Object Oriented Programming

C → Procedure oriented Programming
↓ top to down (solved)
approach

C++ → C' = C + (1) → Object oriented Programming
↓
object

array → structure
function

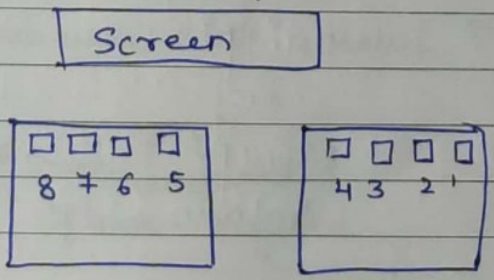
Array → It is a data type defined by user.
↓
Derived type.

Array is used to store multiple values in a single variable.

Syntax : int arr[5];

Contiguous → Element arranged next to each other in side by side manner.

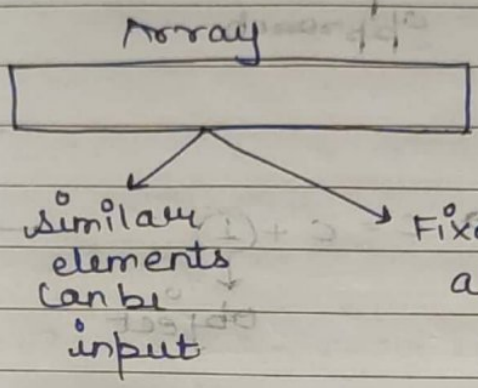
eg:



seat 3,4,5,6 are continuous but not contiguous.

- Structure is used to store data of different types.

Disadvantages of Array:



21/07/22.

FUNCTION

- Function is a self contained block of instructions designed to do some task.
- It has a modular approach. Eg: main(), scanf().
- When a function is called control goes to function definition performs the task and control returns to the calling body.
- When a control returns it may return with a value may not return with a value.

Declaration:

- Function declaration is a promise to the compiler which return type name, type, order & no. of argument. Eg: void abcd(void)
  - void: return type
  - abcd: name
  - (void): argument
- Function may be used in main without declaring ~~the~~ above by using prototype.

return/datatype functionname(int, int);

- It is defined once and called many times.

void() → will not return value.
int → will return value.

- If control returns with a value there should be a receiver.

Defined function:
- Library
- Pre-defined
- User-defined

Eg: int hello(float, int) → It will return only integer type value.

26/07

STRUCTURE

- A structure is a collection of variables of same or different data types.
- We use structures to overcome the drawback of arrays. We already know that arrays in C are bound to store variables that are of similar data types. Creating a structures helps the user to declare multiple variable of different data types treated as a single entity.

Syntax: returntype functionname (argument)
{
Body;
};

eg: int display (void); ← prototype of function
main()
{
display();
};
int display()
{
int i, j;
printf("Enter values to add");
scanf("%d %d", &i, &j);
printf("sum = %d", i+j);
return (i+j);
};

eg: int display (int, int); → ~~Structure prototype~~

main ()

function {

int a, b, c;

c = display (a, b);

printf (" %d", c);

}

int display (int c, int d)

{

return (c+d);

}

Syntax of structure : eg: Struct student

{

int roll;

float avg;

};

main ()

{

struct student S1, S2;

display (S1);

}

This will access the whole structure.

void display (struct student *k)

{

}

Dynamic / Runtime memory allocation:

↳ In this we can add an array index contiguously or delete an array index if not used by the user during run-time of program.

Malloc :- It stands for memory allocation. It allocates / provides a block of memory contiguously.

Syntax : (void*) malloc (size in bytes)

↓

Generic pointer

eg : malloc(20) → It will provide 20 bytes of space contiguously.

→ If space is not available then it will return null.

Pointer : int *p
float *q
char *r

=> • A variable which stores the address of other variable.
• They always return +ve hexadecimal no.
• They can never -ve.

Q.) Why pointer is having a 'data type'?

eg: struct node
{
- -
- -
};

struct node *p = (struct node*) malloc (size of (struct node) x n));

Syntax for malloc.

Calloc : • Allocates the requested memory and returns a pointer to it.
• Malloc function creates a single block of memory of a specific size, while calloc function assigns multiple blocks of memory to a single variable.
• Malloc does not set the memory to zero where as calloc sets allocated memory to zero.

(means contiguous allocation)

27/7

eg:

* struct student
{
int roll;
float avg;
};
main()
{
struct student S1, S2;
S1.roll;
}

member variable → they are by default public (not secured) → access specifier

public is accessable directly

* Class student
{
int roll;
float avg;
};
main()
{
student S1, S2;
}

member variables known as data members.

→ objects of type student

{ In C progg. we follow top to bottom approach }

- Object is the instance of a class.
- It carries the properties of the class.
- It is the instance because when object is created, class gets instantiated that means memory is allocated to non-static data members of the class.

• The function can be a direct member of the class known as member function.

eg:

```
class student
{
    int roll;
    float avg;
    void xyz();
};
```

→ Member function
→ Direct member.

• By default class members are private but structures are bydefault public.

eg:

```
class student
{
    int roll;
    float avg;
};
main()
{
    Student S1, S2;
    S1.roll = 10;
}
```

Inaccessible → Not accessible directly

• An error will be generated in the above program ∵ The class members are bydefault private.

To run the program we need to make the class member public by writing public under class

```
class student
{ public:
  ---
};
```

C++:

Syntax:

eg:

```
#include <iostream>          // input & output stream
using namespace std;
class student
{
    int roll; float avg;     // data members
    public:
    void show()              // Member function (public in nature)
    {
    }
};
main()
{
    student S1;
    S1.show();               // works fine.
    S1.roll;                 // It will generate an error since class member are private in nature
}
```

- Input & Output operation are carried out in C++ with the help of predefined objects & operators.

Input	Output
→ Read.	→ Write.
→ C in object (object to read) (object of istream class)	→ cout object (object to write) (object of ostream class)
→ >> (extraction/read from/get from operator)	→ << (insertion/write to/put to operator).

