Overview Of Programming Language

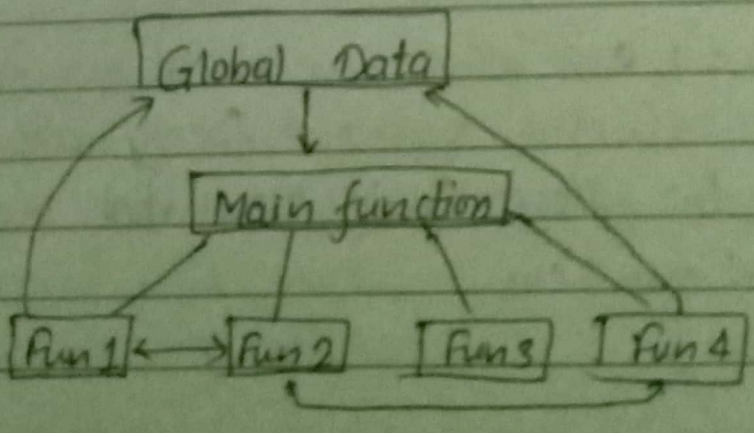
High Level Language

- POP — C, COBOL, FORTAN
- OOP — C++, Java, Phyton, etc

All computer programs consists of two elements : code and data and these two elements govern how a program is constructed. Here are two mechanism to construct the program in HLL.

1) Procedure Oriented Programming approach / languages

- High level language like C, COBOL, FORTAN use procedure oriented model for the development of program. So they are called Procedure Oriented Programming language.
- In this method program is characterized as a series of statement that tells computer to perform specific task.
- Whenever a program becomes large and complex, then it is divided into a number of function and each functions access the globally declared data. So it is called procedure (function) oriented programming.



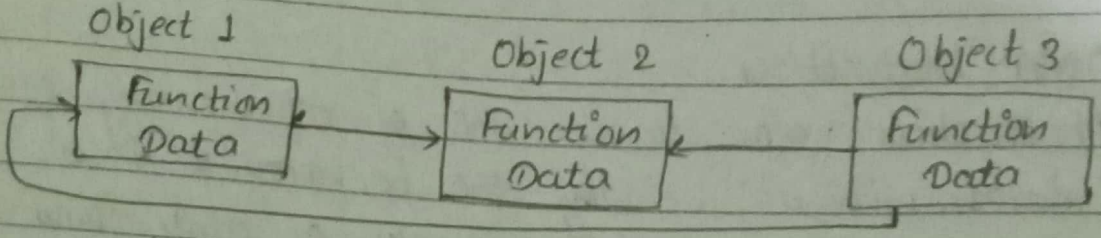
- In POP the local and global data are used however the global data are accessible to all the functions in the program whereas local data are accessible only to the function where it is declared and cannot be accessed by other functions in the program. So local data are more secure than global data. Thus global data are more more vunerable to another programmer to be changed unknowingly.
- The separation arrangement of data and function does a poor job and modeling the things in real world. That's why POP approach doesn't model the world system perfectly.

Basics of OOP

- The POP wouldn't model the real world system perfectly so to overcome the complexity of POP, the concept of OOP was evolved. In OOP, programs are divided into objects where each object consists of data and functions. The data of an object are accessible only to the function belonging to that object but function of one object may or may not access the function of another object.
- In OOP data are treated as critical elements in the program which are restricted to move freely around the person to protect them from their accidental modification outside the function.

Features of OOP

- Emphasis is given to data rather than procedure.
- Programs are divided into objects.
- New data and function can easily be added whenever necessary.
- It follows bottom to top approach of programming.



Concepts of OOP

1) Class
2) Objects
3) Abstraction
4) Inheritance
5) Encapsulation

Class

- A class is a collection of similar type of objects. Once the class has been defined we can create any ~~any~~ number of objects of that class. Each object of a class has its own attributes (data) and function. Thus objects consists of data and function. The class is user defined data type that is used to declare objects that's why objects are variables of type class. For example apple, mango, banana, orange are the objects of class fruit.
- Class are user defined data type like structure in C and behave much similar to the structure.

Objects

Objects are the class type variables & these represent real world elements which have their attributes & functions. For example; bike is an object of class vehicle which have its own name, board, colour, capacity number as a data and it has unique function.

