SIZE];
int n;
INPUT n;
getnumbers(x, n);
```

Prototype `void getnumbers(int [], int);` or `int *, int` ⟶ `for(....)` ~~input...~~ . `scanf(....)` `&p[i]` OR ~~**~~ `(p+i)`.

`void printnumbers(~~int~~ x, n);`

2D Array

```
int mat[ROW][COL];

void getnumbers( int **p
                 int p[][COL] , int r, int c)
```

`p+i` gives ith row.

Recursion

Function calling itself.

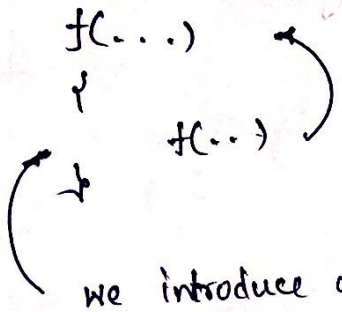
```
main()
{
   ....
   ....
   f(..)
}
```

```
int f(int)
{
   int k;
   k = f(10);
   f(20);
}
```

Not an error.

```
void f(int)
{
   k = f(10);
}
```

↑ Error



Never - ending.

we introduce a condn

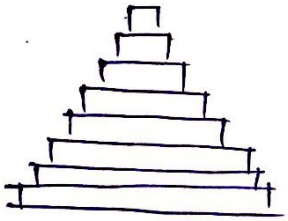
if (..) = True.

f(..)

if (..) = False

break;

Stack



```
int fact(int n)
{
  if (n > 1)
    return n * fact(n-1);
  else
    return 1;
}
```

Recursion may have memory overhead. [Stack overflow]

Ptr to function

```
int *p;      int pointer
int *p[];    array of ptr.
int (*p)[];  ptr to array
int *f();    fn ret. pointer.
int (*f)();  ptr. to fn that returns int.
```

```
int (*pf)(int, char);

int f(int x, char c)
{
  .....
  return(..);
}
```

Alt

```
pf = f;        pf = &f;
*pf (10, 'A')
```

Calling a fn.

but here we can skip it.
& and * can be dropped.

ptr to fn ⇒ has the address of the corresp. code of the fn that it points.

⇓ malloc/dealloc cannot be done.

```
processing (....)
{
   op....
}
```

```
f(1) ...
f(2) ....
  .
  .
f(n) ....
```

ith op.
fi1() ... | ffi()
fi2() ...

```
int (*p[])(int, int)
```

Array to function pointers having 2 integers as arguments

```
p[0] = add;
  .
  .
p[3] = divide.
```

Array

⇒ Collection of homogeneous elements

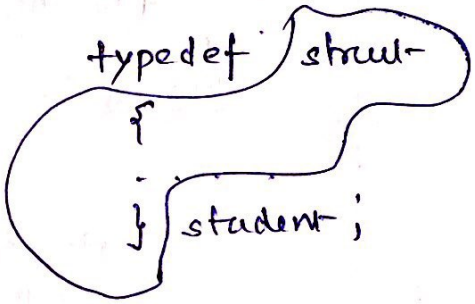
Structure

not homogeneous.

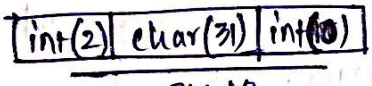
↳ Collection of diff. element - which may not be of same type.

```
struct <tag>
{
    int roll;
    char name[31];
    int marks[5];
} x;
```

```
struct <tag> x;
struct <tag> y;
```



typedef oldtype newtype;



31 + 10 + 2 = 43.

scanf("%d", &x.roll);

x.name, &x.marks[i]

gets(x.name)

p = &x.

p → roll

p → name

&(p → member[i]).

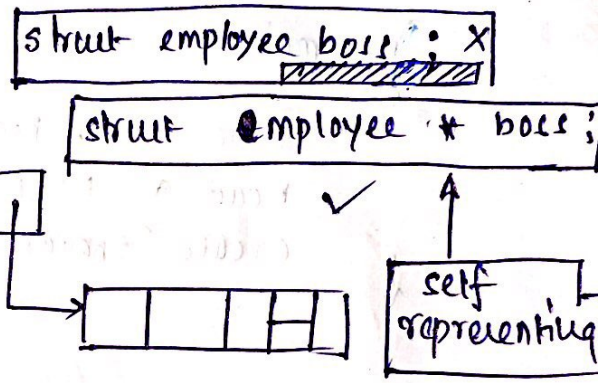
08/8/2019

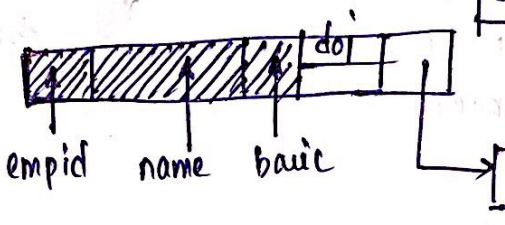
Programming Practice

Prof. Sanjay Kumar Saha

```
struct date
{
    int d, m, y;
};
```

```
struct employee
{
    int empid;
    char name[31];
    float basic;
    struct doj;
};
```



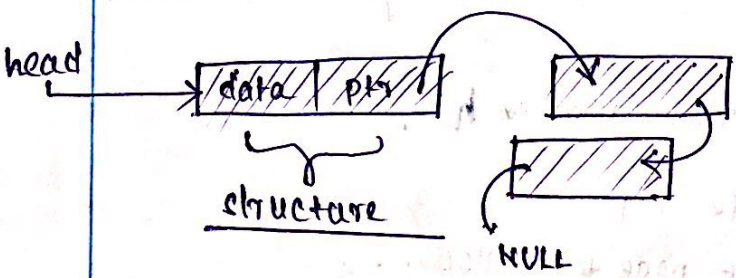


Contiguous Memory Allocation

→ Random Access possible

→ Insertion/Deletion not possible.

Linked List → No contiguous space for all elements required.



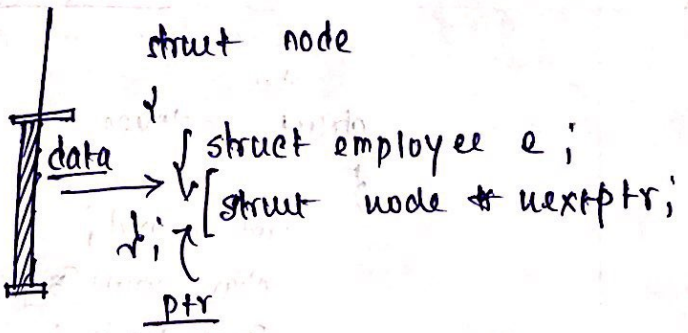
→ Insertion & Deletion easier wrt array.

→ No-predetermined size.

→ Sequential access

→ Overhead:- In each node, a pointer to each node is required.

```
struct employee
{
    int ecode;
    char name[99];
    float basic;
};
```




```
int main()
{
    struct node * head;
    head = NULL;
    create (&head);
}
```

```
void create (struct node **p)
{
    struct node *p;
    p = (struct node *)
        malloc (sizeof(struct
                node));
    p -> nextptr = NULL;
    get_employee_data (&(p->e));
    *hp = p
}
```

```
void get_employee_data ( struct employee * ep)
{
}
```

```
void append ( struct node * h).
{
    struct node * p;
    p = (struct node *) malloc ....
    p -> nextptr = NULL;
    ~~p -> emp~~ get_employee_data (& (p -> e)) ;
    while ( h -> nextptr != NULL)
    {   h = h -> nextptr;
    }
    h -> nextptr = p.
}
```

~~void~~ display - employee - data(h → e)

~~void h;~~ ~~void search (head)~~

search(head, k);

if ((h → e).ecode == k)
{
. . . .
}
if (h == NULL)
{
printf("Not found");
}

~~void~~ insert - before(&head, k)

void insert - before(struct node **hp, int p)
{
struct node *p, *prev, *cur;
cur = *hp;
if ((cur → e).ecode == k)
{
p = ... ; get . . . ;
p → nextptr = ~~cur~~; cur ~~*nextptr~~;
~~head → nextptr =~~ *hp = p;
}
else
{
curr = (*hp) → nextptr;
prev = (*hp);
while (curr != NULL)
{
if ((curr → e).ecode == k)
{ break;
}
else prev = curr;
curr = curr → nextptr;
~~prev~~
}

```
    if (curr == NULL)
    {   printf("Not found");
    }
    else
    {
        p = . . . . . .
    prev ~~curr~~ -> nextptr = p ;
        p -> nextptr = curr ;
    }
}

void delete (struct node *hp, int k).
{       struct node, *p , *prev , * curr;
        curr = hp ;
        if ( (curr->e).ecode == k)
        {   *hp = (*hp) -> nextptr;
        }   free(curr);

        else
        {
            . . . .
            . . .
            prev -> nextptr = curr -> nexptr;
            free(curr);
        }
}
```

File Handling

Array => Large amount of data (IN Primary Memory)
↓
once program is over, everything is lost.

File => Permanent storage of Data.
↓
Data structure (only) with secondary memory.

To work with a file

- => open a file → store data
- => work with file → Access data.
- => close the file

In C

```
FILE *fp;
fp = fopen("mydata", "r");
```

special type of structure (defined in stdout)
↓
stores info. regarding the file

opening mode:
- w => write
- r => read

fopen opens the file in desired mode. If successful, returns file pointer. If it fails, returns NULL.