Syntax:

```
cin >> var
cout << var
```

eg:

```
int c;
cout << "enter the value";
cin >> i;
cout << i;
```

>> → Right shift operator
<< → Left shift operator

C++ program:

(1)

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

int main()
{
    int i, j, sum;
    cout << "Input the nos.";
    cin >> i >> j;
    sum = i + j;
    cout << "TOTAL is" << sum;
    return 0;
}
```

Output:

```
Input the nos. 10
20
TOTAL is 30.
```

28/7

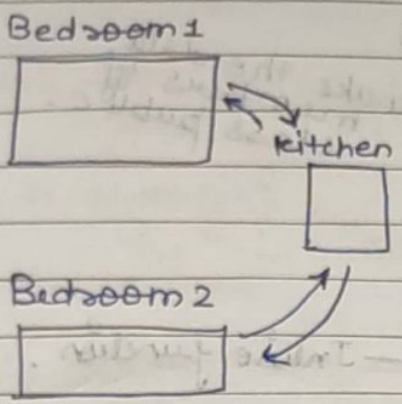
eg:
```
class student
{
    int roll;
    float avg;
    public:
    void input ()
    {
        cout << "input";
        cin >> roll >> avg;
    }
    void output ()
    {
        cout << roll << "\t" << avg;
    }
};
int main ()
{
    student S1, S2; // two instances (objects) created.
    S1. input ();
    S2. output ();
}
```

* endl is the manipulator. It brings the cursor to next inline or we can use \n.

Non-inline

House 1

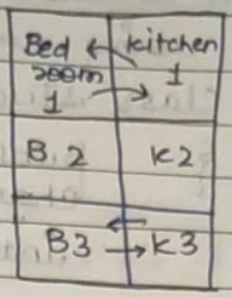
eg:



- Slower
- Less memory (∵ it's defined once)

Inline

House 2



- Faster
- More memory.

- When we define a function in a class it is treated as an inline function it is treated as because compiler may ignore our request and treat it as non-inline function.

- Inline function take more memory because for the respective function call the inline function gets expanded in that line they are expanded there and
- They execute faster as there is no concept of control going and returning.
- We can try to make a non-inline function inline by using the keyword inline before the function.
- Trying to make the function inline is a request not a command or guarantee.
- A function which is defined outside the class is known as non-inline function.
- When there is a loop or large program then it should not be used.

eg:

```
# include <iostream>
using namespace std;
class student
{   public:          <- To make the data members of class public.
    char name [25];
    int roll;

    void input()     <- Inline function.
    { cout << "Enter name";
      cin >> name;
      cout << "Enter roll";
      cin >> roll;
    }
    void output();   <- initialising non-inline.
};
void student :: output()
{
    cout << "Name" << name << "Roll" << roll;
}
```

Non inline function

Passing object as argument :

*this → It is actually a point.

s1.show(s2)
↳ This will print the properties of s1 not s2

```
void show(student k)
{
    cout << "Roll = " << k.roll;
}
int main()
{
    Student s1, s2, s3;
    s1.get();
    s2.get();
    s3.get();
    s1.show(s2);
}
```

↳ Roll of s2 will print.

if we write roll instead of k.roll then s1.show(s2); it will show the roll of s1.

this pointer (*this): is an implicit pointer which stores the address of the object/current calling object when object calls the member function.

- It is a hidden pointer
- automatic pointer
- We can write (this→roll) when we call roll if we want. when we call the member func. then 'this' is written automatically but we can't see it.

§ this is a pointer to class :}

Syntax :

① this → avg

② (*this).avg

↑ if we remove bracket then priority will be given to '.'.

3/8/22

Static Variable :

static int i = 50 ← 'i' will be initialised only once through the code.

- Static data members are stored as part of class not stored as a part of object only one copy ~~of~~ of the static data member is created irrespective of no. of objects. However all objects access same.
- The default initial value is zero. The static data member are ~~have~~ to be defined outside the class by using datatype class name, scope(::) ~~and~~ operator and variable name.
- In C++, we can have static member func. but the static member func. can only access static data. It cannot access non-static.
- Non-static can access both (static & non-static).
- A static ~~not~~ func. can be called as : classname :: static variable ; ← called outside the class.

eg :- int abc :: b ;

{ static data function called inside main ( )
  abc : : input ( );
}

Reference Variable: is an alias name/duplicate name to a variable.

- Any change made in reference also affects to the original variable because it is just duplicate name, no seperate copy is created.

eg: int i = 0;
    int &k = i; { so k is a reference to i }
    k = k + 5
    cout << i;
    Output → i = 15

4/8/22

Function with default arguments :

- In OOP we can create func. with default arguments. But default arguments are to be given at the time of declaration which are assigned from right to left.
- A func. can have all the arguments set to default values.
- If user will not provide any specific value and if the func. is having a default value

then default values are taken.

- If user provides user-defined value and the default value exists, then user defined value supercised (prioritises) default value.

eg: void show (int i = 10) ← default argument
{
}
show (20); → gives 20 as output
show (); → gives 10 " "

eg: void show (int i = 10, float j)
{
}

error will occur since it moves from right to left.

eg: void show (int i, float j = 2.5f)
{
}
show (10, 4.5f); → gives 10 & 4.5 as output

NO error.

Application: EMI

```
void emi(int r=15%)
{
}
```

[∵ rate of interest is fixed for some period of time & when it will be changed then the user will give user-defined argument.]

Encapsulation: It is a property of OOP where together the data members and the member func. are bound under one common type class.

Polymorphism: Exhibiting two or different form (Many forms)

Static or Compile time
- Func. overloading.
- Operator " .

Dynamic or Runtime
- Virtual func.

Function Overloading:
- When we create func. with same name & different argument lists then we say the func. is overloaded.
- Func. overloading is a compile time polymorphism property of OOP.

eg: void show;
void show (int);
void show (int, float);

Q. When a func. is called whether the return type is verified?

Ans. No.

Q. void show (int);
int show (int);

Ans. Error will occur since the func. name & argument is same so it's not overloaded. Although return type is different but it is not verified when func. is called.

```
eg: void show (void)
    {
        cout << a;
    }
    void show (int i)
    {
        cout << b;
    }
    void show (int i, float j)
    {
        cout << c;
    }
    int main()
    {
        show();
        show(20);
        show(20, 4.5f);
    }
```

Output: a
b
c

- While performing func. overloading along with use of default arguments there care must be taken by the user, to avoid ambiguous situation.

eg:

```
void show(void);
{
    cout << a;
}
void show(int)                    ... line no. 8
{
    cout << b;
}
void show(int i, float j = 4.7f)  .... line no. 12
{
    cout << c;
}
int main()
{
    show();
    show(20);        → This leads to ambiguous situation ∵ line no. 8 & 12 both can give output for this line, so its ambiguous.
    show(20, 4.5f);
}
```

Dummy Arguments: It is used when two overloading func. (func. with same name) having same arguments so we use dummy arguments to differentiate b/w the two.

eg:

```
area(float r) → circle
{
    3.14*r*r;
}
area(float a, float b) → rectangle
{
    a*b;
}
area(float a, char ch) → square
{
    a*a;
}
```

dummy argument

10/8/22

Private Member Function :

```
class student
{
    ==
    public:
    void get()
    {
    }
    private:
    void show()
    {
    }
};
int main()
{
    student s1;
    s1.get();  ✓
    s1.show(); ✗ // can't access as it is
    s1.roll(); ✗ private member function.
}
```

- Private member function can be called from another public member function.