Data Abstraction

Abstraction refers to the act of representing essential features without including the background details and explanation. Class uses the concept of abstraction and are defined as a least of abstract attributes and function

Encapsulation

It is the process of wrapping data and function into a single unit. The data is not accessible to the outside world but functions are declared in such way that the class can access them. The functions provide an interface between objects, data and the program. The insulation or blocking of data from direct access by a program is achived through encapsulation and its called data hiding, information hiding.

Inheritence

It is the process of deriving a new class from an existing class. The new class is called child class and existing class is called base class or parent class. During inheri

some or all the properties of base class are inherited to the child class depending upon the derivation mechanism.

Polymorphism

It is the ability of having more than one form. In context of OOP, polymorphism refers to the fact that a single operation can have different behaviours in different instances. To calculate the area of different geometrical shapes like triangle, circle, rectangle, square, etc. a single function area() is used where the function area calculates area for different geometrical shape at different times.

Difference Between OOP & POP

S.N.	POP	OOP
1.	Emphasis is given on procedure.	Emphasis is given on data.
2.	Programs are divided into functions.	Programs are divided into data objects.
3.	Follows top-down approach of program design.	Follows bottom-top approach of program design.
4.	Generally data can't be hidden	Data are hidden through encapsulation
5.	It does not model the real world problems.	It models real world problems.
6.	Maintaining and enhancing code is complex.	Maintaining and enhancing code is easier.
7.	It has less execution time.	It has more execution time
Ex.	C, COBOL, Pascal, FORTRAN, etc	Ex. C++, Java, Phyton, etc.

Benefits of OOP

→ Reduces complexity of program while solving solving problems
→ Through ~~in~~inheritance we can eliminate redundant code and extend use of existing code.
→ Use of classes
→ Data hiding is possible so that program become more secure.
→ Easy to partition the work in projects on objects.
→ OOP can be upgraded from small to large system.

Applications of OOP

→ Real time system (Nuclear power plant)
→ Simulation and Modeling
→ (Web designing) hypertext, hypermedia, expertext.
→ Artificial intelligence (AI)
→ CAD (compuder aided design)

Disadvantages of OOP

→ It requires much more compilation time.
→ Requires detail knowledge of software engineering & programming methodogy to use it.
→ Benefits only in long run while managing large software projects.

Introduction & History of C++

→ C was evolved from BCPL & B. BCPL was developed by Martin Richards (1967) & developed by Martin Richards (1967) & B was developed by Ken Thompson (1970).
→ C++ is an extension of C developed by Bjarne

Stroustrup (1979) at bell laboratory.

→ Initially it was called "C with classes" and later named "C++" in 1983. C++ include all features, attributes and benefits of C.

→ C++ preserves all these advantages and compatibilities of C.

→ In 1982, Stroustrup developed new and better language i.e cleaned up and extended version of "C with classes" as C84. After few month its name changed to C++.

→ Stroustrup designed "Cfront compiler" bet^n 1982 to 1983 using "C with classes".

→ "C with classes" was named to C++ in 1989.

→ In 1989 the ANSI (Americal National Standard Internationalize) committee for C++ was founded.

→ The new standart C++0x is in process to releas & released now

C++ versus C

C Programming	C++ Programming
→ The C language is implemented by microsoft Borland C, GCC and many more.	C++ implemented by GCC, MS visual C++, C++ builder.
→ It follows procedural approach for programming.	It follows OOP approach for programming.
→ C programs execute faster than C++ programs.	C++ programs take much more execution time than C programs.
→ C has fewer libraries than C++	C++ has more libraries than C

Features of C++

- Namespace
- classes
- derived classes
- Access controller (Accessibility or privacy)
- Constructor & destructor
- Reference variable
- Function overloading
- Inline function
- Operator Overloading
- Memory Management in C++
- Stream class libraries
- Run time polymorphism
- Templates
- Exception handling, etc.

Unit 2 Basics Of C++

Some features of C++

1) Comment

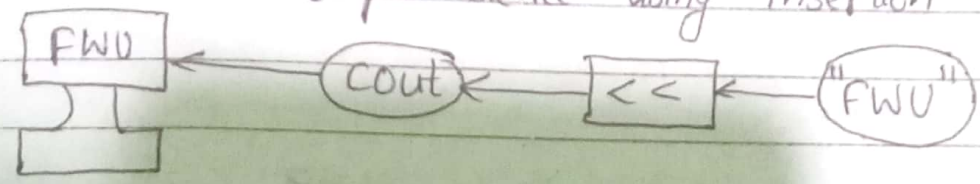
/* - - - - */ ; for multiple line comment

// - - - - ; for single line comment

2) Input Output

→ The "iostream" class contains two objects "cout" & "cin" which are used for output and input data.

