```
void print()
{
  cout<<"Length:"<<
  length<<endl;
  cout<<"Width:"<<width<<
  endl;
    }
      };
void main()
{
  Box box(10,20);
  cout<<"Box1 Detalis"<<endl;
  box1.print();
   Box box2(box1);
   cout<<"Box2 Detalis"<<endl;
   box2.print();
       }
```

Destructor

rules :-

* Destructor name should be same as class name prefixed with (~) - Tilde operator.
* It should be public
* It doesn't contains arugements
* Never contains return types.
* It will be called automatically when the oleject goes out of the scope.

```
Class box
{
int length, width;
Public:
    Box (int x, int y)
    {
      length = x;
      width = y;
    }
```

```
void Print()
{
  cout<<"Length :"<<length<<
  endl;
  cout<<"width :"<<width<<
  endl;
    }
  ~ Box()
  {
   cout<<"object deleted";
     }
       };
void main()
{
  if (1)
    {
      Box box1(10,20);
      cout<<"Box1 Details"<<
      endl;
      box1.print();
        }
      cout<<"Test case";
        }
```

After exit a block it ↓

{

} it will deleted

Friend function

friend function define ~~or declare~~ always outside the ~~cla~~ class and declare in class

```
class box
{
  private:
      int length;
  Public:
      Box (int x)
      {
        length=x;
      }
      friend void print (Box);
  };
  void print (Box x1)
  {
    cout<<"Box length:"<<
    x1.length<<endl;
  }
```

friend — Keyword; void — Type; print — Function name

Box — Class name; x1 — object

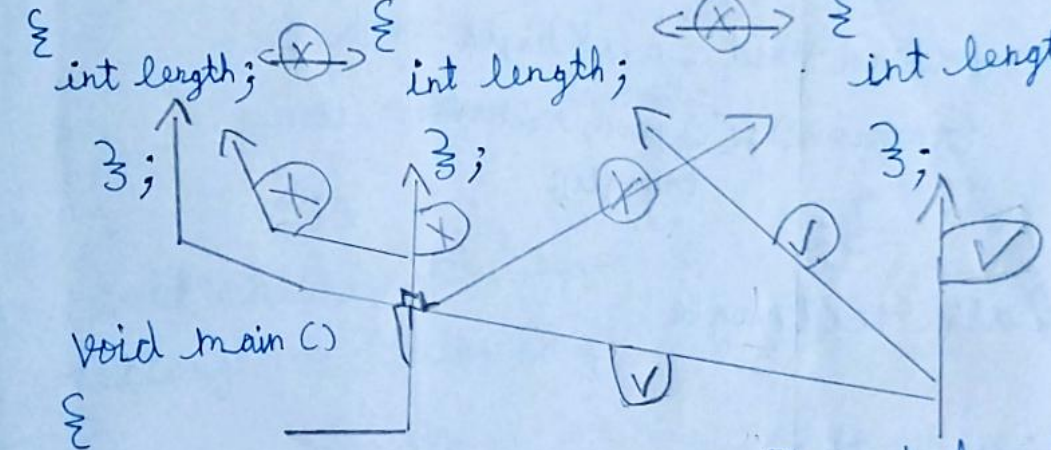
```
void main ()
{
  Box B1(10);
  print(B1);
}
```

friend function purpose

```
class Square
{
int length;
};
```

```
class rectangle
{
int length;
};
```

```
class Triangle
class ~~circle~~
{
int length;
};
```



```
void main ()
{
}
```

Friend function

```
void check ()
{
}
```

* here we have 3 length in our main function to compair 3 length and which one is greater but we can't access. just use friend function we can access the 3 class private. members and all other members.

```
class Square
{
  int length
  Square (int x)
  {
    length = x;
  }
  friend void checklength
  (Square);(Square, rectangle,
            triangle);
  };
class rectangle
{
  int length;
  Public:
    Rectangle (int y)
    {
      length = y;
    }
    friend void checklength
    (rectangle);
    (Square, rectangle,
    triangle); };
```

```
> class rectangle
;
class triangle;
```

```
class triangle
{
  int length;
  Public:
      triangle(int z)
    {
      length = z;
    }
    friend void checklength(
    square, rectangle, triangle);
  };
void checklength(square x,
rectangle g, triangle z)
{
  if (x.length > g.length &&
x.length > z.length)
  {
    cout << "Square length is
    greater";
  }
  else if (x.length > x length
&& g.length > z.length)
```

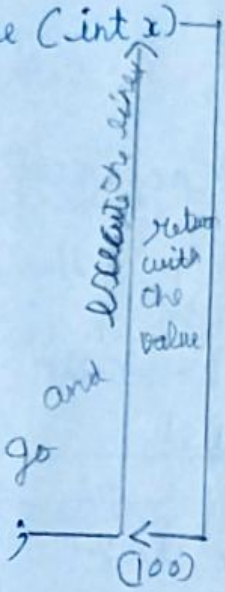
```
{
  cout << "Rectangle length is
    greater";
        }
  else
    {
      cout << "Triangle length
        is greater";
        }
    }

void main ()
{
  Square S(10);
  Rectangle R(20);
  Triangle T(30);
  checklength (S, R, T);
    }
```

~~Inline function~~

Inline function

Normal Function

```
int findSquare (int x)
{
  return x*x;
}
int main()
{
  findSquare(10);
}
```



Inline function

```
inline int findSquare (int x)
{
  return x*x;
}
int main()
{
  findSquare(10);
}
```

Keyword to do inline function.

copy this lines and put here and execute

We request the do in the same line the lines are If I shall it will do. What we request. If the lines are big It will do like normal function.

extra ~~content~~ Content and extra explanation.