```
eg: void get()
    {
        show();
    }
```

Scope of a variable :

-
```
int main()
{   int i=20;
    int i=30;
}
```
it will show error since 'i' is declared two times.

- int main()
{
    int i = 20;
    {
        int i = 30;
    }
    cout << i << endl;
}
// Output will be → 20.

- int main()
{
    int i = 20;
    {
        int i = 30;
        cout << i << endl; //prints 30
    }
    cout << i << endl; //prints 20
}

- int i = 10; // i is global variable to main function.
int main()
{
    int i = 20
    {
        int i = 30;
        cout << i; //prints 30
    }
    cout << i; //prints 20
    cout << ::i; // prints the value of global variable i.e. 10.
}
↳ (global variable is called by ::i)

Friend function :

- A friend function is non-member function of the class.
- The friend keyword is used to make a function friendly to the class.
- The friend function can either be declared or defined in any visibility of the class without affecting its meaning.
- If we declare the friend function within the class & define it outside then it is defined like a normal function. (without class name scope resolution operator).
- A friend func. is to be called directly by its name with its argument list.
- A friend func. cannot access the member of the class directly.
- However, if we pass the object as its argument then the friend func. can access the properties for that instance only.

This violates the principle of OOP :-

1. Security principle.
2. Data hiding → private
3. Data Abstraction.

accessibility

↳ accessibility says only member func. can access but here a non-member func. accesses the data.

eg:

```
class
{   int i;
    friend void abcd();
};
void abcd()
{   cin>>i;
}
```

non-member func.

↳ defined within the scope of class but it is not a member func.

~~X~~ → inaccessible directly

23/8/22 eg:

```
class student
{   int roll;
    public:
        void get() { cin>>roll; }
        void show() { cout<<roll; }
    friend void xyz (student y)
    {   ~~cout<<roll; // not accessible~~
        cout<<y.roll;
    }
main()
{   student s1, s2;
    s1.get(); s2.get();
    xyz(s1);
}
```

→ Now friend func. can access the s1 data.

eg: class
{
=
=
=
friend void xyz (student); ← declared ~~defined~~ inside the class
};
void xyz (student &k) ← ~~and called~~ defined outside
{
cout << k.roll;
}

24/8/22

* Same func. name (eg: void get()) can be given under different scope (class).

* A member of a class can be a member to that class only.

* A friend can be friend to different class also.

* Making a member func. friendly to another class by → [friend void abc :: xyz ( )]
abc ← classname
xyz ← func. name

* friend class abc;
↳ by this line all member func. will become friend to other class.

Constructor

It is a special member func. which is used to construct the object. The name of constructor func. is same as class name having no return type not even void. It is called automatically when the object of the class is created. A constructor can either be declared or defined in the public visibility of the class. A constructor is having no return type because it is called automatically and there is no calling body to which it will return value.

Constructor are of three types:

1. No Argument / Zero Argument Constructor.
2. Parameterized Constructor. (cons. having argument)
3. Copy Constructor.

eg:

```
Class student
{
    int roll, age;
    public:
        student ()  //constructor.
        {
            cout <<"hello" <<endl;
        }
};
int main ()
{
  Student S1, S2;
}
```

Output
hello
hello

eg: class student

```
{
    int roll, age;
    public:
        student()
        {
            cout<<"enter roll & age";
            cin>>roll>>age;
        }
};
int main()
{
    student S1, S2;
    student S3;
}
```

Constructor declared inside & defined outside the class:

```
class student
{
    int roll, age;
    public:
        student();
};
student::student()
{
    cin>>roll>>age;
}
int main()
{
    student S2, S3;
}
```

Constructor Overloading:

```
class student
{
    int roll, age;
    student ()
    {
        roll = 10;
        age = 20;
    }
    student (int k, int l) //parameterized
    {
        roll = k, age = l;
    }
}
int main ()
{
    student S1;
    student S2 (11, 22);
}
```

Or

```
int main ()
{
    student S1;
    int x, y;
    cout << "enter roll & age";
    cin >> x >> y;
    student S2 (x, y);
}
```

S2 → dynamic initialization of object.

Name => Agnim Srivastava
Roll no. => 2105259
Section => CSE-04

Assignment-1

1) Write a program to create class "basic" that stores the loan amount in rupees. Create another class "interest" that stores the number of years and rate of interest as its private data member. Calculate the simple interest and compound interest after providing input by using necessary function.

```
#include <iostream>
#include <math.h>
using namespace std;
class interest;
class basic
{
    public:
    int loan;
    friend void simple_interest(basic s, interest i);
    friend void compound_interest(basic s, interest i);
    basic()
    {
        cout<<"Enter the loan amount in rupees"<<endl;
        cin>>loan;
    }
};
class interest
{
    private:
    int t;
    float r;
    friend void simple_interest(basic s, interest i);
```

```
    friend void compound_interest(basic s, interest i);
    public:
    interest()
    {
        cout<<"Enter the number of years in which you
              want to repay the loan"<<endl;
        cin>>t;
        cout<<"Enter the rate of interest"<<endl
        cin>>r;
    }
};
void simple_interest(basic s, interest i)
{
    float si;
    si=(s.loan*i.r*i.t)/100.0;
    cout<<"Simple Interest="<<si<<endl;
}
void compound_interest(basic s, interest i)
{
    float a=pow((1+(i.r/100)), i.t);
    float ci=(s.loan*a)-s.loan;
    cout<<"Compound Interest="<<ci<<endl;
}
int main()
{
    basic s;
    interest i;
    simple_interest(s,i);
    compound_interest(s, i);
    return 0;
}
```

Output :

```
Enter the loan amount in rupies
20000
Enter the number of years in which you want to repay the loan
5
Enter the rate of interest
8
Simple Interest = 8000
Compound Interest = 9386.57
```

2) Write a program to create a class "student" that stores the name, roll, age of a student. Create another class "mark" which stores the three subject marks and grade. Input the details of a student by using necessary member function. Print all the details by making a member function of "mark" class friendly to "student" class.

```
#include <iostream>
using namespace std;
class mark;
class student
{
    string name;
    int roll;
    int age;
    public:
    void inputdetails()
    {
        cout<<"enter name of the student-";
        cin>>name;
        cout<<"enter roll number-";
        cin>>roll;
        cout<<"enter age-";
        cin>>age;
    }
    friend void display(student, mark);
};
class mark
{
    int marks[3];
    char grade;
```

```
public:
void inputmarks()
{
    cout<<"enter grade - ";
    cin>>grade;
    for(int i=0;i<3;i++)
    {
        cout<<"enter marks for "<<(i+1)<<"subject-";
        cin>>marks[i];
    }
}
friend void display(student,mark);
};
void display(student S1, mark m1)
{
    cout<<"\n student details\n";
    cout<<"Name - "<<S1.name<<endl;
    cout<<"Roll - "<<S1.roll<<endl;
    cout<<"Age - "<<S1.age<<endl;
    cout<<"Grade - "<<m1.grade<<endl;
    for(int i=0; i<3;i++)
    {
        cout<<"Marks - "<<m1.marks[i]<<endl;
    }
}
int main()
{
    student S1;
    S1.inputdetails();
    mark m1;
    m1.inputmarks();
    display(S1,m1);
    return 0;
}
```

Output :-

```
enter name of the student - agrim
enter roll number - 59
enter age - 18
enter grade - A
enter marks for 1 subject - 99
enter marks for 2 subject - 95
enter marks for 3 subject - 98

student details
Name - agrim
Roll - 59
Age - 18
Grade - A
Marks - 99
Marks - 95
Marks - 98
```

30/8/22

Dynamic Memory Allocation:-

Dynamic Memory Allocation prevents memory wastage.

function	C++
malloc()	operators
calloc()	→ new
realloc()	→ delete
free()	

syntax: new datatype

```
int *p=new int ;
cout<<"enter value";
cin>>*p;
cout<<*p;
```

```
int *p;
cout<<"enter value";
cin>>*p;
cout<<*p;
```

// Segmentation fault will come bec. int *p=new int; not written.