```
if (fp == NULL)
{
    printf("ERROR");
    exit(-1);
}
```

- ptr to the buffer
- actual pos^n in the buffer
- whether EOF has occurred.

FILE

- ASCII
  - Data elements are represented by ASCII value. Can be opened using any editor.
- Binary → .....

fprintf(fileptr, ...)

fscanf(

②

Buffer reads & stores blocks.

↳ inode contains info about ~~file~~ block and is maintained by the operating system.

fprintf(...)

fscanf(..)

- ASCII aka formatted file.
- variable length of record ⇒ Maintenance of such files is difficult (Issue arises when jumps is happening) value of an attribute may get changed.

store it in a variable

```
struct rec
{
    int roll;
    char name[31];
    int score;
};

struct rec s;

fwrite(&s, sizeof(s), 1, fp);

struct rec x[100];
fwrite(x+i, sizeof(x[0]), n, fp)
```

'w' → if the file is existing the entire data will be lost. To append, we have 'a' mode.
(append) → added at the end

Reading data from the file

→ 'r' mode
and the file must exist.

```
struct rec s;
fread(&s, sizeof(s), 1, fp);
```

→ after reading the file marker also shifts.

feof(fileptr)

if marker encounters end of file.

without feof implementation

```
while(!fread(....))
```

fclose(file).

fseek → sets the marker at a posn in file.

fseek(fileptr, offset, ref)

[offset (non-negative)]

ref (long integer):
- → 0, wrt start of the file
- → 1, wrt current posn
- → 2, wrt end of file. [offset (non-positive)]

fseek(fp, 0, 0) → start of the file.

ftell(fp) → specify the current byte posn.

To get kth record

start = (k-1) * sizeof(~~starting point~~ ~~struct~~ struct rec[0])

Modify Records

old thing will be lost

r+ | r++

modifying ~~roo~~ roll no, 10.

```
fp = fopen(..., "r+");
while(!fread(...)
{
    check = 0.
    ~~check~~.if(...)
    {
        check = 1;
        break;
    }
    ....
}

if (check == 0)
    printf(.....)
```

To modify

```
fseek(fp, -1 * sizeof(struct rec), 1);
...
fwrite(&s, sizeof(struct rec), 1, fp);
```

Sorted - order

→ the dummy records.

Deleting

→ Take the help of a temporary file.

Logical deletion , Physical deletion

⇓

Assigning dummy to file to be deleted.

20/8/2019

Object-Oriented Programming

Prof. Sanjay Kumar Saha

Low level ↓ High Level

→ Procedural Approach
→ Modular Approach
→ Object-oriented Approach

Emphasis is on ~~Programming~~ algorithm/ organisation of data is ignored

To some extent data org. is considered
Related proc. & data ⇒ module

Language which support object oriented feature ⇒ object oriented prog. lang.

OOP

Object ⇒ corresponds to real world entity or concept

Similar type of object ⇓ Group/set ← Description & Behaviour

Class is the collection of similar objects. Class def^n provides the description & behaviour of the objects, belonging to the type.

```
class Student
{
    Roll, Name, Phone_No, dob      → Description
    admission()                    → Behaviour / Method / Member fn.
    change_details()
```

Data Member / Attribute

Data Abstraction

Hiding the complexity of internal representation

Student x ← object is the instance of the class

As end user, implementation details are not sufficient required to know the methods one can utilize.

→ Polymorphism

Same method can have different implementation in different places.

Logically same operation :

⇒ Actual implementation differs

} Implementation to be followed is based on target object.

→ Inheritance → helps in easy maintenance

Sharing of attributes & Behaviour.

Makes the code reusable.

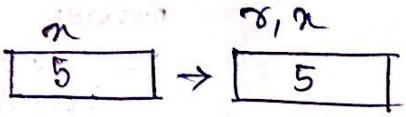
Class (I)	Class (II)
~~f((ff(~~	↓
f11()	f21()
f12()	f22()

C++

Reference variable

```
int n;
int &r = n;
```

& Think as an alias.



↳ r is a reference variable, and it refers to integer variable

C++ : Whenever a reference variable is declared, it must refer to an existing variable. No reference space is allocated, as if it is another name for the variable it refers.

&r, &n ⇒ Same
No seperate memory is allocated

In C++, the reference variable will refer to same variable throughout its lifetime (as per scope).

①
x = 5.
After x = 5

②
x = 5
After x = 10

③

```
main()
{
    f(x);
}
```

```
void f(int p)
{
    p = p + 5;
}
```

→ pass by value.

```
void f(int &p)
{
    p = p + 5;
}
```

→ by reference

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

To display.

```
cout << "Hello World" << "\n";
```

cascading

```
cout << "_ ";
cout << "\n";
```

insertion operator

Output Stream → Related to standard output device (monitor).

Reference to ostream class.

```
cout << "Value is " << a << "\n";
```

```
cin >> a;
```

input — extraction operator.

must have reference

```
~~int &c = max(int a, int b);~~
int & max(int &, int &)
int &c = max(a,b)
```

get the reference.

```
int *c = max(a,b)
```

get the value

```
main
int a,b;
cin>>a;
cin>>b;
int &c = max(a,b);
```

a > b

do certain operations affecting a.

→ #define max(x,y) ((x)>(y)) ? (x) : (y)

Preprocessor directive.

prior to the compilation macro substitution takes place

No fn call hence no push/pop in the stack present

→ There is no type checking in macro substitution, may create problem during run time.

~~inline fn~~ (inline) int max (int a, int b)

request to the compile to make the fn inline. It the compiler grants, every call to the fn will be replaced by equivalent code during compilation.

```
{
    if (a>b)
        return a;
    else
        return b;
}
```

Simple fn => Inline.
Compile time ↑
Run time => fast-

Functions with default parameter(s).

```
calc-percentage (int marks, int total = 100);

float calc-percentage (int s, int tm)
{
    return ~~(s * tm);~~ ~~(s * tm/100~~
           (s * 100.0)/tm;
}
```

Note

→ Only the rightmost subset of arguments can have default values.

Function overloading

A function with same name with different implementation.

→ argument list

Each defn must differ in signature

↓

No. of ~~types~~ arguments and/or ordering type of arguments

```
void f(int a)
{
}
void f(char c)
{
}
```

main

```
char x;
int p, float x;
f(p); f(x); f(z);
```

↑ converted to int

Rules

→ exact match / match after trivial conversion

T	T&
T&	T
T	const T

~~f(int a, int b)~~

→ Promotion (char to int), (short int to int), (float to double)

→ Standard conversion

→ User defined conversion

→ ellipsis (...)