Program is set of instructions

```
#include<iostream>
using namespace std;
int main()
{
  cout<<"Hello";
}
```

`#include<iostream>` — cout object present in iostream it present in iostream library file.

`using namespace std;`

main or void main is the beginning of program

`int main()` → function

`{` `}`

`cout<<"Hello";` — cout → object, << → insertion operator

Compile - Change your program to binary code.

↓

Execute or run

do the - What instructions you give 01.

Every lines of your program is statement the statement end with (;).

Program -> compiler -> Machin code
0 1
↓
Processor
output

using namespace std;-
like this
you also use this inside using directive
int main() main but it work inside
next is declaration The main block({ });
outside of a function.
std:: cout << "Hello";
directive declare
If you use namespace std
you don't need to use std::.

endl is present in <iostream>
(using std::cout;
using std::endl;) - using
What use a
If you donot use using from <iostream> declartion.
namespace std
you need to declare like this
std::endl

```
int main ()
{
  int interestRate = 12;
  int Edrate;
  Edrate = interestRate;
  interestRate = 14;
  cout << "Interest rate is:" <<
  interestRate << endl;
  cout << "Ed rate is :" << Edrate;
}
```

Data types in C / C++

* Primary :-
  * integer
  * character
  * Boolean
  * ~~Floating~~ Floating - Point
  * Double Floating Point
  * void
  * Wide character ~~or string~~
* Derived :-
  * Function
  * Array
  * Pointer
  * Reference
* User Defined :-
  * Structure
  * Class
  * Typedef
  * Union
  * Enum

#Include <iomanip> is a bsi library file that is used to manipulate the output.

```
int main()
{
  float f = 3.3333367F;
  double d = 3.3333367;
  cout << setprecision(13);
  cout << f << " " << d;
  return 0;
}
```

We can give 6 numeber after(.) to precision. Show you to use 6 more number after(.) you need to use double.

Implicit Type Conversion

```
int main()
{
  int a = 5.6;
  bool b = 85;
  bool c = 0;
  cout << a << b << c;
  int i = 'A';
  Char ch = 66;
  cout << ch << i;
}
```

Literals

int a = 45;
↓
literal

int a = 16;
↓
decimal number

a = 020;
↓
octal number for 16.

(a = 0x10; - hexa-decimal
a = 0X10;) small x or big x can give.

	Prefix	Meaning
Character	L	wide character
	u	unicode 16
	U	unicode 32
	u8	UTF-8
Integer	Suffix	Meaning
	u or U	unsigned
	l or L	long
	ul or uL	unsigned long
float	f or F	float
	l or L	long double

Types of literals

* int and float literals.
* char and string literals.
* Escape sequences
* Boolen literal - true - false
* Pointer literal - nullptr

Keywords

* alignas * alignof * bool * char16_t * char32_t * class * constexpr * const_cast * decltype * dynamic_cast * explicit * export * false * goto * mutable * namespace * ~~noexcept~~ noexcept * nullptr * reinterpret_cast * static_assert * static_cast * thread_local * true * typeid * typename * using * wchar_t * and * bitand * compl * not_eq * or_eq * xor_eq * and_eq * bitor * not * or * xor

Identifiers

int count

count ↓ Identifier

user give names are identifier.

rules

* can contain numbers, letters or underscores.
* No limit on length.
* Should be begin with a letter or underscore.

* case sensitive - count, Count.

Identifiers - conventions not rules.

* Meaningful name.
* only lower case.

* Multicword name - interestRate.

interest_rate.

C concepts

* operators
* if-else.
* switch
* loops - for, while, do while.
* Break, continue, goto.
* functions basics, types, recursion.
* Pointer
* structures and union.

& - Bitwise and operator.

| - Bitwise or operator.

^ - Bitwise XOR operator.

~ - Bitwise complement operator.

<< - Bitwise shift left operator.

>> - Bitwise shift right operator

Right shift = 8 -> 0000 1000 get out

left shift = 12 -> 0000 1100

Logical

o/p -> always 0 or 1

I/p -> handled as true or false

3 && 2 -> true

↓ ↓

true true

Bitwise

o/p -> any number

I/p -> handled bit by bit

3 & 2

↓ ↓ -> $010_2 + 2_{10}$

011 010

To the operators in each bit, output a bit

change to decimal number.

Recursion same.

calling a function inside that same function.

* Should be able to define solution of a problem in terms of solution of similar smaller problems.
* Proper terminating condition.

```
int fact(int n)
{
  if (n == 0)
    return 1;
  else
    return n*fact(n-1);
}
int main()
{
  int n;
cout << "enter num" << endl;
  cin >> n;
cout << "factorial is : " << fact(n);
    return 0;
}
```

```
typedef struct student stu;
stu s1;
stu *p;
```

Typedef is simply a way of giving a new name to an existing data type.

Unions: A union is a type of structure that can be used where the amount of memory used is a key factor.

* Similarly to the structure, the union can contain different types of data types.

* Each time a new variable is initialized from the union it overwrites the previous in C language but in C++ we also donot need this keyword and uses that memory location

* This is most useful when the type of data being passed through functions is unknown, using a union which contains all possible data types can remedy this problem.