```
int *p= new int (5);
```

```
int *p = new int[5];
for(int i=0; i<5 ;i++) // Giving 5 values to pointer.
{
    cin>>*(p+i);
    sum = sum +*(p+i);
}
for(int i=0;i<5; i++)      //print the 5 values.
{
    cout<<*(p+i)<<endl;
}
```

```
char *p = new char[n];

cout << "enter name";
cin >> p;
cout << p;
```

delete:

Syntax→ delete variable_name;

eg:-

```
int *p = new int;
=====
delete p;
```

eg:-
```
int *p = new int[n];
=====
delete []p;
```

The delete operator deallocate the memory which is allocated or pointed by the variable. By deleting actually the memory possession is released. Neither the variable nor the memory is deleted.
A good programmers habit is to deallocate the memory if not used. So that during the execution of the program the released memory may

Q. When constructor is called?
A. When object is ~~called~~ created it is automatically called.

possibly be allocated if requested.

Dynamic Constructor:

```
class student
{
    char *name;
    public:
    student (char p[])
    {   int n = strlen(p);
        name=new char[p●];  // allocates exact memory as required by name.
        strcpy(name, p);
    }
    void show()
    {
        cout<<" name is: "<<name;
    }
};
int main()
{
    char arr[20];
    cout<<" enter name";
    cin >> arr;
    student s1 (arr);
    s1.show();
    cout<<" enter name";
    cin >> arr;
    student s2(arr);
```

06/09/22

Copy constructor:

User can write user-defined copy constructor to copy the property of one object to another. The copy constructor takes the arguments of its own type which has to be a reference not a value. If it is received by value then it will become recursive and program will not work.

Reason:- When user writes user-defined constructor then it possess more priority than system defined constructor. So in a copy constructor if the receiver is by value then in order to copy it invokes repeatedly the user-defined copy constructor.

```
class student
{
    int roll, age;
    public:
     student () {  }
     student (int i, int j)
     { roll = i; age = j; }
     student (student & k)
     {
         roll = k.roll;
         age = k.age;
     }
};
```

} copy constructor.

```
main()
{
    student S1;
    student S2 (1,19);
    Student S3 (S2);
}
```

Destructor:

Destructor is a special member function which is used to deallocate memory constructed by the object. The name of the function is same as class name preceeded by tilde symbol. Destructor is having no argument and no return-type. Destructors cannot be overloaded since they don't take any argument. Destructor is called automatically when control goes out of the scope for which the memory was constructed. The destructor are called in reverse order of the way the memory is allocated.

```
class student
{
    int roll;
    public:
    student() {roll=1}
    Student (int i) {roll=2}
    Student (student & k) {roll=3}
    ~student() { cout<< roll <<endl; }
};
main()
{
    student S1;
    Student S2(5);
    Student S3(S2);
}
```

constructor

Destructor

output: 3
2
1

```
main()
{
    Student s1;
    Student s2(5);
    {
        Student s3(32);
    }
    cout << "hello" << endl;
}
```

Output :-
3
hello
2
1

07/9/22

Destructor called outside class :-

```
~Student(); // declared inside the class.
};
Student::~Student(){cout << roll << "destroyed";
// defined outside the class.
```

Note → Always define zero argument constructor in your program.

```
#include <iostream>
using namespace std; ~~int roll; public:~~
class student(){~~cout << "zero argument constructor"~~
~~class~~ int roll;
    public:
student(){roll=1; cout << "Zero arg cons " << endl;
student(int k){roll=2; cout << "Parameterised cons "
```

~student() { cout<<roll<<" destroyed "<<endl; }
};

- int main()
{
student *p = new student; // an unnamed object is created
return 0;
}

Output:
zero argument constructor

- int main()
{
student *p = new student;
delete p;
return 0;
}

Output:
zero argument cons
1 destroyed.

- int main()
{
Student *p;
{ p = new student;
delete p; // it will go to destructor.
}
p = new student(5);
delete p;
return 0;
}

Output:
zero argument constructor
1 destroyed (//because of delete.)
parameterised constructor
2 destroyed.

```
int main()
{
    Student *p;
    {
        p = new Student;
    }
    p = new student(5);
    delete p;
    return 0;
}
```

Output:
zero argument constructor
parameterised constructor
2 destroyed.

* As soon as user-defined constructor is made the compiler suppresses the system-defined constructor.

* In call by reference as 'k' makes duplicate and not space then the constructor as well as destructor is not called as there is no construction of k takes place.

Destructor initialised inside the class & defined outside the class :

```
class student
{
    ===
    ===
    ===
    ~student();
};
student::~student()
{
    cout<<" destructor "<<endl;
}
int main()
{
    ==
    ==

}
```

* When ~~an~~ unnamed instances are called then the memory is not deleted automatically.

eg:
```
int main()
{
    student *p = new student //unnamed instance is made.
    delete p;
    return 0;
}
```

Proof:

```
* int main()
{
    Student *p;
    {
        p= new student;
        delete p;
    }
    p= new student;
    delete p;
    return 0;
}
```

Output:
Zero argument constructor
destructor
Zero argument constructor
destructor.