```
f(...)
{
}
```

All rules are followed then relaxition and other is done.

n arguments in the fn call

```
for (i = 0; i < n; i++)
    s[i] = '\0';
for (i = 0; i<n; i++)
{   find match the ith
    argument ⟶ Apply the
                Rule-(1) for
                all defn.
```

$S \cap S[i]$

taking the intersection

```
if (#{S}) ≠ 1.
    Repeat. Report
        ambiguity
else
    the defn in S
    is executed.
```

if no match

```
{
    Apply Rule-(2)
    if no match
    {
        Rule-(3)
        . . . Rule (4)
        . . . Rule (t)
        [check for
         ellipsis].
```

```
f(int a, int b)
f(char c, char d)
f(int, float)
```

```
int i; char x;
f(i, n).
```

S_0 = f(int, float), f(int, c)

S_1 = f(char, char).

$S_0 \cap S_1$ = NULL

```
structure Student
{
    int roll;
    char name[31];
    void get_into();
    void display();
};
```

scope operator

```
void Student::get_into()
{
    cin>> roll;
    cin>> name;
}
```

```
Student s;
s.get_info();
s.display();
```

→ However structures have no access control

```
class Student
{
    int roll;
    char name[31];
    public:
    void get_info()
    {
        :
    }
    void display(void)
    {
        :
    }
};
```

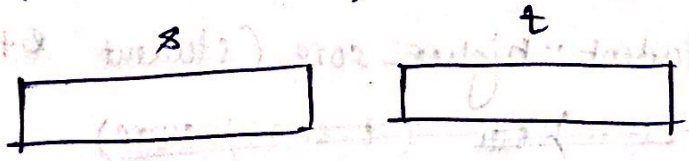
inline

inline

Request to make it inline to the compiler.

```
void Student::display(void)
{
    :
}

int main(void)
{
    Student s;
    Student t;
```

for each object space is allocated



s.get_info()

invoking object

→ Each object will have its own copy of data members.

→ All objects of the class share single copy of member fn code.

A member fn is always invoked by an object.

(8) Reference of invoking object is automatically passed.

public	private
accessible outside the class also.	can be accessed using member functions.

→ By default (if nothing is mentioned) => ~~Public~~ private

Attributes & methods that can work with the data members are bundled together ⟶ ENCAPSULATION

An user of a class should know public members only. for public members => prototype

Rest into => Rest hidden

Another method

```
int compare (Student x)
{
    if (score == x.score)
        return 0;
    if (score > x.score)
        return 1;
    else
        return -1;
```

Student *sp = x.higherscore(7);

Student & higherscore (student s);

(Student &t) Student :: higher_score (Student &t)

{

~~if (t.score) > (t.score) score)~~

if (t.score > score)
    return t;
else
    return *this;

}

How many students have taken admission during the session of the program?

```
:
std[i].admission();
i++;
```

Suppose. We use static int cnt;

for a class there is only one copy of static data members, all object share this.

→ Student s;

Each object will have their own copy of data members.

→ Declare the static data member outside the class.

~~int Student::count = 0;~~

→ int student::cnt = 0;

```
static void show_count()
cout << count << "\n";
```

became a part of the class (can be invoked without the object).

No non static elements can be referred. But one can declare local object and can work with that.

Student::static_fn(...)

26/8/2019 Programming Practices

Prof. Sanjay Kumar Saha

→ What is class?
- → Encapsulation
- → Abstraction.

→ How to define a class?
- → data members/attributes
- → member fn
  - can be defined inside the class defn
  - can be defined outside the class (scope resolution operator).

→ Any member fn has to be invoked by the object of the class

→ Reference of invoking object is automatically passed.

→ Access control → (i) Private → Can be accessed only in member fn
(ii) Public → Everywhere.

data members → Private
member function → Public.

→ if data members become public, then there would be a compromise in security.

→ Data members → public
[NOT ADVISABLE]

No. of methods have certain common activity ⇒ repeat in all the methods

private
After
Make a seperate fn with those common parts, and use it in the methods whenever necessary.

②

INDEPENDENT — PUBLIC

WILL ALWAYS BE A PART — PRIVATE

To hide sensitive issues we privati else we public.

↑ security purpose.

static data member

⇓

attribute of the class itself not of the individual object | All objects share this attribute

Static member fn

⇓

Does not depend on any object.

sizeof(object) ≥ Sum of size of non-static members.

```
class Student
{
    int roll;
    char name[31];
    int score;
public:
    void getdata();
    void showdata();
};
```

```
void student::getdata()
{
}

void student::showdata()
{
}

Student s,t;
s.getdata();
s.showdata();
t = s;   => Byte wise assignment.
```

Dynamic memory allocation

```
int *p;
p = new int;
delete(p);   ← freeing up memory
```

```
int *p;
p = new int[10];
p[0] ⟶ first element of the array.
```

```
Student *s, x;
s → get_data();
(*s).get_data();
x.get_data();
```

In a class, one can have objects of other class.

```
class Date
{
    int y, m, d;
    public:
    ....
};
```

```
class Student
{
    ....
    Date ....
};
```

If student is above Date then we put `class Date;` ⟶ Forward Declaration above.

We cannot use `cin << dob;` Not possible

④ Nesting of class defn.

```
class A
{
    :
    class myData
    {
    };   or.   } dt;
};  . . . .
```

↑ create an object.

Defn is limited to that class

```
void A::myDate::get_data( ) .
```

```
class Item
{
    char icode[5];
    char iname[31];
    float price;
    float qty;

    public:
    void get_data();
    void show_data();
    void set_price(float);
    void change_qty(float);
};
```

Code {...} Code {...} Code {...}

~~class ItemList()~~
~~{~~
~~}~~

```
class ItemList
{
    Item list[10];
public:
    void getdata();
    void show_all();
    Item costliest_item();
};
```

If trying to access a private member from outside, we need to get the data using a getter method.

```
ItemList Interface
{
    static void operation()
    {
    ...
};  }
```

Implement menu in main()
OR ItemListInterface :: operation()

27/8/2019 Programming Practices Prof. Sanjay Kumar Saha

```
class Income
{
    float amt;
    public:
    void set_income(float f)
    ....
    void show...
};
```

1) In the classes, declare the fn as friend.
2) If required, forward declaration of classes area.

```
class Expense
{
    float amt;
    public:
    ....
};
```

```
float netBalance(Income ic,
                 Expense exp)
{
    return (ic.amt -
            exp.amt);
}
```

4) objects of all the classes of which it is a friend are to be passed as arguments.

3) In each of the class declare it as a friend.

Friend fn

If a fn is friend of a class then that fn can access, ~~the public members~~ private members of that class. (Eg. if netBalance becomes friend then it can access the private data members).

If a method working with objects of different type.

Global fn

Working as friend of no. of classes.

```
friend ~~void~~ float netBalance(Income, Expense)
```

inside the class defn.

A fn member of one class acting as friend of others, then we use.

float

→ friend ^ Income :: netBalance (Expense)

→ I may skip forward declaration.

friend class B ;

⇒ all member fn of class B will act as my friend.

getter fn → mutuator / MODIfier

getter fn → accesor

```
int retx() const
{
    x++ ;
    return x ;
}
```

→ Mutuator → Management

→ Accessor

Constructor

Destructor

⇒ special type of member fn which are called automatically at the time of object creation

Can be used to initialize the object

⇒ can have any / arg. with default value.
no. of arguments

Normally, public / protected.
(in some case, it can be made private)

⇒ fn name is same as class Name

⇒ No return type (even, not void).
[implicitly return the object reference]

⇒ Can never be static

→ If user does not provide any constructor, system provides a constructor called default constructor.

→ When user writes a constructor, system withdraws default constructor.

```
class student
{
...
public :
student(int n = 0)
{ ... }.
```

can take a single OR no argument

Object declaration must match with exactly one constructor

default.

Constructor with no arguments or all arguments with default value ⇒ Default constructor.

```
student s(5);
student t = s;
```

⇒ Copy constructor — System automatically provides unless declared by the user.

Bytewise copying

To invoke copy constructor

~~student t;~~ } Not an issue of copy constructor.

```
t = s;
```

~~student s;~~

```
student s;
student t = s;      ← } copy constructor.
student t(s)        ←
```

student t(student).

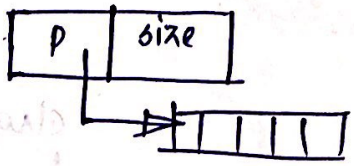
↑

Diff. case altogether

(4)

```
class Array
{
    int *p;
    int size;
  public:
    Array(int s = 0)
    { ...
    { ...
};
```

Array x(5);



Array y = x; ⟶ Copy constructor.

ptr data member, for which memory is dynamically allocated.

→ Now, system given copy constructor will make the ptr of new object (i.e which copied) to point to the same location.

→ Two object will share ~~the~~ same single copy of ~~reference~~ referred location. changes made by one will affect other.

→ If it is undesirable, one must write own copy constructor to have seperate allocation.

copy constructor ⟶ Array(const Array &t)
. . . . .

Destructor

↓

automatically

called when the object is destroyed.

→ public → non static

→ no return type → no argument

→ ~classname()

not even void.