* It is declared by using the keyword "union.

```
union OTFG {
        int Greek1;
        char Greek2;
        float Greek3;
    };
```

```
int main()
{
  union GFG G1;
  G1.Geek1 = 34;
Cout<<"The ~~next value stor~~ first
value at "<<"the allocated memory:"
<<G1.Geek1<<endl;
  G1.Geek2 = 34;
Cout<<"The next value stored"
 <<"after removing the "
 <<"previous value :"<<G1.Geek2<<
 endl;
  G1.Geek3 = 34.314;
Cout<<"The final value "<<"value at
the same allocated "<<"memory
space :"<<G1.Geek3<<endl;
  return 0;
}
```

Geek2 variable is assigned an integer (34). But by being of char type, the value is transformed through coercion into its char equivalent (").

```
struct student {
char name[25];
int rollno;
int mark;
};
```

→ void Print (struct student);

```
int main()
Struct student s1 = {"John",
7,92};
Print(s1);
return 0;
}
Void Print(struct student s)
{
cout << s.name;
cout << s.rollno;
cout << s.mark;
}
```

Solving a quadratic equation :- $ax^2+bx+c=0$

$r1, r2 = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ $disc = b^2-4ac$

$\sqrt{-1} = i$

$\sqrt{-3} = \sqrt{(-1*3)} = \sqrt{4}*\sqrt{3} = \sqrt{3}i$

$r1 = (-b + \sqrt{disc})/2a$

$r2 = (-b + \sqrt{disc})/2a$

```
int main()                    -> #include <cmath>
{
    double a, b, c, disc, r1, r2;
cout << "Enter the values of the coefficients a, b and c of the quadratic equation ax2+bx+c=0";
cin >> a >> b >> c;
disc = (b*b) - (4*a*c);
r1 = (-b + sqrt(disc)) / (2*a);
r2 = (-b + sqrt(disc)) / (2*a);
cout << "r1 is " << r1 << "r2 is " << r2;
    return 0;
cout.precision(15); }
```

```
int main()
{
double a, b, c, disc, r1, r2, rp, im;
cout << "Enter the values of a, b and c";
cin >> a >> b >> c;
cout.precision(10);
```

```
if (d disc = (b*b) - (4*a*c);
cout.precision(10);
if (disc >= 0)
{
r1 = (-b + sqrt(disc)) / (2*a);
r2 = (-b - sqrt(disc)) / (2*a);
cout << "r1 is: " << r1 << "r2 is: "
<< r2;
}
else
{
rp = -b / (2*a);
im = sqrt(-disc) / (2*a);
cout << "r1 is: " << rp << "+" <<
im << "i";
cout << "r2 is: " << rp << "-" <<
im << "i";
}
return 0;
}
```

```
class room
{
  public:
      int length;
      int breath;
      int calculate area()
      { return length*breadth;
      }
  };
  int main()
{
  room r1;
  r1.length=10;
  r1.breadth=8;
cout<<"Area is "<<r1.calculate
Area();
      }
```

```
class room
{
  int length;
  int breadth;
  public:
      void setlengthbreadth(
int l, int b)
```

```
{
  length = l;
  breadth = b;
}
int calculateArea()
{
  return length * breadth;
}
};
int main()
{
  room r1;
r1.setlengthbreadth(15,10);
cout << "Area is" << r1.calculateArea
() << endl;
cout << "length of r1 is" << r1.get
length();
        return 0;
}
```

```
> int getlength()
{
return length;
}
```

class **Encapsulation** — Binds the code and data

```
int x;
char y;        - Data
```

```
F1()   F2()    Functions - code
{      {
}      }
```

* Binds data and code
* Acts like black box and protects data.
* Public methods act as interface for accessing private data.
* Enables us to write methods without any

parameters as data and code are bound

object

object access Public. Public access the private data and function.

Public Functions :

Private data
Private Function

Public Data

You can show different by the single data type or count of that data type.

function overloading

You can create a function like the name but for that you pass parameter like :-

(int a)

int absl (int a)

data types

or by

count

int absl (int a, int b)

int absl (int a, int b, int c)

You can passes parameter of a single data type or by the count : 1, 2, 3.

```
int absl (int );
double absl (double);
int main ()
{
  int a = -5;
  cout << absl (a) << endl;
  double d = -9.0;
  cout << absl (d) << endl;
    return 0;
  }
int absl (int a)
{
  cout << "From int fun";
  return a < 0 ? -a : a;
  }
double absl (double a)
{
```

```
Cout << "From double fn";
return a<0 ?-a :a;
}
```

```
int main()
{
String s1="C++";
string s2=("Hello");
string s3=s1; -Copy initialization
string s4(s2); -direct initialization
string s5(6,'b'); string s6;
getline(cin, s6);
Cout << s5;
cout << s6;
return 0;
}
```

String comparison

```
int main()
{
    string s1="Hello";
    string s2=("Her");
    if (s1<s2)
    cout<<s1<<"is less than"<<s2;
    return 0;
}
```

l - 108
m - 109
n - 110 n is greater value so s1 is less than s2

Hello

Her

H and H is equal
e and e is equal
but l and n not equal because l is less than n in ASCII and place in order

```
if (s1 == s2)
   cout << "equal";

else
   cout << "not equal";
```

String concatenation

```
int main()
{
 string s1 = "Happy";
 string s2 = "Birthday";
 string s3 = s1 + " " + s2;
 cout << s3;
  return 0;
   }
```

-> = "Happy" + s2; is correct

= "Happy" + "Birthday"; is incorrect.