→ The object 'cout' is used to display the content of a variable into output device using insertion operator '<<'



Syntax : cout << variable-name ;

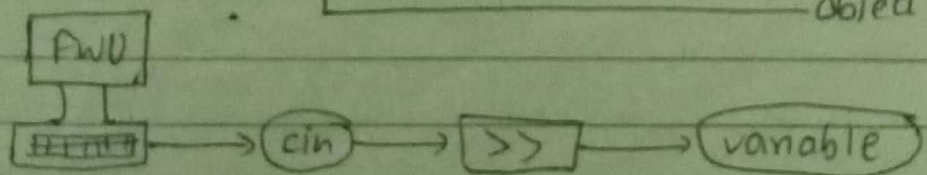
- variable
- ~~insertion~~ extraction operator
- Object cout

→ The object "cin" is predefined object of class "iostream".

→ The object "cin" is used to input data from keyboard to put it on the variable

Syntax : cin >> variable-name ;

- variable
- ~~insertion~~ extraction operator
- object cin



[Fig: i/p with cin & extraction operator]

* cascading of input & output operator

```
cin >>x >>y>>z; // multiple input
cout<<x <<y<<z; // multiple o/p
```

3. class Syntax:

```
class class-name
{ private : data-members;
            member-functions;
  public : data-members;
           member-functions;
}
```

Object creation

```
class_name obj 1, obj 2, obj 3, - - - obj n;
```

Structure of C++ program

→ Basic structure of C++ program comprises

i) include files
ii) class declaration & definition
iii) member function & definition
iv) Main function program (object creation, etc)

We can this ... at the ... → will be ... will ...

eg.

```
# include <iostream.h>
# include <conio.h>
        class student
        {
          private : int roll.no;
          public :
                  void enter ()
                  {
                   cout << " enter roll no endl";
                   cin >> roll.no;
                  }
                   void display ()
                  {
                    cout << " Roll_no = " << roll.no;
                  }
                 };
                void main ()
               {  clscr ();
                  student a;   // object creation
                  a.enter();   } // function call using dot operator
                  a.display(); }
                  getch ();
               }
```

O/P : enter roll no: 10
roll no = 10

i) **Tokens**

The smallest individual unit ~~units~~ in programming are called tokens. e.g. keywords, identifiers, constants, strings, operators.

ii) **Keywords**

Reserved identifers are called keywords. They can not be used as user defined names for program variables or other user defined program elements. e.g. int, float, class, void, etc.

iii) **Identifers**

User defined names of variables, functions, classess, str. are called identifers. e.g. int a, b ← identifes

iv) **Constants**

It refers to fixed value that do not change during program execution.

Constants
- → String Constant : "Hello" ; inside double inverted comma
- → Character Constant : 'A' ; inside single inverted comma
- → Numeric Constant : 0, 1, 2, 3, ... etc

Operators

i) Arithmetic Operators (+, -, *, /)
ii) Assignment (=, +=, -=)
iii) Comparison (>, <, >=, <=)
iv) special operator

a) Unnary operator (++, --)
b) binary operator (+, -)
c) comma operator (,)
d) scope resolution operator (::)
e) ternary operator (?:)

Manipulators

→ Used to format the data display

a) endl (newline)

e.g. cout << a << endl << b;

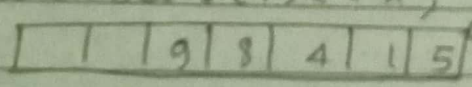
o/p; a
b

b) setw (field width)

eg. if x = 98415 then

cout << x; ⇒ x = 98415

cout << setw(7) << x; gives o/p as



← from righ to left placement

4. Implicit Conversion

m = 2+3.5;

→ C++ automatically converts data types when mixed expression occur. It is called implict or automatic conversion. Smaller type is converted to wider type.
Here int ⟶ float (since float is wider type data)
∴ m = 2.0 + 3.5 = 5.5

Data Types in C++

1) **Fundamental / Built in data types**
   - → Void
   - → integral type ⟶ int (2 byte)
   - → char (1 byte)
   - → floating type → float (4 byte)
     → double (8 byte)