```
~ student()
{ ...
}
```

stored in stack.

1.s
2.t

1.t
2.s

→ We must write our own destructor when memory allocation is made

delete ap ;

```
Array a(*ap);
delete ap;
```

↓

System will delete only ap.

Our destructor will delete the entire thing.

02/9/19

Programming Practice

Prof. Sanjay Kumar Saha

Class Test

Syllabus upto (day before 02/9/19).

⇓

upto Constructor & Destructor

Date : 16th Sept, 12:00 pm

DATA ABSTRACTION

- To hide the complexity of representation
- Implementation details are hidden from the user.
- Behaviour of similar types should be similar.

eg. a ⊛ b , ⊛ p

↑ playing multiple role

<< , >>

Left shift, Right shift.

Operator Overloading

```
class Complex
{
    int r, im;
public:
    Complex(int a=0, int b=0)
    {
        r=a;
        im=b;
    }
...
};
```

```
c3 = add(c1, c2);

Complex Complex::add
(Complex t)
{
. . . .
}

Complex add(Complex t1,
Complex t2).
{
. . . .
}
```

c3 = c1 + c2

↑ overload the '+' operator.

To overload ann operator.

fn name

operator symbol ( )

eg. operator + ( )

c3 = c1 + c2 ; → c1.operator + (c2)

(operator on the left taken as invoking object)

OR operator + (c1, c2)

either write a member fn or a friend fn.

Cannot have any default value

~~complex~~

```
complex Complex :: operator + (complex t)
{
    complex c ;
    :
    return c ;
}
```

Cannot discriminat pre & post

Binary Operator

Friend fn

→ Operator on the left is 1st arg.

→ operator on the right is 2nd arg.

No implicit type casting

Overloading operators to handle casting.

for eg. ~~c3 = c~~

$c_2 = -c_1$

$c_3 = c_1 - c_2$.

operator − (complex)

```
complex operator * (int k,
                    complex c)
{
    return (c*k)
}
```

$c_2 = c_1 * k$

```
Complex operator * (int t)
{ ... }
friend Complex operator * (int i, complex c).
```

complex c1;

istream → cin >> c1;

ostream ← cout << c1;

ostream objects.

friend void operator<<(ostream &, complex)

↓

ostream

... return t;

friend ostream operator >> (

istream &, complex &)

~~friend void operator <<~~

~~friend void~~

Note :- Destructor should be there for a Constructor

class Array

{

int *p;

int size;

public :

...

};

Array (int size)

{

...

}

+ copy constructor

Array (const Array &t)

~ Array ()

{

if (p != NULL)

delete p;

}

byte-wise assignment

c = a

↓

c.operator = (a)

void operator = (Array t)

{

}

=

↑

overloading is done by member fn and NEVER by a friend fn.

Array b = a

Array (a)

Array b(a)

To have no. of references

operator [] (int)

↑

Member fn only.

int operator [] (int n)

{ ...

}

Array

{

...

int * ref_count

...

Array (int s = 0)

{

: * ref_count = 1

else

ref_count = NULL;

}

④

```
Array (Array &t),
{ ...
}
void operator(..)
 {
 }
```

~~Destructor~~
~~Destructor~~
~~{ ... (decrease refcount),~~
~~}~~

Destructor { [decrease refcount] } ;

03/09/2019

Assignment Operator overloading

Programming Practice

Sanjay Kumar Saha

→ Return type as per requirement

return type — void

obj1 = obj2

obj1.operator(obj2)

Don't inherit the entire class, if you need only one part

return type object

obj1 = obj2 = obj3

Student

Roll
Name
Dept

Dept

Dept
Dname

StudentList

Array of student object

DeptList

Array of dept. object

Dept

DeptList

Student

Student List

Student list[100]

display()

to display department, we need to provide support

Student return_student (int rollno)

{ ....

return student

}

(2)

Type Casting

- Source
  - → Primitive data type
- Destination
  - → Primitive data type
  - → Object of a class

destination = source

x = y

explicit . (float) x.
or float(x)

```
Time t;
.
.
t = n;
```

int no. of sec.

```
class Time
{
    int min, sec, hr;
public:
    ...
};
```

⇓

Have a constructor with basic type as argument

Source t = Destination n

```
Time(int n)
{
    hr = n/3600;
};
```

Destination n — Source t — member fn.

Write suitable type cast operator

```
operator int()
{
    ...
}
```

operator φ ( )

type in which casting is required

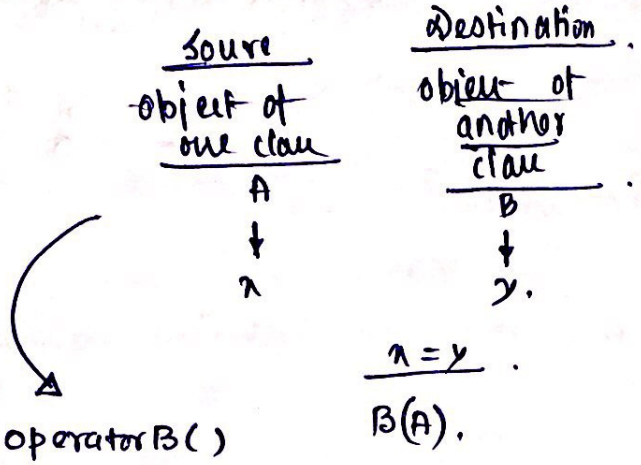
type cast operator

⇓

member fn no return type is implied

```
double d
d = float(d)
```

C1 = C2 + (3) → user-defined function




```
class month_day
{
    int month, day;
public:
    month_day(day_of_year k);
    ...
};

class dayOfYear
{ int day;
  ...
  month_day m;
  day_of_year d;
  m = d;
public:
```

```
operator month_day()
{
    m = ___
    d = ___
    return
      month_day(m,d)
}
```

09/09/2019 Programming Practice Prof. Sanjay Kumar Saha.

Inheritance

Base class / Super class / Parent class

↓ Inheritance → To share attributes & behaviour of parent class with the child class.

Derived class / Sub class / Child class

Need to add certain attributes / behaviour to modify in an existing class

⇓

direct change in the class may not be possible (source not available) even if accessible some case old behaviour is required.

For same logical order & in some case modified behaviour is required.

'OR'

Redoing old things which are required and designing own class covering all desired features.

Best Soln.

Existing Class

↓ Inherit

Child Class

Add additional attributes / behaviour or change existing behaviour.

```
class A
{
    int x;
public:
    getdata(void)
    {
        cin>>x;
    }
    show(void)
    {
    }
};
```

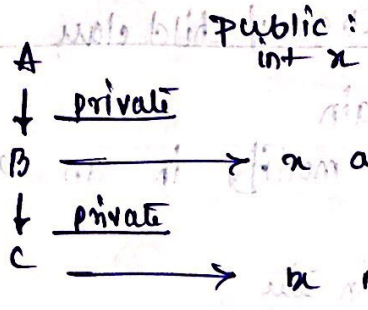
class B : private A → Type / Mode of Inheritance.

② private data of base class are not accessible in derived class (but those are inherited)

Important

Protected ⇓ Protected members are accessible in class itself & in derived class

Multi-Level Inheritance



Type of Inheritance	Access Specifier in base class	Accessibility in derived class
~~Inheritance~~ Inherits everything	Private	Not accessible
	Public	accessible Private
~~Protected~~	~~Public~~ Protected.	Private
Protected.	Private	Not accessible
	Protected	Protected
	Public	Protected

When inherited in private

public base members ⟶ private in derived class.

Selectively the public members can be inherited as public/private

Public → Private / Protected / Public

Private → Not accessible

Protected → Protected

Public → Public.

Use scope resolution operator :: to access the fn from the base class having the same name.

function overwriting

If a derived class fn has same name (signature may not be same) as a base class fn then it is that base class defn that is overwritten in derived class.

With a derived class object if overwritten fn is invoked, it always goes till derived class.

to go to the base class, use scope resolution operator, (class_name::)

Lookup for a fn

~~An object invokes a fn~~

C++ → fn overloading is in a single class.

LOOKUP Mechanism

→ Find class in the same level

→ if not found, goes to the class at higher level

→ if not found, error.

→ if an variable is present in the ~~fn, class~~, fn use that

if not found → if variable is present in the class, use that

→ if not found go to the higher class

→ finally global variable

(4) static members of base class

→ Also be shared by derived class instances

→ accessibility in derived class to protected/public

Friendship

A ⫽ B

friend class B granted friendship to A

A → D automatically does not inherit friendship.

A(const A &b);

A ⇒ Aa ; def. constr.

↓

B ⇒ Bb ; def. constructor

A → B derived class const. can work with only additional attributes.