Strong Password Program

Write a function to check if the password is strong or not. A password is considered to be strong if :-

* Min length is 8 characters
* contains atlest one number
* contains atlest one special character.
* contains atlest one upper case letter.

```
#include<cctype>
bool isStrongPassword(string);
int main()
{
  string password;
  cout<<"Enter Password";
  getline(cin, password);
if(isStrongPassword(Password))
   cout<<"strong Password";
else
   cout<<"Not a strong Password";
   return 0;
      }
bool isStrong Password(string s)
{
  bool containsUpper = false,
containsSpecialchar = false, contains
Number = false;
   For(auto c:s)
{
  if(isupper(c))
   containsUpper = true;
```

```
if (ispunct(c))
    containsSpecialchar = true;
if (isdigit(c))
    containsNumber = true;
    }
  if (containsUpper && containsSpecialchar && containsNumber
  && s.size() >= 8)
    return true;
return false;
    }
```

isalnum	Check if character is alphanumeric
isalpha	Check if character is alphabetic
isblank	Check if character is blank
iscntrl	Check if character is a control character
isdigit	Check if character is decimal digit
isdigraph	Check if character has graphical representation
islower	Check if character is lowercase letter
isprint	Check if character is printable
ispunct	Check if character is a punctuation character
~~isupper~~ isspace	Check if character is a white-space
~~isspace~~ isupper	Check if character is uppercase letter
isxdigit	Check if character is hexadecimal digit
tolower	Convert uppercase letter to lowercase
toupper	Convert lowercase letter to uppercase

```
int main()
{
  string s = "hello";
for(auto c:s)
   cout<<c<<" ";
   return 0;
        }
```

```
int main
{
  string s = "hello";
  s[0] = toupper(s[0]);
   cout<<s;
      return 0;
         }
```

```
int main()
{
  string s = "hello";
  if(s.empty())
     cout<<"string is empty";
else
     cout<<"string is not empty";
      return 0;
         }
```

vectors

#include <vector>

~~int main()~~ using namespace std;

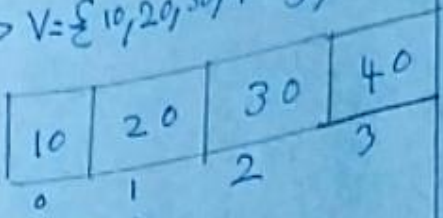
```
{   Class Room {
    Public:
        int l;
        int b;
            };

int main ()
{
    int a[3] = {1,2,3};
    vector <int> V1;
    vector <Room> rooms;
    Room rooms;
}
```

vector — Keyword; int — data type; V1 — variable

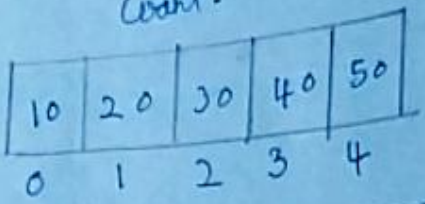
* collection of objects of same type.
* often referred as a 'container'.
* vector is a class template

how to add new ~~number~~ element in vector.

vector <int> V={10,20,30,40};



V.push_back (50); what element you want.



initializing vectors

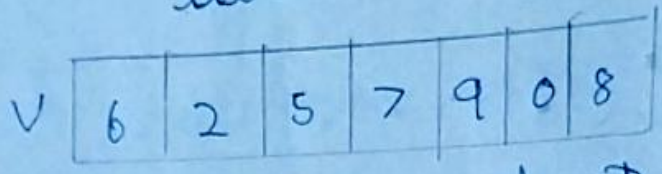
```
int main()
{
  vector <int> V1;
  vector <int> V2 = {2,4,5,6};
  vector <int> V3(V2);
  vector <int> V4 = V2;
  vector <int> V5 {4,5,6,1};
  vector <int> V6 (6);
                  size
  vector <int> V7 (5,3);

  for (auto V:V7)
    cout << V << " ";
  return 0;
}
```

```
int main()
{
vector <int> v = { 1,2,3,4,5};
v.push_back(6);
For (auto i:v) {
 cout<<i<<" ";
    }
 Cout <<v[:];
 return 0;
    }
```

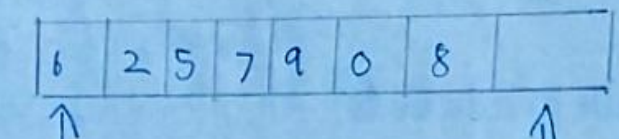
```
  int main()
{
 vector <string> grocerylist;
 string item;
 Cout<<"Enter items to be added to
grocery list";
   while (cin >> item)
{
 grocerylist.push_back(item);
   }
 Cout<<"Your list contains following
items"<<endl;
```

```
for (auto i : grocery_list)
{
    cout << i << " ";
}
    return 0;
}
```

iterators



* Used to access the elements of a container (like vectors)
* like pointers



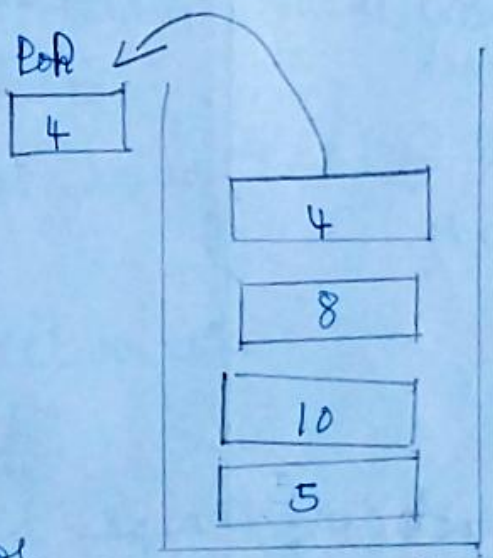
v.begin()
this returns a iterator

v.end()
this also returns a iterator

```
int main()
{
    vector<int> v = {4,5,6,3,7,6};
```

```
auto b = v.begin();
while (b != v.end())
{
  cout << *b << " ";
  b++;
}
return 0;
}
```

stacks using vectors



If you store a new number in stack is called push.

If you take a number for stack is called POP.

```
int main()
{
    stack S;
    S.push(1);
    S.push(2);
    S.push(3);
cout << S.pop();
cout << S.pop();
cout << S.top();
return 0;
}
```

It is used to pop a element for stack and the element it will be delete for the stack.