2) User defined type
   - → Structure
   - → Union
   - → Class
   - → Enumeration (enum)

3) Derived type
   - → Array
   - → Pointer
   - → Function
   - → Reference (b = &a);

Enumeration

→ It is used to create symbolic constant. It provides attaching numbers to names

Syntax : enum (keyword) name_of_enumeration { mem1, mem2, ..., memn }
(name_of_enumeration: user defined name)

e.g.
```
enum sample { red, yellow, green }
cout << red << yellow << green ;
```
o/p : 0 1 2 // by default first member is initialized zero and others 1, 2, ... accordingly

Assigning integer values to enumerators

e.g. 1
```
enum name { ram = 2, shyam = 4, Hari }
cout << ram << shyam << Hari ;
```
o/p : 2, 4, 5

Reference

A reference provides an alias, a different name for a variable. Used in passing arguments to a function by reference. Denoted by "&" operator.

Syntax :

```
data-type & reference-name = variable
```

e.g.
```
int x;
int &b = x;
```
here x and b both refer to same data in memory

Functions (Methods) in C++

The functions are bulding blocks of c++ programming. It groups a number of program instructions into a single unit and can be called (invoked) by a part of program as per requirement. Its main advantage is to reduce the size of program by eliminating the code and by calling it whenever required.

Basic needs for function use:

(a) Function declaration (prototype)
(b) Function definition
(c) Function call

(a) Function Declaration

Function declaration not needed if function is defined above main function

Syntax

```
return-type function-name (argument list);
```

(b) Function definition

Syntax

```
return_type function_name (argument list)
{
  body statements;
}
```

(c) Function call

Syntax

```
function_name (variables)
```

★ WAP to find the maximum of three numbers using function.

```
# include <iostream>
# include <conio.h>
void max (int, int, int);
   void main()
{
    clrscr();
    int x,y,z;
    cout <<"enter three numbers \n";
    cin >> x >>y >> z;
    max(x,y,z);
    getch();
  }
  void max(int a, int b, int c)
{
 if (a>>b && b>>c) cout<<"max =" << a;
else if (b>>a && a>>c) cout <<"max =" <<b;
 else
    cout <<" max =" <<c;
 }
```

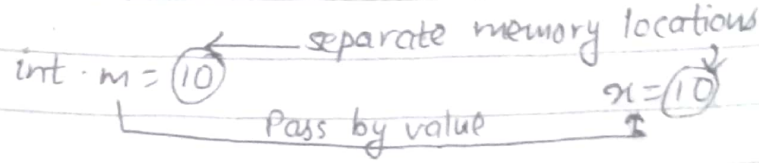
Calling Function

(i) Call by value (ii) call by reference

(a) Call by value

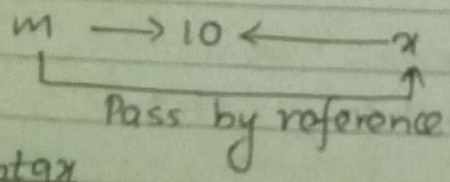
Here the value of particular variable is copied in called

function.



(b) Pass by reference

Reference provides an alternative name for the single variable. One name is used in calling function & other name is used in called function with same memory location. In this method, a reference to the actual argument in the calling function is passed so that the called function doesn't create own copy of original data in new memory location & this works same as in calling function but only with different name. So any change in original data in the called function is reflected back to the calling function.



Syntax

a) Using pointer

Swapping two numbers

```
void swap (int *a, int *b) // function definition
 {
    int t;
    t = *a
    *a = *b;
    *b = t;
 }
```

In this ... single force at the ... If $\vec{F_R} = 0$, the $\vec{C_e}$ will be the ... $\vec{F_R}$ & $\vec{C_e}$ are both null vector then ...