```
B(const B &t) : A(t).
{
    y = t.y ;
}
```

```
B(int a=0, int b=0) : A(a)
    y = b ;
```

Call default const. of base class.

explicit call to the base class constructor.

~~20/9/19~~
17/9/19

Programming Practices

Prof. Sanjay Kumar Saha

Inheritance

→ Sharing of attributes behaviour

Generalization

Child class → Mode of inheritance
↓ Common. → Scope Rule.
Parent

Specialization

General class/parent
↓
Subclass.

Over-riding

fn name same as that of base class in derived class → Signature may/may not be same.

Whenever derived class obj => calls a fn.

derived class reference is automatically converted to base class reference.

Derived class to Base class

B(const B &t) : A(t).

A a;
B b;

~~B~~ a = b.
↳ Takes base class part & rest is ignored.

b = a
NOT AUTOMATIC

Base class to Derived class

B(A _)

OR operator B()
{
:
}

② A ⟶ B.

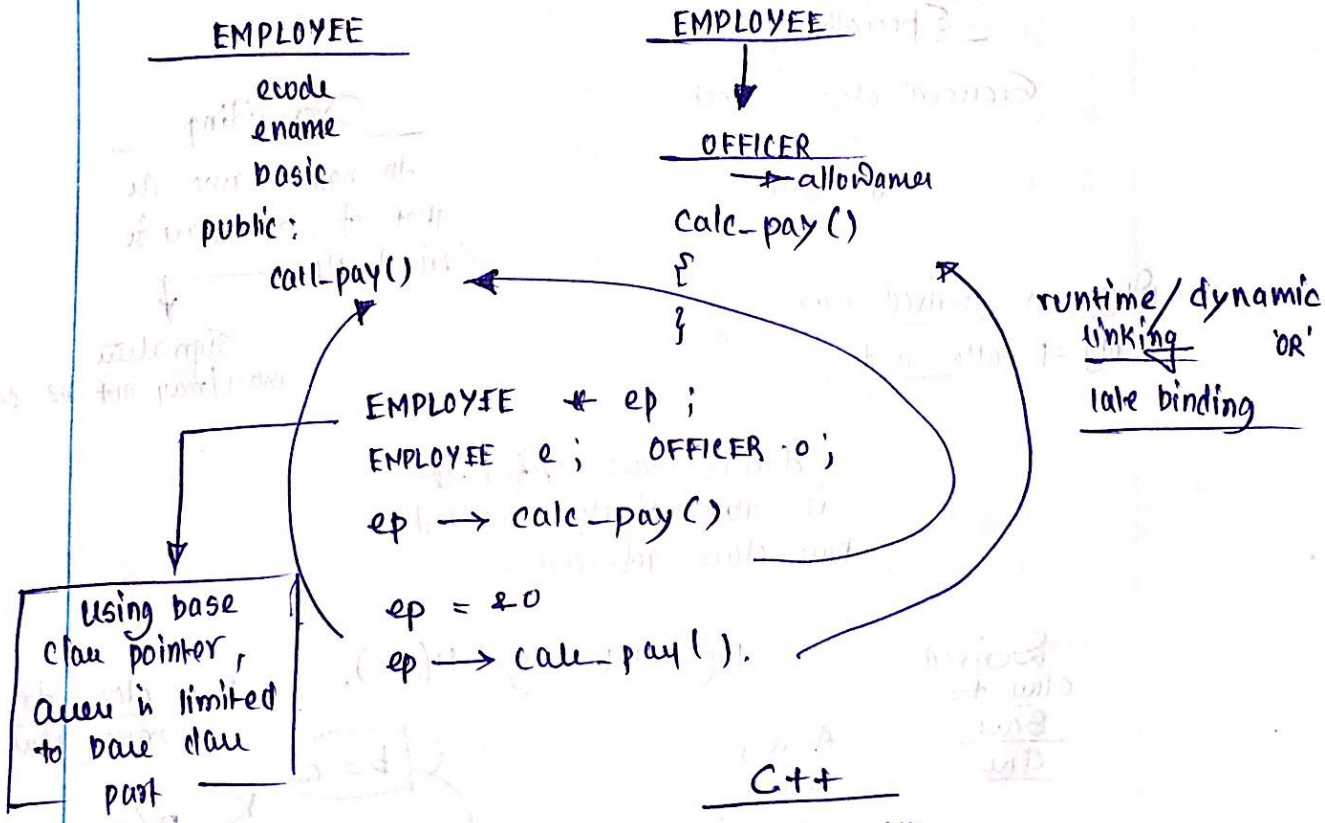
A *p ;

⇩

point to either a base class object
or a derived class object

p = &a ; p = &b ;

p = new A ;
p = new B ;



C++

- Base class fn for which run time polymorphism is required has to be overriden in derived class with same signature.
- Base class is to be declared as virtual.

hence Base class.

```
EMPLOYEE
{ .....
{ virtual void call_pay() { . . . }
```

System

Whenever a class has a virtual fn, a table for the class called VTBL is prepared. Maintains an entry for each virtual fn in the class.

→ fn prototype & starting address of the code.

In the class, system keeps an attribute VPTR.

VPTR ⇒ points to the VTBL of the class.

A derived class inherits from the base class.

VTBL of base class is copied.

VPTR of any derived object point to VTBL of its own class.

If any virtual fn is modified then its address in VTBL is updated. If any new virtual function is added, then an entry will be made.

A ⟶ B.

ap = new B(. . .);

⇓ constructor.

delete ap; ⟵ makes a call to the base class destructor.

↳ To handle define the destructor as virtual.

~~Why?~~ Why?

derived class destructor

↓

base class destructor.

24/9/2019

Programming Practice

Prof. Sanjay Kumar Saha

Class Student, Faculty, Staff may have common attribute which can be used for generalization. We use a higher class for this.

Abstract Class

In reality, there exists child/sub/derived class objects. They have lots of commonalities. To avoid repetition, take out common attribute/behaviour and form a input/base class. If it is so then reality, an implementation object of the class will not exist then make it abstract class.

A class with atleast one pure virtual fn is an abstract class.

Note Any class derived from an abstract class must over-ride the pure virtual fn.

Same name - same signature