```
* int main()
{
    Student * p;
    {
        p= new Student;
    }
    p= new student;
    delete p;
    return 0;
}
```

→ Delete nhi hua 'p' upha pr automatically

Output:
Zero argument constructor
Zero " "
destructor.

13/9/22

Inheritance (Reusability) :-

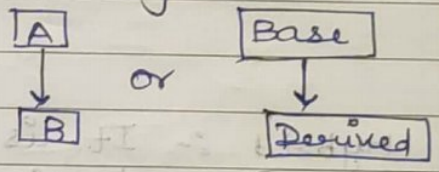
Inheritance is the property of OOP where one class property is inherited to another class. It can also be defined as deriving a new class from another existing class.

The class whose property is inherited is called base class and the class which receives the property from base class is called derived class.

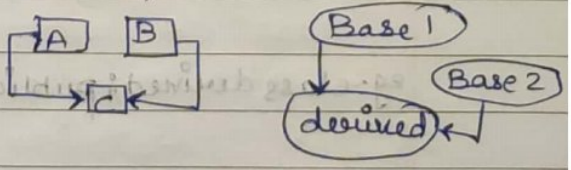
base (Parent)
↓
derived (child)

Types of inheritance :

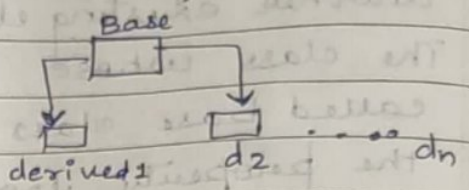
① Single Inheritance :- When one class is derived from another class is known as single inheritance.



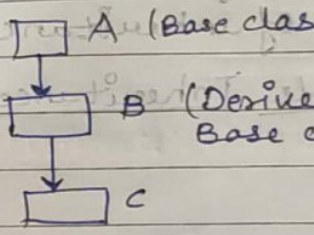
② Multiple inheritance :- When a class is derived from two or more classes, it is known as multiple inheritance.



③ Hierarchical inheritance : When two or more classes are derived from one base class.



④ Multi-level Inheritance : when one class is derived from another derived class in two or more level.



⑤ Hybrid :- It is a combination of two or more basic type of inheritance.
eg: Multi-level and single in one case.

Syntax:
class derivedclassname : visibility baseclass name

eg: class derived : public base

the visibility of the mode can be public, private or protected which explains the final visibility of base class inherited members in derived class.

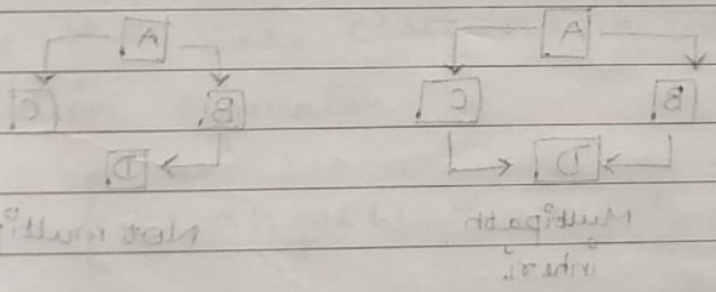
* private member never inherited.

* The significance of protected access specifier is they are secured like private and inheritable like public.

← mode of inheritance →

Base class members	private	protected	public
private	×	×	×
protected	private	protected	protected
public	private	protected	public

Class base
{



14/9/22

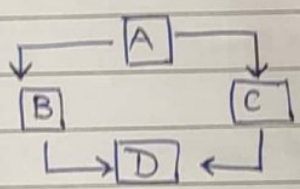
Note :- If a data member is private ~~there~~ in base class then make member function of this class so that derived class can ~~access the~~ access the member function.

eg: class a { }; class b : public a { }; class c : public b { };

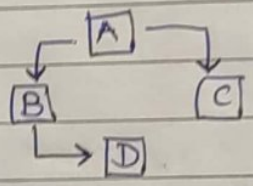
Multilevel inheritance

Multipath Inheritance :

ultimate base class & ultimate derived class pres



Multipath inheri.



Not multipath inheri.

- In multipath inheritance there is possibility of having multiple copies from the ultimate base class to ultimate derived class through multiple paths. This leads to ambiguity.
- This ambiguity is resolved by making the base class virtual at the time of inheritance.
- The keyword virtual is used either before or after the inheritance mode so the base

class becomes the virtual base class & compiler ensures a single copy from the ultimate base class being available in the ultimate base class in multipath inheritance

Function Overriding:-

In inheritance base class inherited member function can have the same signature with the derived class member function. So whenever we try to call the member func. through derived class object, always the derived version gets executed immediately. As it possess more priority. This is known as func. overriding. However, if we want to call the base version you have to call explicitly by using the base class name with scope resolution operator.

```
b1.a::get();  <- To call the derived class member.
```

15/9/22

Base Initialization :-

- class student
{
int i, j;
public:
Student (int x, int y) [or] student (int x, int y) : i(x)
{ i = x; j = y;
↑ first this will be initialised. then this is initialised.
}

- class student
{
int i, j;
public:
Student (int x, int y) : j(i), i(y)
{
}
}

first 'i' will be assigned value of y; 'i' is initialised first.

We can initialise the data members of a class with the initialisation list in a constructor function.
The initialisation of the data members happens the way they are declared in the class irrespective of the initialisation order.

11/10/22

Constructors in inheritance :-

```
class base
{
    public:
    base ()
    {
        cout << "1" << endl;
    }
    base (int k)
    {
        cout << "2" << endl;
    }
    ~base()
    {
        cout << " base class destructor" << endl;
    }
};
class derived : public base
{
    public:
    derived () : base()
    {
        cout << "3" << endl;
    }
    derived (int k) : base()
    {
        cout << "4" << endl;
    }
    ~derived ()
    {
        cout << " derived class destructor" << endl;
    }
};
```

this is interpreted by the computer because of public base.

:base() will be written automatically

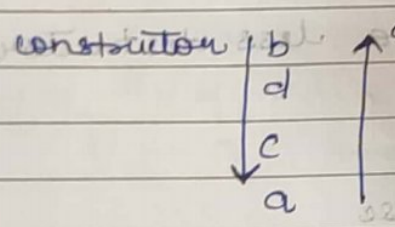
To directly call the parameterised derived class we have to give an argument in base i.e. base(100) here.

If we'll give an argument in main only then first base class with ~~two~~ argument will be called & destroyed then parameterised will be called.

```
int main ()
{   derived d1 ;
    derived d2(10);
    derived d2;

}
```

→ In inheritance when the object of the derived class is created the base class zero argument constructor is invoked first by default.
The base class parameterised constructor is not invoked implicitly, it has to be called explicitely from the derived class constructor.



12/10/22

Dynamic Binding:

Polymorphism

Static
(compile time)
(Early time)
- Func overloading
- Operator overloading

Dynamic
(Run time)
(Late time)
- virtual functio