It is used to see the top element in stack

```
class stack {
    vector<int> v;
    Public:
void push(int a)
    {
        v.push_back(a);
    }
    int top()
{
return v[v.size()-1];
}
```

```
int pop()
{
  if (v.size() != 0)
  {
    int t = top();
    v.pop_back();
    return t;
  }
};
```

Pass by value and Pass by Reference.

```
int square(int a)
{
  return a*a;
}
void squarebyref(int &b)
{
  b = b*b;
}
```

```
int main()
{
    int a = 5;
    cout << Square(a) << endl;
    cout << "The value of a is:" << a <<
    endl;
    int b = 4;
    Squarebyref(b);
    cout << "The value of b is :" << b;
}
```

Pass by value :-
* Variables are copied
* Secured
* More space for larger variables

Pass by Reference :-
* Direct access is granted to original variables.
* No overhead of copying large data
* Not secure

```
int somefun()
{
    int a = 10;
    return a;
}
```

```
int main()
{
  int x
  x = somefun();
  cout << x;
}
```

```
int & somefn()
{
  int a=10;
  return a;
}
int main()
{
  int x
  x = somefun();
  cout << x;
}
```

error program

- dangling reference

We are returning a reference but a variable is not in memory. int main() try to access a variable so error.

Don't return a local variable reference for a function.

If we return a local variable reference it is called dangling reference.

returning References

```
int & some fun()
{
  static int a = 10;
```

```
return a;
}
int main()
{
  int x;
 x = somefun();
 cout << x;
}
```

```
int & somefun()
{
  static int a = 10;
  return a;
}
int main()
{
 & somefun() = 15;
  cout << somefun();
}
```

Default Arguments

```
int product (int a =1, int b, int c =1,
int d = 1);
int main ()
{
  cout << product (3,5);
}
int product (int a, int b, int c, int d)
{
  return a*b*c*d;
}
```

You Default Arguments like last one or everything in your variables. compiler will add what value your give variable. like this :-
product (3,5,1,1);

incorrect :-
int a, int b, int c=1,
int d.

correct
int a=1, int b=1, int c=1,
int d=1;

int a, int b, int c =1,
int d=1;

int a, int b, int c,
int d =1;

Function templates

don't need to overload your function. templates will make easy.

```
template < typename T >   (typename or class for class; T = vari)
T Maximum (T a, T b, T c)
{
  T max;
  max = a;
  if (b > max)
    max = b;
  if (c > max)
    max = c;
  return max; }
```

```
int main()
{
cout << "int:" << maximum(3,5,7)
<< endl;
cout << "double:" << maximum(4.0,
6.5,9.7)); << endl;
cout << "char:" << maximum('x'
'z','w','b'));
}
```

Default constructors

```
Class course {
    string courseName;
};
int main()
{
course c1;
}
```

If we don't use any constructors. It will call the string class constructor.

interface

* Hides implementation or main code
* Describes the behaviour of a class.

File : stack.h

```
#include <vector>
class stack {
    std::vector <int> v;
  Public:
  void push (int a);
  int top ();
  int pop ();
        };
```

File : stack.cpp

```
#include "stack.h"
void stack::push (int a)
{
  v.push_back (a);
  }
   int stack::top ()
{
  return v[v.size()-1];
  }
```

```
int stack::pop()
{
  if (v.size() != 0)
  {
    int t = top();
    v.pop_back();
    return t;
  }
}
```

File : main

```
#include "stack.h"
using namespace std;
int main()
{
  stack s;
  s.push(1);
  s.push(2);
  s.push(3);
  cout << s.pop();
  cout << s.pop();
  cout << s.top();
  return 0;
}
```

Create a project and add 3 file and build and run

Const objects and const member functions.

```
class book {
    String bookname;
    String authorname;
public:
    Book (string bn, string an)
    {
        bookname = bn;
        authorname = an;
    }
    String getBookname() const
    {
        return bookname;
    }
};
int main()
{
const book b1("c++", "raj");
```

Const objects can access const functions only.

If you want call a ~~not~~ const other ~~function~~ functions in const function that ~~fun~~

```
cout << b1.getbookname();
}
```

function will ~~be~~ ~~const~~ want to const function

member initializer list

```
Class date
{
  int day;
  int month;
  int year;
  Public:
   date (int d=; int m=1, int y=2023)
   {
     day = d;
     month = m;
     year = y;
   }
   string getdate ()
  {
return to_string ((day) + '/' +
to_string (month) + '/' + to_string
(year);
   }
};
```

```
class Student {
    string name;
    Date dob;
Public:
    Student (string n, int d, int m,
int y)
    : name(n), dob (d, m, y)
    {
    }
        member initializer
        list
    string getdob()
{
    return dob.getData();
    }
};
int main()
{
    Student S1 ("Ram", 2, 5, 2010);
    Cout << S1.getdob();
    }
```

```
class count {
  friend void setx (count &c, int
x1);
  private:
      int x;
  public:
      count()
      : x(0)
    {
      }
     int getx()
   {
     return x;
          }
              };
void setx (count &c, int x1)
{
  c.x = x1;
     }
int main()
{
 count c1;
 setx (c1,9);
 cout << c1.getx();
    }
```

this pointer

```
class Message {
  private:
      string Erom;
      string to;
      string body;
  Public:
      void setfrom (string
from)
    {
      this -> Erom = from;
    }
    string getfrom ()
    {
      return Erom;
    }
    };
int main()
{
  Message m;
  m.setfrom ("raj");
  cout << m.getfrom ();
    }
```