(2) Using Reference

```
void swap (int &, int &);
void main ()
{
  int x,y;
  cin >> x >>y;
  swap (x,y);
  cout <<x<<y;
  void swap (int &a, int &b)
  { int t;
    t=a
    a=b
    b=t;
  }
```

Returning Function

(i) Return by value

(ii) Return by reference

(a) Return by value

e.g.
```
int sum(int, int);
void main()
{
  int a, b, c;
  cin >> a >> b;
  c = sum(a,b);
  cout << c;
  getch();
}
int sum(int x, int y)
{ int s = x+y;
  return s; // returns by value
}
```

(b) Return by Reference

The primary reason for return by reference is to allow to use a function call on the left side of the equal sign.

e.g.
```
int & max(int & x, int & y);
void main()
{ int a,b;
  cin >> a >> b;
  Cout << " before calling function" << " a " = " << a << " and b " = " << b;
  max(a,b) = 100;
  cout << " after calling function a = " << a << " and b = " << b;
  getch()
}
```

```
int & max (int &x , int &y)
{
  if (x>y)
    return x;
  else return y;
}
```

Inline function (Used before main function)

In C++, these functions are designed to speed up the program execution ~~proe~~ process. A function is made inline by writting keywoard "inline" before that function but that function must have at most one or two line code only. Bigger functions can't be inline functions. Unlike other functions, when an inline function is called, the the computer copies entire code of inline function and postes it where it is called and executes it but in normal function call, program flow goes to that function which is called and executes it and finally returns back to the line from where it was called before. Thus execution time is more in normal function call. Inline function syntax is

```
inline return-type function_name ()
{
  function body ;  // one or 2 lines only
}
```

Inline function does not work for following situations:

→ Functions having branches or loops or goto statements, switch stat,
→ If function is recursive
→ For functions having static variables

Benefits

→ Fast program execution by eliminating the call & return process.

Drawbacks

→ Consume more memory space since code is inserted at each function call location.

```
# include <iostream.h>
# include <conio.h>
inline void sum (int a, int b)
{   int sum = a+b;
      return sum;
}
   void main()
{ int x, y, k;
cin >> x >> y;
k = sum (x,y);
cout << k;
getch();
}
```

Function Overloading

Overloading means use of same name for different purposes. In C++ overload functions refer to a function having one name and more than one distinct meaning. Overload functions have same name and are differentiable by the type of arguments. It is also called function polymorphism in C++.

e.g.

```
# include <iostream.h>
# include <conio.h>
   float volume (float);
   float volume (int, float);
   float volume (float, float, float);
   void main()
{ float l, b, h, s ; int r;
  cout cin >> l >> b >> h >> s >> r ;
cout << " volume of cylinder = " << volume (r, h); << end l;
cout << " volume of cube = " << volume(s); << end l;
cout << " volume of rect. box = " << volume (l, b, h);
}
   float volume (int r, float h)
    { float v = 3.14 * r * r * h;
        return v;
    }
float volum (float s)
 { float v = s * s * s;
    return v;
 }
```

```
float volume (float l, float b, float h)
{ float v = l*b*h;
    return v;
}
```

Dynamic Memory allocation with new and delete

* Static Memory allocation
→ When we know about of memory to be allocated beforehand and the memory is allocated during the compilation itself, is called static memory allocation.
→ for this purpose
int num1; char name[20]; etc.

* Dynamic memory allocation (DMA)
When we don't know about the ~~ber~~ memory to be allocated beforehand and the required memory is allocated during runtime/execution is called dynamic memory allocation.
→ In C++, we use "new" & "delete" operators for this purpose
→ The operator "new" allocates the memory dynamically and returns a pointer storing memory address of allocated memory.
→ The operator "delete" deallocates the memory pointed by the given pointer (reverse the effect of new)

→ The general form of using "new" operator is:

```
data-type * pointer_variable;
pointer_variable = new data-type, // allocating a single variable
```

→ For array

```
pointer_variable = new data type[size] ; // allocates with the size elemen
```

→ The general form for 'delete' operator

```
delete pointer_variable;
    or,
delete [size] pointer_variable ; // for array
or, delete [ ] pointer_variable ;
```

e.g. WAP to add two numbers using DMA.

```
#include <iostream.h>
#include <conio.h>
   Using namespace std ;
int main()
{
int * pa * pb, * psum ;
pa = new int;
pb = new int;
psum = new int;
cout << " enter two integers" << endl;
cin >> *pa >> *pb;
```

```
*psum = *pa + *pb;
cout << "sum = " << *psum;
delete pa;
delete pb;
getch;
return 0;
}
```