```
class base
{
    public :
      void get()
       {
          cout<<"base class get method"<<endl;
       }
    virtual void show()
       {
          cout<<"base class show method"<<endl;
       }
};
class desired : public base
{
    public :
    void get()
     {
        cout<<"derived class get method"<<endl;
     }
    void show ()
     {
        cout<<"derived class show method"<<
                                        endl;
     }
};
int main()
{
    desired d;
      d.get();
       d.base::get();  → called explicitly
       base *ptr;
       ptr=&d;
       ptr→get();      → base class invoked
       ptr→show();     → desired class invoked
```

this will give priority to its type

Note:
In inheritance, the base class pointer can point to derived class object the reverse is not true.
Dynamic binding or runtime polymorphism is achieved when a base class pointer is pointing to derived class object & trying to access a virtual function having same signature in base & derived.
The binding to respective version of the function is decided in late or during runtime so it also known as late binding.

13/10/22

* We can't make constructor virtual but can make a destructor virtual.

Note: In dynamic binding, when we use virtual function, a virtual look up table is created and binding to a virtual func. through base class pointer pointing to derived class object occurs by mapping the func. virtual pointer in the virtual look up table. Constructor being responsible for the construction of instance it cannot be made virtual.
A Destructor can be made ~~virtual~~ virtual. In fact making a destructor virtual is actually a need in order to prevent memory leak.

Pure virtual function;
Syntax → virtual void show() = 0;

Note :

A virtual func. is used to support the feature of dynamic binding, so a base class virtual func. can be made as pure virtual func.

Syntax : virtual return_type function_name() = 0 ;

When a class is consisting a pure virtual func. then that class is known as abstract class.

Operator Overloading :

18-10-22

Operator
- Unary: ++, --, +i, -i
- binary: *, /, >, ff, f, +, -
- Ternary: ?:

In a+b or a-b, +, - are binary but in +b or -b, +, - are unary

* >> (Right shift)
* << (Left shift)

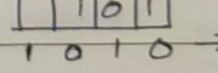
* cin >> i
  - ↑ extraction / Read / get from
* cout << i
  - ↑ insertion / write / put to.

• cin / cout is an object of predefined class iostream.

a >> b → a/2^b

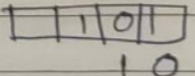
a << b → a * 2^b

eg: 5<<1



1 0 1 0 → 10

5>>1



1 0

Note: Operator overloading is the feature of compile time polymorphism where some of the existing operators can be used with user defined data type (object).
Using operator overloading as per the requirement a different approach is justified but the original meaning of the operator is never changed. In order to achieve the operator overloading the class must have a member func. or friend func. depending upon the type of overloading. Some of the operators out of the total existing operators can only be overloaded. Using operator overloading no newly seperate operator is created. Certain operator which are not overloaded are sizeof, ternary (?:), *, scope(::), dot(.).
(indirection)

Using member func.:

```
Class test
{
    int i;
    float j;
    public:
    void get()
    {
        cin>>i>>j;
    }
    void show()
    {
        cout<<i<<"\t"<<j<<endl;
    }
    void operator+(int k)
    { i=i+k; j=j+k; }
```

~~void show()~~
~~{~~
~~cout<<i<<j;~~
~~}~~

main()
{
test t1, t2;
t1.get();
//t2.get();
t1.show();
//t2.show();
t1+5;
t1.show();
}

	i	j
t_1	5	2.5
t_2	10	3.0

t1+5; → member func. → t1.operator+(5)
→ friend func. → operator+(t1, 5)

Output:

5 2.5
5 2.5
10 7.5

19/10/22 Using friend func.:

friend ~~void~~ operator + (~~test~~ a, ~~test~~ b) or we can pass value by reference also.
{
a.i = a.i + b.i;
a.j = a.j + b.j;
}
};
int main()
{
—
operator+(t1, t2);
t1.show();

we can use only 'k' also.

```
void operator + (test & k)
{
    i = i + k.i;
    j = j + k.j;
}
main()
{
    test t1, t2;
    t1.operator + (t2);
    t1.show();
}
```

```
class test
{
    =
    friend void operator + (test & k, test & l)
    {
        k.i = k.i + l.i;
        k.j = k.j + l.j;
    }
};
main()
{
    test t1, t2;
    operator + (t2, t1);
    t2.show();      // added value will come
}
```

	i	j
(l) t1	2	3
(k) t2	4	5

If we remove '&' from k :

```
class test
{
    int i;
    float j;
    ---
    ---
friend void operator+(test k, test &l)
{
        k.i = k.i + l.i;
        k.j = k.j + l.j;
}

main()
{
    test t1, t2;
    operator+(t2, t1);
        t2.show();    // added value will not come since k is ~~this~~ ~~also~~ seperately created because of which ~~who~~ the value of t2 will not ~~come~~ change.
}
```

↑ this rule is only for friend func.

```
class test
{
    int i;
    float j;
    ---
    ---
friend void operator+(test &k, test &l)
{
        k.i = k.i + l.i;
        k.j = k.j + l.j;
}
```

```
main()
{
    test t1, t2;
    operator+(t2, t1);
    t2.show();  // added value will come since we are calling t2 to show the value of t2 and k is its duplicate/alias name ∵ it is called by ref. so it will work.
}
```

* If we will write both i.e., the member func. & the friend func. the ambiguous situation will occur.

* this pointer is not accessible in friend function ∵ this pointer is not used outside non-static member function.

20/10/22

Overloading Greater than (>):

Member func. :

s1 > s2

operator>(s1, s2) (friend func.)

s1.operator>(s2) (member func.)

```
Class student
{
    =
    public :
    =
student operator>(student k)
    {                                  (Compiler will interpret it this.arg or s1.arg.)
        if (arg > k.arg)
        {
            cout<<" student s1 marks are higher";
                                       or return *this;
        }
    }       else { return k; }

main ()
{
    student s1, s2;
    s1. operator>(s2);
}
```

Friend func. :

```
friend student operator>(student a, student b)
    {
        if(a.arg > b.arg)
        {       return a;
        }
        else { return b; }
    }
```

* Member function are called by "object".

* There are certain situations where only member func. or only friend func. are used.

* In binary operator friend func. takes 2 argument while unary takes 1 & member func. in binary takes 1 argument while unary take no argument.

Unary minus (-) :

Friend func.

```
class test
{
    int a;
    float b;
    public:
friend void operator - (test k)
    {
        k.a = -k.a;
        k.b = -k.b;
    }
};
int main()
{  test t1;
   -t1;
}
```

Member func.

```
class test
{  int marks;
   public:
   test()
   {  cout << "Enter marks";
      cin >> marks;
   }
   void operator-()
   {  marks = -marks;
      cout << marks;
};
int main()
{
   test t1;
   -t1;
} return 0;
```

bit wise AND (&) :

(!)

Member func.:

```
class test
{
   int a;
   =
   void operator &(test k)
   {
      a = a & k.a;
   }
}
```

(True && True = true)

5 && 6 = 1

5 & 6 = 101 (5)
110 (6)
100 (4)

```
int main()
{
    t1 & t2;
    t1.operator&(t2);
    operator&(t1,t2);
}
```

Friend func.:

```
class test
{
    int a;

    friend void operator&(test &k, test l)
    {
        k.a = k.a & l.a;
    }
};
```

26/10/22

Unary increment & decrement (++, --)

i++	++i
post increment	pre increment
i=20, j;	i=20, j;
j= i++;	j= ++i
cout << j; → 20	cout<<j; → 21

```
class abc
{ int i;
```

public:

get() | show() → any one.