Static members

```
class Room {
    static int roomsCount;
    int length;
    int breadth;
public:
    room (int l, int b)
    {
        length = l;
        breadth = b;
        roomsCount ++;
    }
    int calculateArea()
    {
        return length * breadth;
    }
static int getRoomsCount()
    {
        return roomsCount;
    }
};
```

```
int Room::roomsCount = 0;

int main()
{
    room r1(30,20);
    room r2(15,15);
    room r3(15,10);
    cout<<r3.calculateArea()<<endl;
    cout<<Room::getRoomsCount();
}
```

Operator overloading

~~*(+) *(-)~~ * list of operators that can be overloaded.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

operators that cannot be overloaded

::	.*	.	?:

Rules of operator overloading :-

* only existing operators can be overloaded.
* Meaning of how the operator works should not be changed.
* Return types should be compatible with built in types.
* Precedence and associativity of operator cannot be changed

Overloading I O operator
(<< and >>)

```
Class Room
{
  private:
        int length;
        int breadth;
  Public:
friend istream & operator >>
(istream & input, Room & r)
  {
    input >> r.length >> r.
breadth;
      return input;
        }
friend ostream & operator <<
(ostream & out, Room & r)
  {
    out << r.length << "x" <<
    r.breadth;
    return out;
      }
        };
```

> cout << "Enter length and breadth";

```
int main()
{
  room r1;
  Cin >> r1;
  cout << r1;
}
```

operator overloading increment (++)

```
class Time {
  private:
  int hours;
  int minutes;

  public:
    Time (int h, int m)
{
  hours = h;
  minutes = m;
}
  void operator ++ ()
{
  minutes ++;
  if (minutes >= 60)
{
```

```
hours++;
minutes -= 60;
    }
}
void displayTime()
{
  cout << hours << ":" << minutes;
    }
};
int main() {
  time t1(6,30);
  ++t1;
  t++;
  t1.displayTime();
}
```

```
void operator
++(int)
{
  minutes++;
  if (minutes >= 60)
  {
    hours++;
    minutes -= 60;
  }
}
```

operator overloading binary operators (+, <)

```
class Distance {
  private:
    int feet;
    int inches;
```

```
Public:
friend istream &operator >>(
istream &in, Distance &d)
{
  cout << "Enter feet:";
  in >> d.feet;
  cout << "Enter inches = should
be between 0 and 11 inclusive:";
  in >> d.inches;
  return in;
}
friend ostream &operater <<(
ostream &out, Distance &d)
{
  out << d.feet << "ft" << d.
inches << "in";
  return out;
}
Distance operator + (Distance d)
{
```

~~this -> feet + d.feet~~

```
Distance temp;
temp.feet = this -> feet +
d.feet;
temp.inches = this -> inches+
d.inches;
if (temp.inches > 11)
{
temp.feet ++;
temp . inches -= 12;
}
return temp;
}
bool operators < (Distance d)
{
if (this -> feet < d.feet)
return true;
if (this -> feet == d.feet
&& this -> inches < d.inches)
return true;
return false;
}
```

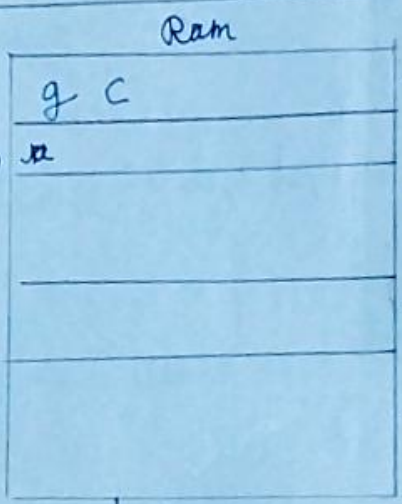
```
};
(+)
int main()
 Distance d1, d2, d3;
 Cin >> d1;
 Cout cin >> d2;
 d3 = d1 + d2;
 Cout << d3;
        }
(<)
int main() {
 Distance d1, d2, d3;
 Cin >> d1;
 Cin >> d2;
 if (d1 < d2)
 {
   cout << "d1 is shorter";
        }
 else
 {
   cout << "d1 is longer";
    } }
```

~~New and del~~
New and delete

Dynamic Memory: It is the memory that can be allocated - de-allocated by the operating system during the run-time of a C++ program. It is more efficient than static memory because we can de-allocate and reuse ~~such~~ our memory during the run-time of our program.

```
int g;
main()
{
  int a;
  static char c;
}
```

Global variable and static variable store in Data segment



When we are allocated Dynamic memory it will reallocated in Heap. and it will be return with a address. We can access the memory with pointer only. ~~new and delete~~ C++ supports dynamic allocation and de-allocation of objects using the new and delete operators. These operators allocate memory for objects from a pool called the free store (also known as the heap).

```
int main (){
int *p = Null; NULL;
p = new int (10);
```

or we can declare as :-

```
int *p = new int (10);
int *p2 = new int [20];
delete p;
delete [] p2;
}
```

value

allocated 4 memory for 20 int data type.

```
class Room {
    int length;
    int breadth;
};
int main (){
Room *p = new Room [10];
}
```

Copy Constructors and Explicit constructors.

```
class complex {
    int real;
    int img;
```