WAP to read 5 integers and sort them in ascending order using DMA.

```
# include <iostream.h>
# include <conio.h>
    int main ()
{
  int i, j, temp, *ptr;
   ptr = new int [5];
  clrscr ();
 cout << "enter 5 integers";
for (i=0; i<5; i++)
   cin >> ptr [i];
for(i=0; i<4 ; i++)
{
 for(j = i+1, j<5 ; j++) ||for(j= i+1, j<5; j++)
 { if (ptr [j] < ptr [i])
  {
    temp = ptr [i];
```

```
        ptr[i] = ptr[j];
        ptr[j] = temp;
      }
    }
  }
  cout << "numbers in ascending order are" << endl;
    for(i=0; i<5; i++)
      cout << ptr[i] << "\t";
       delete [] ptr;
      getch();
       return 0;
    }
cout << "numbers in ascending order are" << end;
```

Unit 3

Concept of Object and Class

Introduction to Class

OOP incapsulates data and function into a single unit called class. A programmer can create different objects of a class that have their own data & function to operate on the data.

The data of a class are called data members & functions are called function members.

In OOP emphasis is given on data rather than the function.

Syntax of a class

```
class class_name
{
    private: data_members;
             function_members;
    public : data_members;
             function_members;
};
```

```
int main()
{
  name of class obj1, obj2, obj3
  obj1. fun_name();
  :  :  :
  getch();
}
```

```
class student
 {
   private : int roll;
             float marks;
             char name[20];
 public: int input( );

   public :  int x;
             void display enter.( )
            { cout<<"input data";
              cin >> x >>roll >>marks >>name ;
            }
              void display ( )
            { cout<<x<<roll<< marks<< name ;}
    };
                          int student :: input ( )
                           cout<<"Ram is a boss";
int void main ( )
   {
        student  S1, S2, S3;
       S1.input ( );
       S2 . enter( );
       S3 . display ( );

     return 0;
```

```
#include <iostream>
using namespace std;
class student
{ private : int roll;
            float marks;
            string name;
  public : int x;
           int input();
           void enter()
    { cout << " input data ";
      cin >> x >> roll >> marks >> name; }
      void display()
    { cout << x << endl << roll << endl << marks << endl << name; }
};
int main()
{ student s1, s2, s3;
  s1.input();
  s2.x = 9;
  s2.enter();
  s3.enter();
  s2.display();
  s3.display();
  return 0;
}
  int student::input()
 { cout << "ram is boss" << endl; }
```

→ The data under private section can be accessed by the fundions defined inside the class.

→ If data are in private section of a class, those data cannot be accessed directly from outside of the class through the object of a class. So to access those data we use object to call the function declared or defined in public section of a class.

Namespace

It defines the scope for the identifess that are used in program. To use the ideutifers defined in namespace's space, we must include the "using" directive like,

```
using namespace std;
```

Below the header files and above the class defintion we use this.

Here "std" is the namespace called standard namespace where all standard class library are defined. All C++ programs must include this directive & this will bring all identifiers in "std" to the current global scope. Here "using" & "namespace" both are the keywords in C++.

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

using namespace std;
{
    - - -
}

main()
{ - - -
  - - - -
}
```

Object as function argument

We can pass object of a class as a function argument in two ways.

1) A copy of entire object is passed to the function.
2) Only address of object is passed to the function.

The 1st method is called pass by value method since copy of object is passed to the function and any changes made to the object inside the called function do not affect the object used in calling function.

The 2nd method is called pass by reference method When an address of object is passed the called function directly works on the actual object used in function call. This means that any changes made to the object inside the function will reflect back in the actual object. The pass by reference method is more efficient since it requires to pass only the address and not the entire object.

Program using structure

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

  struct student
{   int roll, x;
     float marks;
      char name[10];
   } s1, s2;
    void input()
{
  cout<<"ram is a boss";
}
    student enter (student s)
{ cout <<"input data";
  cin >> s.roll >> s.x >> s.marks >> s.name;
   return s;
}   void display (student s)
{ cout << s.roll<< s.x << s.marks << s.name; }

   int main()
   { input ();
      s1 = enter (s1);
      s2 = enter (s2);
       display(s1);
       display (s2);
      return 0;
   }
```

Write a program to add two times given in the form of hour and minutes to illustrate the concept of passing objects to the function as a argument.