```
class A
{
    public:
    virtual void f(-) = 0;
};
void A::f(-)
{
}
```

```
class B
{
    :
public:
    void f(-)
{
}
};
```

```
b.f();
A a;
```

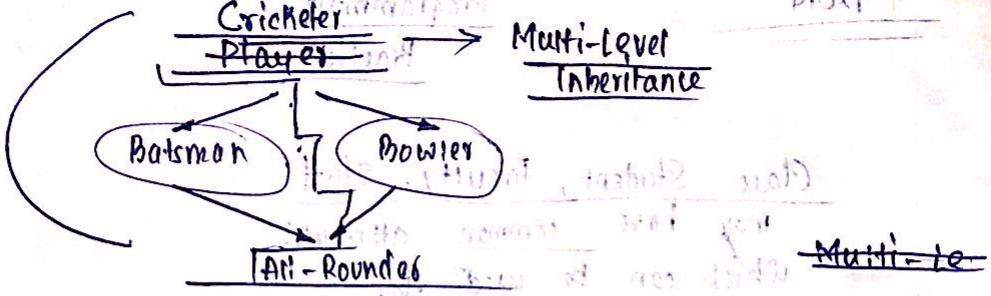
✗ Not possible

No direct instance of abstract class

A a; → ✗

A *p → ✓

(2)



Multiple Inheritance

~~Multi-le~~

Multiple instance of common ancestor via different path.

Note:- Do not consider a consists of relationship.

Eg. ~~Pen~~ : person

husband wife

family

Destructor can never be pure virtual fn

Common ancestor will be declared as virtual base class for its children

class batsman : virtual public cricketer

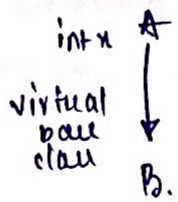
{

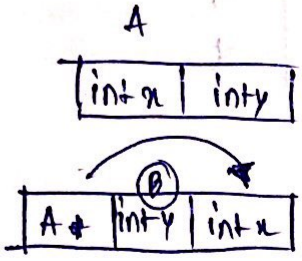
}

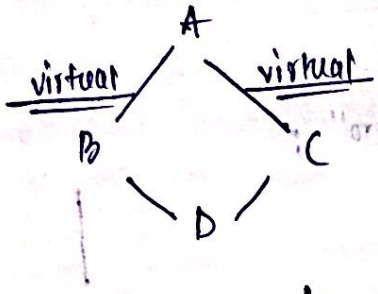
class ~~batsman~~ bowler : virtual public cricketer

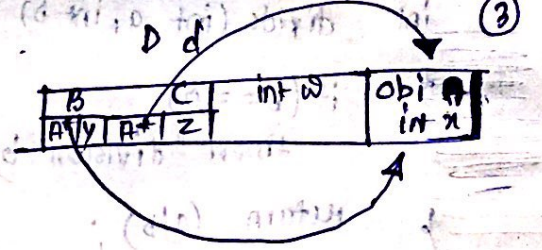
{

}









If there is any fn with same name present in both B and C, then d must override this.

```
D d;
A *p;
p = &d;
```
← limited access to base class

Exception Handling

```
int divide(int a, int b)
{
return a/b;
}
```

```
if (b==0)
  exit(1);
```

```
int main(void)
{
char c;
int x; int y;
do
{
  cin>>x;
  :
  cout<< divide(x,y) <<"\n";
  :
} while(c=='y' || c=='Y');
```

try block

```
try {
:
throw exception
}
catch (_)
{
}
catch (_)
{
}
```

To raise the exception

abrupt termination of the program

```
if (y==0)
};
```

division by zero

exact match / match after trivial conversion.

No promotion / No standard conversion

Comes out from subsequent catch blocks

```
int divide (int a, int b)
{
    if (b == 0)
        throw "division by zero";
    return (a/b);
}
```

```
class myexception object-
{
    char s[];
    int x, y;
    public:
        myexception object (.....)
        {   strcpy(s,c);  x = z;
            y = j;  throw exception exp r;
        }
    void show()
```

```
a < b }
throw
myexception object
    t("__", a, b).
} throw b;
```

Instead of default terminator.
fn

```
#include <excep.h>
set_terminate (function pointer)
set_unexpected (...)
```

```
return type fn-name(args[]) throw exp1, exp2, ....
    try {
    fn-name()
    }
    catch()
    {
    }
```

```
int max(int, int)
float max(float, float)
float max(char, char).
```

|| Same code repeatedly in all defn.

Function Template

used to create an instance of the fn for a specific type.

It is a skeleton/template for generating a number of function definitions. Formal arguments may be of generic type. Instead of specific type, one can put placeholder for data type. Actual type will be decided looking into the actual parameter.

actual parameters

c = max(a,b)

~~max(Ta, Tb)~~

~~max(Ta, Tb)~~

```
template <class T>
T max(T a, T b)
{
.....
}
```

each generic type must appear at least once in the argument list.

```
if a > b
    return a;
else
    return b;
```

```
int a, b;
float f1, f2;
cout << max(a, b);
```

An instance of max(int, int) exists or not? If not then create an instance using function template.

```
cout << max(f1, f2);
```

max(float, float)

Whenever an instance is created for a fn template

it will work only with exact match.

however

cout << max(a, f1) → create an error

① Exact match with normally defined (not template) overloaded version. If found, then.

else ② tries to create an instance with template if possible consider that else looks into normal overloaded fn with promotion, conversion etc.

②

```
class Student
{
    ...
};
```

must have → Student operator > (Student)

```
cout << max(s1, s2)
```

```
{
    :
}
```

friend ostream& operator << (ostream&, Student)

```
{
}
```

No. of classes with same attributes & methods may have to be defined over if only certain attribute type is different but role/interpretation is same.

→ All generic types required for class defn.

```
template <class T>
class Array_1D
{
    T *arr;
    int size;
  public:
    Array_1D(int u=0)
    {
        size = u;
        if(size==0)
            arr = NULL;
        else
            arr = new T[size];
    }
    T add()
    {
        ...
    }
};
```

Array_1D <int> x;

Every member fn of a class template is a fn template -

```
template <class T>
T Array_1D<T>::add(...)
{
    int i;
    ....
}
```

fn template

```
template <class T>
int Array<T>::get_size()
{
    return (size);
}
```

template fn

class template

When an object is created for a type and it is the first object for the generic type.

- Instance of the ~~instance~~ class defn has to be created → All static & non-static data members in a static data member (if at all) are also created.

For the next object, only the non-static data members are created.

```
class c1 < class <c2> _ >
                  ↑
                space.
```

if func is a static data member.

```
template <class T>
T Array_ID<T>::c = 0
char Array_ID<char>::c = 'a';
```

specific initialization

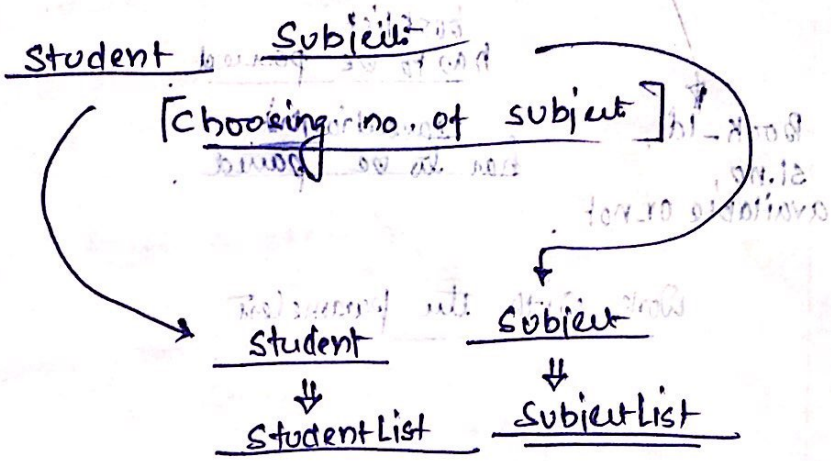
```
template <class T>
class ~~Simpl~~ Spl_Array
{
    ...
    friend class Array_ID<T>;
};  ...