```
abc operator ++()
{
    ++i;
    return *this
}
main()
{
    abc a1, a2;
    a1.get();
    a1.show(); → 20
```

a2 = ++a1

→ member func:
a1.show() → 21
a2.show() → 21

→ friend func:
operator++(a1)

```
//Now friend func.
friend abc operator ++(abc &k
{
    ++k.i
    return k;
}
//Another way
return abc(++k.i);
```

Postfix:

a2 = a1++;

→ member func.
a1.operator(0)

→ friend func.
operator++(a1, 0)

```
abc operator ++(int k)
{
    abc y
    y.i = i++;
    return y
}
```

If you write `return *this` → this will return 22 but we require the return value 21 to justify the answer.

Note: As the unary ++, -- operator exhibits two different mode, i.e., pre and post form so 1 default argument of integer type is taken as an argument in post form inorder to distinguish between ~~pre~~ is preform and postform the default value 0.

27/10/22

Overloading insertion & extraction operators:

`int i,j;`
`i>>j;`
or
`i<<j;`
	`cin>>i;`
	`cout<<j;`
		`Student S1;`
		`cin>>S1;`
		`cout<<S1;`

input stream
istream

output stream
cout<<

a op b friend func.

member func.
a operator op(b).

friend operator op(a,b)

cin >> S1

member → cin.operator op(S1) ✗

friend → friend operator op(cin, S1)

Note: Cannot be overloaded using member func. bec. cin is not the object of class student.

```
class student
{
    int roll;
    char name[15];
    public:
friend istream &operator >> (istream &k, student &t)
{
    k >> t.name;
    k >> t.roll;
    return k;
}
friend ostream &operator << (ostream &k,
                             student &t)
{
    k << t.name;
    k << t.roll;
    return k;
}
main ();
{
  student S1;
  cin >> S1;
  cout << S1;
}
```

Assignment operator = (Binary operator)
Cannot be overloaded with friend function

```
void operator = (student k)
{
    strcpy (name, k.name)
    strcpy (roll, k.roll)

main()
{ Student S1, S2;
  S2 = S1;
  S2.show();
  Student S3 (S2);
  Student S4 = S3;
```

Overloading array index operator.

```
class student{
    int arr[5]
public:
    void operator[](int k){
        for(int i=0; i<k; i++){
            cin>>arr[i];
        }
```

```
main(){
    int arr[5];
    student s1[5];
    s1[5];
```

Overloading function bound operator.

```
void operator()(int y){
    arr[2]=y;
}
```

```
student s2(4);
s2(8);
```

Overloading new & delete operator.

```
void * new(size_t, size)
{
    void *p = malloc(size);
    return p;
}
```

```
student *s = new student(0)
```

Writing new only calls recursively, so writing malloc instead. But you want to use new.

```
void *new(size_t, size){
    void *p = ::operator new(size);
    return p;
}
```

```
void operator delete (void *ob){
    free (ob);
}
```

```
main(){
student *s1 = new student;
delete s1;
```

Explicit Typecasting.

```
class student{
    int roll;
    float avg;
    char section;
public:
    student (int k){
        roll = k
    }
```

```
int main(){
    student s1;        (zero arg conv.)
    s1 = 5;            (deallocate + parame)
```

Note: There are three types of explicit typecasting allowed
(i) System def. = user def.
(ii) user def. = system def.
(iii) One user def. = Another user def.

In order to convert, system def. type to user def. class, the class must be having parameterized constructor, that accepts the system defined datatype as its argument.

In order to convert user def. type object to system def. type, the class must be having casting operator function which is having no return type and

no argument.

```
class student{
  - - -
  - - -
public:
  operator int (){
    return roll;
  }
```

```
main(){
student s1;
int k = s1;
```

no writing return type

As soon as the parameterized constructor is called while converting system def. type to user def. object, the previous got destructed/totally get flushed out and another new object with same name is constructed.

⇒ If the par. constructed is not defined, it will get an error mentioning not finding operator = function

⇒ If both assignment operator overloading function and parameterised constructor is present, operator = () function gets more priority.

Method-1

```
class student{
  int i;
public
  student(Employee k){
    i = k.reply();
  }
```

```
int main(){
  emp e1;
  Student s1;
  int k = s1
  s1 = k
  s1 = e1
```

casting Operator ⇒
parameterised constructor ⇒

Date: / /

Method-2

```
class student{

operator emp(){
    return emp(i,j,...);
}
```

```
main(){
int k=s1
float k=s1
e1=s1
```

① Perform the below operator by writing casting operator fun^n.
② Also use the constructor method.

③ WAP to create a class rupee as its private dm. Create another class dollar that stores the respective amount in dollar as its pdm. Input amount of dollar from keyboard and convert res. amount in rupee and store in rupees class and display it

Standard io device ⇒ keyboard, monitor.

NATARAJ

Date: __/__/____

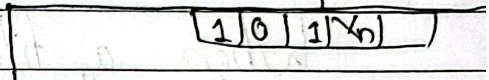
Working with File.

Read	Write
input	output
get	put
extract	insert.

output

to flush input / buffer in program, use fflush(stdin).

Note: Program takes input from i buffer through i buffer



Library for file handling ⇒ fstream

cin >> roll >> grade
take roll = 5 but sometime it take \n as a character and give garbage value

Code:
```
main(){
    ofstream out;
    out.open("abcd.txt");
    out << "hello";        → writing hello to file abcd.
}
```

open func^n method.

const. method
```
{ ofstream out("abcd.txt");
  out << "hello";
}
```

ofstream = class
out = object

out << var_name; [display (var_name) in file].

In order to interact with file, user has to create user defined input stream (for read op. from file) and output stream (for write op. to file). a file can be opened using two method ① constructor method which is helpful when we want to interact with on single file only. ② Open func^n method which is helpful when we want to interact with two or more file using same stream.

Page No.: _____

Note: 1 stream can be used to point one file at a time only for read write operation. In order to interact with another file, we have to close the file and need to point again to req. file.

(#) Write operation:

When a file is opened in write mode, if the file exists, all the previous contents are erased and the pointer points to the first location. index = 0.
If the file doesn't exist, a new file is created and pointer points to the first location.

(#) Read operation

When we want to read a content from a file a pointer points to the first location, if file exist.
If file doesn't exist, it returns NULL.

To terminate pointer pointing to file, out.close();

Code:

```
ofstream out("abcd.txt");          => out object pointing abcd
out<<name;                         => ~~cin~~ name to abcd.txt
out.close();                       => out becoming free
ifstream in("abcd.txt");           => in object ~~creating~~ pointing abcd
char y;
cin>>y;                            => taking input from keyboard
out.open("def.txt");               => out pointing new file def
out<<y;                            => Printing y to file def.
}
```

(a) The end of file can be detected using two methods
(i) Using EOF function. It returns 0 when it finds a character and returns 1 when it reaches end of file.
(ii) Using ifstream object It returns 1 when it finds a character and 0 at the end of file.

Code:
```
main(){
  ifstream in;
  in.open("abcd.txt");
  char ch;
  while(in.eof()==0){
    cout<<ch;
    in>>ch;
  }
```
```
while(in){
  cout<<ch;
  in>>ch;
}
```

in doesn't read blanck space. So use method in.get(ch) for proper input.

Note: cin also doesn't read blank. So Use cin.getline(ch, size). Size includes \0 also.

Tellg & Tellp / File Pointer (know the position of pointer)

Tellg() and tellp() function are used to return the correct position of the file pointer. The tellg() function is called by ifstream class object which returns the current position of get function and tellp() by ofstream class object which returns current position of put pointer.