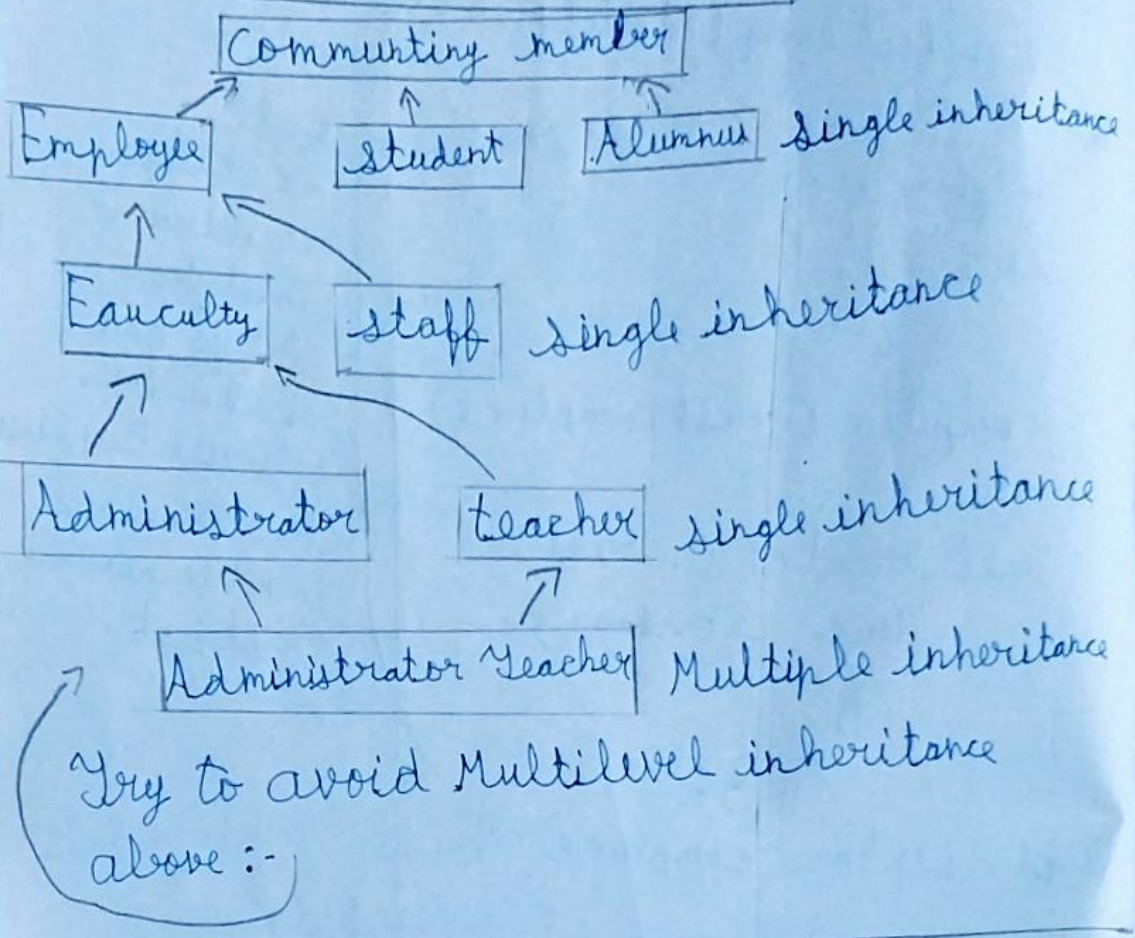
```
Public:
  Complex(int r=0, int i=0)
{
  real = r;
  img = i;
}
  complex(const complex &c)
{
  real = c.real;
  img = c.img;
}
  void display()
{
  cout << real << "+" << img << "i";
}
};
int main() {
  complex c1(3,4);
  complex c2(c1); or c2=c1;
  c2.display();
}
```

```
class complex {
  public:
    int real;
    int img;
Explicit constructors  =0  =0
  Explicit complex (int r, int i)
{
  real = r;
  img = i;
      }
  complex (const complex c)
{
  real real = c.real;
      img = c.img;
    }
      };
void display (complex c) c=10
{
  cout << c.real << "+" << c.img << "i";
    }
  int main () {
    complex c1;
error- display (10);
correct - display (complex (10));
    }
```

Call.
We change number to object.
It is bad Behavior.
so to use Explicit Constructors to not do numbers to object.

inheritance

* Acquiring the property of an existing class.
* A form of software Reuse.




```
class Employee {
  int empId;
  string name;
  double salary;
  Public:
```

```
Employee(int eId, string n,
double sal=0)
{
    empId = eId;
    name = n;
    salary = sal;
}
void print()
{
cout << "Employee Id:" << empId << endl;
cout << "Name:" << name << endl;
cout << "Salary:" << salary << endl;
}
void setname(string n)
{
    name = n;
}
string getname()
{
    return name;
}
```

```
double/void setSalary(double s)
{
  salary = s;
}
double getSalary()
{
  return salary;
}
};
Class ContractEmployee : public
Employee {
  Private:
      int workhours;

  Public:
    ContractEmployee(int eId,
string n, int WH)
: Employee(eId, n)
{
  workhours = WH;
  setSalary();
}
```

```
void SetSalary()
{
  Employee :: SetSalary (work Hours*
100);
    }
};
int main() {
  Employee e1(1,"raj",25000);
  e1.print();
  e1.setSalary(26000);
  e1.print();
 contract Employee ce2(2,"
By Riya",140);
  ce2.print();
}
```

If want out works hours write this or don't write. If you don't write it will call base class Public Functions

```
void Print()
{
  Employee :: Print
;
cout<<"work Hours:"
<<workhours<<
endl;
```

Protected Access Specifier

Protected:

derived class only and access the protected data and function.

```
int empId;
fn() and
{
}
```

friend functions only

we can store datatypes and functions in protected.