template <class T>
class B: public A(T)
{
    B(_) : A<T>(_);
};
```

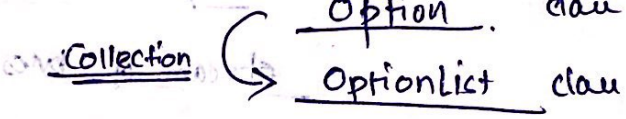
Do have, to specify

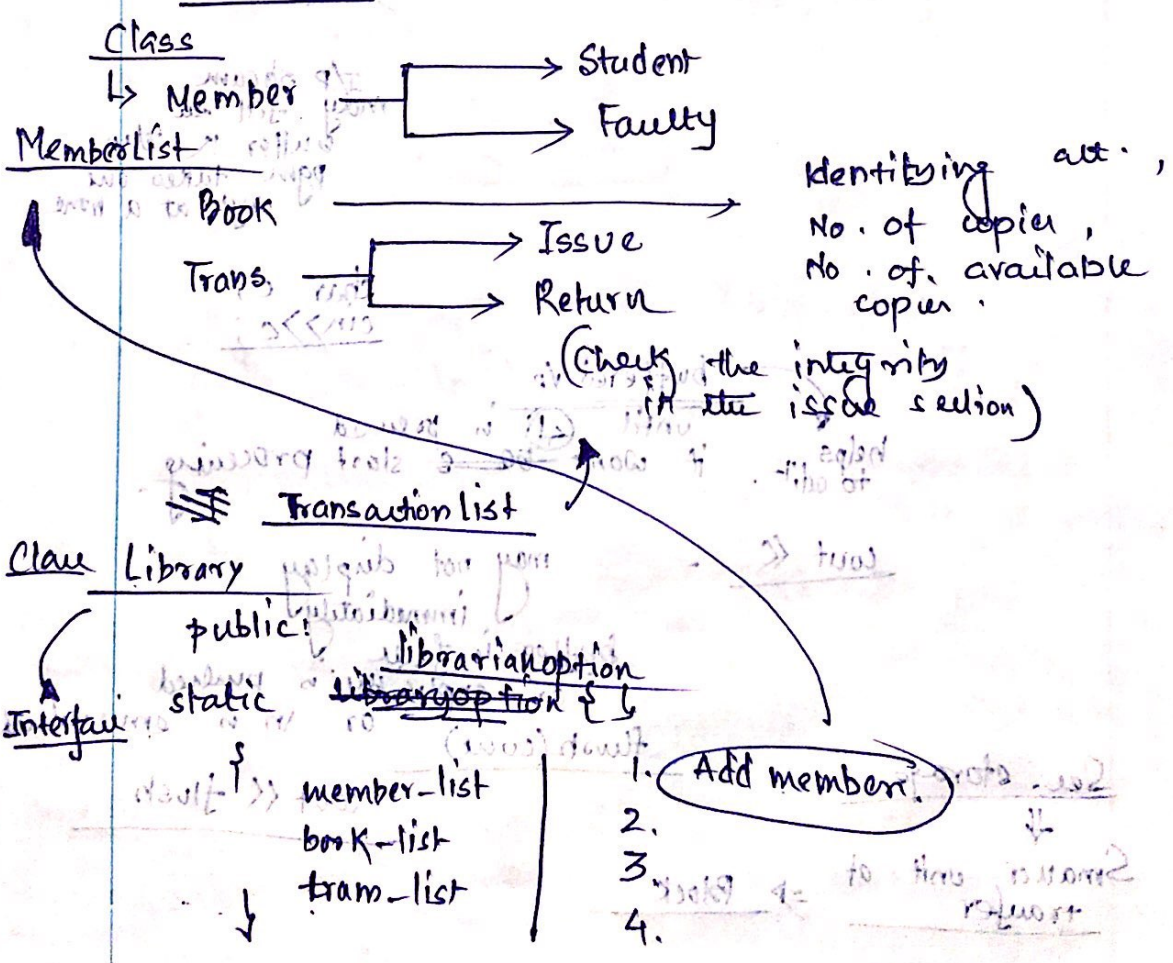
21/10/2019

Programming Practice

Prof. Sanjay Kumar Saha







Issue.procedure_issue(...)

booklist has to be passed

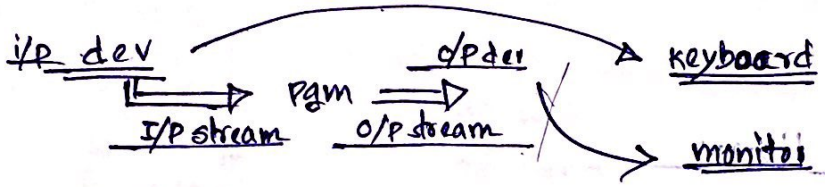
transaction has to be passed

Book_Id, sl.no, available or not

work with the parameters

I/O & FILE

stream of bytes



I/P stream may fill the buffer & the pgm takes one byte at a time

char c;
cin>>c;

buffered i/p until ↵ is pressed

helps to edit. it won't start processing.

cout << - may not display immediately

buffer is full or explicitly is pushed or \n is encountered

flush(cout)

cout << flush

See: storage

Smaller unit of transfer => Block

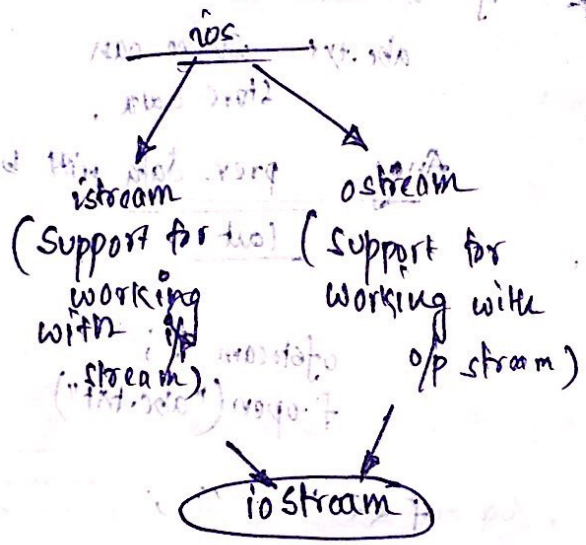
streambuf class

↓ buffer management

information about the memory for buffer to support buffer filling, reading from buffer, etc. (buffer mgmt.)

ios | keeps a ptr to 'streamingobject'

keeps basic file info like binary/text file information open for reading & writing




```
char ch;
cin>>ch;
while (ch!='\n')
{
    cout<<ch;
    cin>>ch;
}
```

ignores tab/space/'\n'

Go on casting

cin.get(ch).
get(ch).
...

→ cin.get(ch); returns istream reference.

ch = cin.get()
↳ return int ascii value

cin.get(str, 50) ← takes at most 49 char

cin.getline(str, 50); ← or it encounters a newline which ever happens first

'\n' will stay in the stream

abc
abc\nxyz\n
'\n' will be deleted out

(14)

cout.put('a');

~~istream~~ → ifstream class

ostream → ofstream

iostream → fstream

ofstream f("abc.txt")

f is related with file abc.txt where can store data.

Any prev. data will be lost

file is opened for writing.

ofstream f;
f.open("abc.txt")

f << name << " ";

One can work with fstream f

ifstream f("abc.txt")

ofstream → [f << name; f << "\n"; f << age; f << "\n";]

f >> name

~~while(f.get(c))~~ while(f.get(c))

~~{~~ { cout << c;

~~}~~ }

~~f.close();~~ f.close();

f.open("abc.txt", ios::out | ios::app)

Programming Practices

22/10/19
SKS Sir
2nd CT
→ NOV 18
~~(except FILE)~~

Binary file

1) ios::in => A file must exist (to read)
2) ios::out => to Write (old data lost)
3) ios::app => to add data at the end.

```
class Student
{ ... };
```

Writing

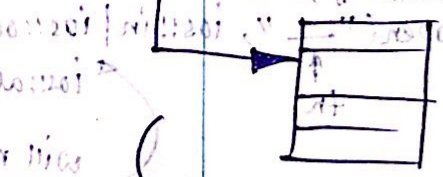
```
fstream f;
f.open("students.dat", ios::out|ios::binary);
student s;
s.getdata();
f.write((char*)&s, sizeof(student));
```

write(char *, no. of bytes)
⇓
from where to get data to be stored,

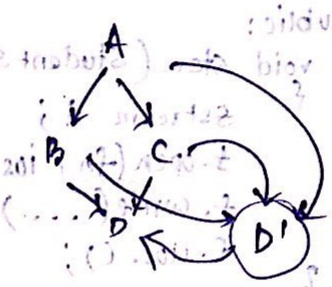
read(char *, no. of bytes)
⇓
Where to put

Student

	roll	name	score
3	100	78	



if address is stored, then who says the address will stay the same.



Read

```
fstream f;
f.open("__", ios::in | ios::binary);
while(f.read(...))
{
  s.display();
}
```

OR

```
while(!f.eof())
{
  f.read(...)
}
```

→ to prevent last record coming twice.

~~fseek~~ Random Access

get marker → from where to read

put marker → where to put

object.seekg()

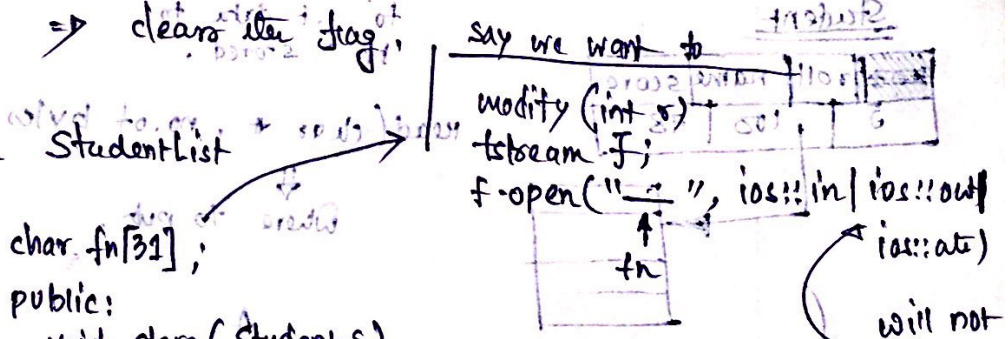
~~offset, reference~~ ⇒ ios::beg (wrt start)
ios::end (wrt start)
ios::cur (wrt current).

f.seekg(ios::beg, 0)
f.read(...) ← gets the first object.

On encountering eof, a flag is set.
Subsequent read operations are ignored.

object.fail() ⇒ if last operation fails.

f.clear() ⇒ clears the flag.