```
ofstream out("file1.txt");
i = tellp();          ⇒ 0
out << "abcde";
i = tellp();          ⇒ 5
```

NATARAJ

Date: __/__/____

Random Access.

seekg()
seekp().

```
fstream in("myfile.txt");
in.seekg(3, ios::beg);
```

⇒ directly move 3 step from the beginning.

```
in.seekg(0, ios::end);
int i = in.tellg();
cout<<i
```

// gives index of last but if you have written in.tellg(ch) it will read ch and move the one pointer ahead.

Once you fetch / read / write from the file, pointer automatically move one step forward.

KIIT - UNIVERSITY

Methods, Constructors

File Modes

1. Read
2. Write
3. Append

ofstream out;
out.open("abcd.txt", ios::out) → default argument.
→ erase content if existing file.

If you want to open a file in append mode to add new content to existing file, we have to explicitly open it in the append mode (ios::app) will override the (ios::out)

Case 1- If file doesn't exist, new file is created, pointer point to 0 index.

Case 2- If file exist, file is opened, previous content if any are not erased and pointer point to the last index.

Binary File.

By default .txt

.bin = binary fi

```
int y = 100;
out.open("abcd.bin");
```

Syntax: out.write((char *)add. of variable, size of variable);

Code: out.write((char *)&y, sizeof(y));
out.close();

ifstream in;

```
in.open("abcd.bin");
in.read((char *)&temp, sizeof(temp));
cout<<temp;
```

Imp Reading from class object and writing to binary file.

```
out.write((char *)&S1, sizeof(S1));
in.read((char *)&S2, sizeof(S2));
S2.show();
```

reading from S1 converting to bin, pulling to S2.

Generic Programming using Templates.

Syntax: `template <class generictype>`

The generic prog. is implemented using templates. It helps to create a common datatypes that can receive various datatype which makes the program compact. If we want to receive different types of data then we have to create respective numbes of generic type

Eg.
```
template <class T>
void show(T i)
{
    cout<<i;
}

int main{
show(2);    //o/p=2
show(2.5);  //o/p=2.5
```

Note:

One generic type can take one type of data in one instance of time.

```
void show(T i, T j){
    - - - -
}
```

show(4,4) ✓
show(2, 2.5) ✓
show(4, 7.5) ✓
show(7.5, 4) ✗

If you want different type at same instance, you have to create those many times of template, you have different type of argument.

```
template (class T1, class T2){
void show(T1 i, T2 j)
{ - - -
}
```

show(5, 4.5);
show(4.5, 5);

(*) Swapping two numbers.

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

- Exact datatype
- Typecasting
- Default arguments
- Generic Programming

```
int main(){
int x, y
cout<<x<<y;
swap(x,y);
char a,b;
cout<<x<<y
swap(a,b)
```

both should be written

Copy content 1 f → Another f in reverse.

```
in.seekg(0, ios::end);          ← goes to last index.
int k = in.tellg();             ← give value of last index.
for(int i=1; i<=k; i++){
    in.seekg(-i, ios::end);
    in.get(ch);
    out<<ch;
}
```

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

```
main(){
Student <int, float> (2, 9.5f);
~~void reverse~~
```

Note: When we create a class template, we must mention the respective type of data to be accepted as a generic type. According memory is allocated to the object at the time of creation. So the class template are created while creating a function template, as compiler already knows pre-def. datatype, this specific allocation of datatype are not required.

Defining ~~fun~~ const. outside class, write again,

```
template <class T1, class T2>
Student <T1, T2> :: student (T1 x, T2 y){..
```

Exceptions. (not on errors).

Exceptions are runtime anomalies that can happen during the execution of the program.
Eg: - dividing by 0
- array bound exceeds.

The exceptions should be managed by the user so that the end user can know about the runtime exceptional situation. Exceptions in C++ is managed by three keyword - try
- catch (throws)
- throw.

The part of the code which is likely to generate exception is enclosed within a try block. When the exception is detected it has to be reported by using the keyword 'throw'.

Syntax:

```
throw exception;
or, throw exception;
or, throw;  ⟶ (rethrowing mechanism)
```

When an exception is thrown, control immediately jumps out of the try block which is followed by a catch blocks. The catch blocks are used to receive the thrown exception. When an exception is thrown, compiler tries to match the respective

of catch block 1 by 1. When a proper match is found, that particular catch block get executed and the remaining any other catch blocks are skipped.

If an exception is thrown but compiler is unable to find a proper match in catch block, then system defined abort function is called implicitly and program gets terminated.

```
int main(){
    int i, j;
    cin>>i>>j;
    try{
        if(j==0){
            throw 0;
        else{
            cout<<i*1.0/j;
        }
    }
    catch(int y){
        cout<<"Denominator value can't be zero";
    }
```

(No auto typecasting allowed)

Using inside function.

```
void calculate (int p, int r, int t){

    if ( p <= 10,000  &&  p >= 1000000){
        throw p;
    }
    else if ( n >= 6){
        throw 5.5f;
    }
    else{
        printf(" Interest : ", (pxrxt)/100.0);
    }
}


int main (){
    int i, j;
    try{
        calculate(i, j, k);
    }
    catch (int y){
        cout << y << " is not a valid principal ";
    }
    catch (float y){
        cout << y << " is not a valid time ";
    }
```

Generic catch block.

Though we have multiple catch block to receive various exceptions thrown by try block, We can have a ~~various~~ generic catch block to receive all types.

Note: If we have generic catch block along with the specific catch block, then the generic catch block must be the last among the other catch blocks.

Syntax: catch (...) {
...
}

Rethrowing Exception.

Where an exception is received in a catch block, the catch block may not process it in that point of the end and want to throw it again. If it is thrown again, then it is received in next enclosing try catch block. This concept is known as rethrowing an exception.

throw: throwing received exp.

throw() ⇒ restricted, no ~~re~~throw type.

Rethrowing Exception:

~~throw~~:

```
void show(int i, int j){
 try {
    if (j==0){
       throw j; }
    else {
       cout << i/j;
    }
 catch(int n){
    ~~if~~ throw;
 }
}
```

```
int main {
 try {
      show (n,y)}
 catch (   ){
   -
   -
 }
```

* Specifying / Restricted Exception.

Users can restrict and specify the type of exception to be thrown by mentioning the type in the throw list

```
void show ( (int i, int j) , throw (int))
```

If we throw any other type apart from the mentioned type in the throw list. Then abnormal program termination occurs.