Inheritance types

Base class member access specifier	Public inheritance	Private inheritance	Protected inheritance
Public	Public will be available as Public	Private	Protected
Protected	Protected will be available as Protected	Private	Protected
Private	inaccessible	Inaccessible	inaccessible

Virtual functions

```
class Shape {
public:
~~virtual~~ double base, height;
    Shape (double a, double b)
  {
    base = a;
    height = b;
    }
```

```
virtual double area()
{
  cout << "Base class";
  return 0;
    }
      };
class triangle : public Shape {
public:
    Triangle (double a, double b):
Shape(a,b) { }
  double area()
{
  cout << "Triangle area:";
  return base * height/2;
    }
      };
class Rectange : public Shape {
public:
    Rectangle (double a, double b)
: Shape (a,b) { }
  double area ()
{
  cout << "Rectangle area =";
```

* Virtual function is a member function of a base class that is expected to be redefined (overridden) in the derived class.

→ If we write like this it will be Abstract class :-
virtual double area () = 0;

```
return base*height;
    }
    };
int main()
{
  Shape *s;
  triangle t(10.0,20.0);
  cout << t.area();
  s = &t;
  cout << s->area();
  Rectangle r(10.5,20.7);
  s = &r;
  cout << s->area();
    }
```

Virtual functions help to do dynamic binding or late binding

- static binding or early binding.

- dynamic binding or late binding

virtual double area()=0; - pure virtual function. giving virtual function like in a classes is called Abstract classes.

Abstract classes

We can't create object for Abstract classes. but we can create objects for derived class from Abstract class. and we can create pointer.

derived

virtual double area() = 0;

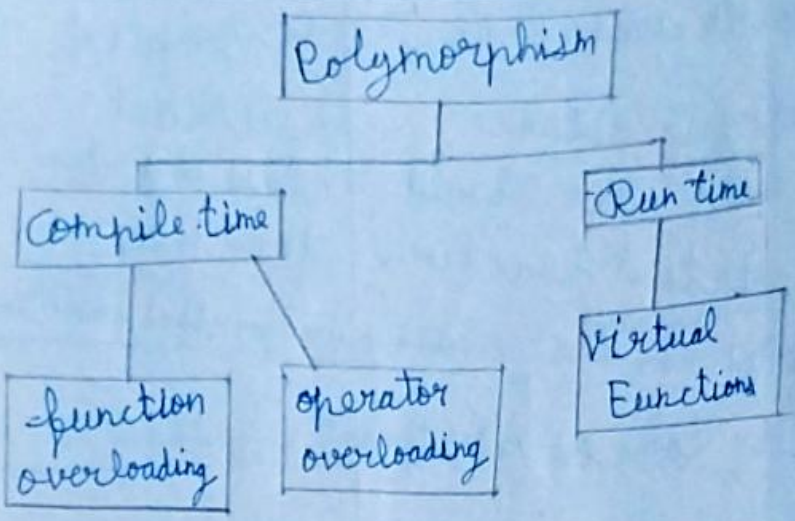
Example of Abstract class in virtual function.

which use class inheritance
Abstract class ~~the~~ should
2 overwrite
use pure virtual function.
or 3 if you don't use overwrite that class pure virtual function.
that class 1 also & compiler take as Abstract class.
4 5

* Base classes that make the code reusable and extendable
* Must have atlest one pure function
* Cannot create objects for abstract classes.
* Derived class must provide definition for the pure virtual function otherwise it is also becomes an abstract class.

Polymorphism

* Polymorphism means "many forms".
* Ability of the code to behave different in different scenarios.



Exception Handling

```
# include <iostream>
# include <stdexcept>
  using namespace std;
int main()
{
  int a, b;
  cout << "Enter 2 numbers: ";
  cin >> a >> b;
  try {
    if (b == 0
      throw "Divide by zero error"; or 5;
    cout << a / b;
```

```
catch(const char *ch)
{
cout << "Error occurred.." << ch;
}
catch(int i)
{
cout << "Math error.." << i;
}
cout << "End of program";
}
```

nesting of try possible.

default throw;

default catch(...)
{
cout << "Error occurred..";
}

```
int main()
{
  try
  {
    int *arr = new int[10000];
  }
  catch(exception &e)
{
cerr << "standard exception :"
<< e.what();
}
```

Files

```
#include <iostream>
#include <Estream>
using namespace std;
int main()
{
  fstream file;
file.open("myfile25", ios::out);
cout<<"writing to console";
file<<"I am writing this to the file";
  file.close();
  string S;
file.open("myfile25", ios::in);
  while(file>>S)
{
  cout<<S<<' ';
    }
```

*ios::in - open for input operations.

*ios::out - open for output operations

*ios::binary - open in binary mode.

*ios::ate - Set the inital position at the end of the file. If this flag is not set, the initial position is the beginning of the file.

*ios::app - All output operations are performed at the end of the file, appending the contents to the current content of the file.

*ios::trunc :- If the file is opened for output operations and it already existed, its previous content is deleted and

final

```
class Employee final {
  Public:
    String name;
    int Salary;
    void hello() {
      cout << "Hello";
    }
};
```

If you use final in a class that class. can't be inherited in other class. ~~in other class.~~

Error

```
class contract Employe :
Employee {
  int contract Duration;
  void hi() {
    cout << "hi bro";
  }
};
```

replaced by the new one.

All these flags can be combined using the bitwise operator or (|):-

1. ofstream myfile;
2. myfile.open ("example.bin", ios::out | ios::app | ios::binary);

Class templates

```
template <class T>
Class Stack {
    vector<T> v;
  Public:
   void push (T a)
   {
     v.push_back(a);
        }
   T top()
  {
   return v[v.size()-1];
        }
  T pop()
  {
    if (v.size()!=0)
    {
      T t = top();
      v.pop_back();
```

```
return t;
    }
        }
            };
int main()
{
  stack<int> s;
        data type
  s.push(1);
  s.push(2);
  s.push(3);
  cout << s.pop();
  cout << s.pop();
  cout << s.top();
        }
```