```
class StudentList
{
    char fn[31];
public:
    void store(Student s)
    {
        fstream f;
        f.open(fn, ios::out | ios::app);
        f.write(....);
        f.close();
    }
    void showall()
    { }
};
```

```
void searchastudent(int p)
{ }
```

tellg() → posn of getmarker
tellp() → posn of putmarker

```
searchstudent(int p).
if(rec!=0)
{
    f.seekg(ios::beg, (rec-1)*sizeof(student));
    f.read(...);
    f.seekp(                    );
    f.write(...)
}
f.close();
```

if we have variable sizes, then we can store the size look up table in another memory structure.

~~05/10~~

05/11/2019

Programming Practice

Prof. Sanjay Kumar Saha

NAMESPACE

⇓

- To group named entities within a scope. (other than Global). (Identifier, fn, class, etc).
- To avoid name collision
  entities by static name can exist in diff. namespace.
- May span multiple-blocks ⇓ maybe in different files also

should not be part of main()

```
namespace nds
{
    int x;        ⇒ available to all fn in this namespace.
    void display();
    class ..... };
```

May have diff. files

→ nspace1.cpp
→ nspace2.cpp.

void ....f(void).

Anyother global fn.

~~std~~ nds :: f(void)

⇓

use this name space.

void nds :: A :: display(..)
{ ... }

~~to~~

#include nspace1.cpp

2 ways

using declaration

using ~~declaration~~ declarator.

using nsl :: abc;

for every member, I have to provide a declaration

using namespace nsl;

Standard Template Library

is a collection of lots of tools & facilities with efficient implementation that can be used for developing applications.

List of Nos.
we have to find the average.

→ Array
→ Linked List

→ start from 1st element
add..

→ calc. average by dividing by count.

Taking help of (stl) →
1. container
2. iterator
3. Generic algorithm
4. function object
5. Allocator & Adapter.

Provide limited interface in a container

```
vector<int> v;
vector<int> :: iterator p;
```

Container → is an object that holds no. of objects say, vector, list.

- Mostly are template & abstract.
- General use of container.

Iterator → Helps to iterate through the container without identifying the name, type, ...

- Container specific

```
containertype <type> : iterator i;
*i → gives the value.
i++ → Increments.
```

Generic algorithm → to provide basic support for the container.

Function object → to customize the behaviour of certain std. algorithm.

Stack, Queue

Vector

Random access,
↑ Array like, expandable.

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

vector <int> v;
```

To add data at the end :

v.pushback(10);
v.size() → Gets the size
v.at(index) → checks the bounds.

v1.swap(v2) → Swap the contents
v1 = v2 (all elements) assignment-

==, !=, >, <

void pop_back() → Remove the last element-
↑ will not return any value

v.front()
& v.back
Both getter & setter.

v.resize().
Resizes, and add default element if necessary.

v.erase()
all.
~~v.erase()~~
v.erase(index)
Remove the element at the index

begin()
return iterator ptor 1st posn.

end()
last + 1

Iterator

```
vector<int> :: iterator it;
it = v.begin()
while(it != v.end())
{ ... }.
```

~~list()~~ #include<list>

List

All features of vector present here.

void push_front()
void pop_front();
void push_back();
void pop_back();

if container contains user-defined objects then for comparison on container

⇓

necessary support must be overloading of == , != , <

remove (iterator)
→ all occurances will be removed

v.sort() == must be overloaded

support for operator <().

l1.merge(l2);

rbegin(), rend(),

#include<stack>

Stack

stack<int> stk;

Returns the top element ⇐ st.push(—);
st.pop(—);

Stack(adapter),

```
while( !st.empty())
{ cout<<(st.top()
} st.pop();
```

#include<queue>

queue<int> q

q.front()
q.back()

pop_front()
push_front()
... back();

Class Test-(18/9/19)
Remaining
stdexcept file

11/11/2019

STL

Prof. Sanjay kumar Saha

(1)

⇓

different components

contains → sequence container
vector, (list)

adapter
stack, queue, etc.

Iterators

generic algorithm
→ fn objects
→ allocators.

Stack

stack <> st

st.push()
st.pop()
st.top()
.size()
.empty()

```
while(!st.empty())
{
    cout << st.top()
    st.pop();
}
```

```
while(!st.size())
{
}
```

Queue

empty()
size()

can be used for assignment

front() → returns 1st element
back() → returns last element
pop()
push() → append

Priority Queue

```
#include <priority_queue>
priority_queue <string> pq;
pq.push (...)
```

basic data type

higher the value, higher the priority.

```
pq.push(10);
pq.push(20);
```

20 > 10

pq.top() ⟶ element with the highest priority.

```
class mystring
{
    string w;
public:
    mystring (string s)
    {
        w = s;
    }
};
```

operator overloading

```
bool operator < (mystring s)
{
    return (w > s.w)
}
```

if there are issues

```
bool operator < (mystring s1, mystring s2)
{
}
```

#include<string>

string s1("abc");

s1 = "abc"

string s2(s1)

*s1 ⟶ x (Not possible).

s1[index]

↳ s1.at(index).

if out of range, then exception is raised.

s1.find(string s2);

↳ it returns the start index when s2

if it cannot find

returns string::npos.

↳ "aeiou AEIOU"

find_first_of(string)

⟶ posn of first occurence of match with any of the char in argument

find_last_of(string).

find_not_first_of(^)

↳ find the first occurence of mismatch.

to get a substring.

substr() => returns another copy

substr(pos) => return substring posn to end.

substr(start.posn, end.posn).

size(), empty(), swap().

~~replace()~~ .

~~replace()~~ replace (arg1, arg2, arg3, arg4)

arg1 ⇓ from which the replacement starts

arg2 ⇓ how many ch. to be replaced

arg3 ⇓ Replacing string

arg4 ⇓ starting pos^n in replacing string.

```
#include <pair>
```

pair ⇒ collection of 2 values.

```
pair<type1, type2> p.
p.first
p.second
```

p = pair<—, —> (value1, value2)

↳ one ~~key~~ copy of key allowed

map

↳ sorted associative

pairs are stored in a sorted manner. on key.

value ⇐ information associated with the key.

queue, dequeue

```
#include<queue>
```

deque ↳ double ended

```
map<int, string>
```

↑ type of key, ↑ type of value.

```
m2(m1)
m2(&pa[i], &pa[j])
```

```
m2.insert( pair<int,string> :: value_type (1, "abc") );
```

⇓

Returns a pair

<iterator, bool>

one copy of key is allowed

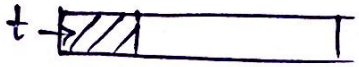
↳ pair <map<int, string> :: iterator, bool>

r.second == true.

begin()
end()

map<_,_>::iterator t;

t = ~~m1.m~~ m1.begin()
t++



r.first = t
value. (r.first) → first

m1[key1]
⇓
m1.at(key)
OutofRangeException

if datatype not a basic datatype then <, > operators must be overloaded.

hash_map
⇒ associated array.
elements are stored according to hashing scheme on the key.

to access the element in a faster way without the burden of sorting on key
⇓
read in map.

String
char const *c = c_str(string)

hash_map <type1, type2>
⇓ ⇓
~~my~~ value type.
my type

hash_map <key type, value type, Hash fn, Equality object>

class template,
fn template,
namespace template