→

```
#include <iostream>

using namespace std;
class time
{ int hours, minutes;
public : void get_time (int h, int m)
        { hours = h; minutes = m; }

        void display ()
    { cout << hours << "hours and" ;
      cout << minutes << " minutes" << endl;
    }
    void sum (time, time);
};
    void time :: sum (time t1, time t2)
{ minutes = t1.minutes + t2.minutes;
  hours = minutes/60;
  minutes = minutes % 60;
  hours = hours + t1.hours + t2.hours;
}
  int main()
{ time T1, T2, T3;  //these are identifier so we can write any capital too
  T1.get_time (8, 50);
```

```
T2.get_time(2,40);
T3.sum(T1,T2);
T3.display();
getch;
}
```

O/P:
11 hours & 30 minutes

Static data member and function member

1) Static Data

→ The data member which is globally shared of all objects of that class.

→ They are usually used to store the values that are common to the entire class.

Characteristics of static data members

- Only one copy of static data member is created for entire class and is shared by all objects of that class
- By default it is initialized to zero when first object of a class is created.
- Its lifetime is over the entire program.

2) static variable declaration

— Declared inside the class by putting the keyword "static" before the data type of that variable

Definition of static variable

Static variable must be defined outside of the class.

```
data_type class-name :: variable-name = value;
```

e.g. `int student :: x = 7;`

☆ Write a program to illustrate the concept of static data members with the help of class and object.

```
class item
{ static int count;
    float price;
public : void getdata (float a)
        { float price = a;
          count ++;
        }
    void showcount()
{
  cout << "count = " << count << endl;
  }
};
    int item :: count;
    int main()
```

```
{   item i1, i2, i3;
    i1.showcount();
    i2.showcount();
    i3.showcount();
    i4.showcount();
    i1.getdata(1,1);
    i2.getdata(1,2);
    i3.getdata(5,4);
cout<<endl<<" after reading data"<<endl;
    i1.showcount();
    i2.showcount();
    i3.showcount();
    getch();
}
```

Memory

i_1	i_2	i_3
count = 0	count = 0	count = 0
price = 2.1	price 1.2	price 5.4
count=3	count = 3	count=3

Output

```
count = 0
count = 0
count = 0
After reading data
count = 3
count = 3
count = 3
```

static Function

Static Member function

- It is a member function which can access only static members (either data member or function) of the same class.
- The keyword "static" is written before the return type of a function to make it static function.
- To call static function we use class name followed by scope resolution operator instead of dot operator to call the particular function.

Syntax: class name :: function-name();

```
class BE
{  int code;
   static int count;
   public : void setcode()
            { cout++;
              code = count++;
            }
   void showcode()
{ cout<<" student code :"<<code<<"\BE1077"<<endl;}
   static void showcount()
   {cout<<"records of"<<count<<"students found"<<endl;}
};  int BE :: count;
int main()
{ BE C1, C2;
  C1.setcode();
  C2.setcode();
BE :: showcount(); // calling static f^n using class
```

```
BE C3;    // object creation
C3.setcode();
BE::showcount();
C1.showcode();
C2.showcode();
C3.showcode();
getch();
}
```

output

```
Records of 2 students found
Records of 3 students found
student code: 1/BE/077
student code: 2/BE/077
student code: 3/BE/077
```

```
BE C3;      // object creation
   C3.setcode();
   BE::showcount();
   C1.showcode();
   C2.showcode();
   C3.showcode();
   getch();
}
```

output

```
Records of 2 students found
Records of 3 students found
student code: 1/BE/077
student code: 2/BE/077
student code: 3/BE/077
```

Friend function (of a class)

- It is a function which is not the member of a class but has full access to the private, public as well as the protected members.
- The functions are made friend by just writting the keyword friend before the returntype of a function in function declaration or definitions.
- A function can be declared as a friend of any number of classes

Characteristics of friend function

- It is not in the scope of class of which it is friend.
- Since it is not in the scope of class, it cannot be used called using object of that class but it is called like normal function without using object.
- Friend function takes object as a argument to access the data of the class of ~~thi~~ which it is friend.

Friend Function Declaration and Definition

```
class ABC
{ - - -
  - - -
public: friend void xyz(arguments); // FF declaration
        - - -
};      - - -
        - - -

void xyz(argument)  // FF definition
{ - - -
  - - -
```

☆ WAP to find the average of data of a vector class using friend function.

```
class vector
{ int a[10]
  float arg;
public: void getdata();
  friend void average(vector); // FF declaration
};
```

```
void vector :: getdata ()          // definition of fn
{ for (int i=0; j<10; i++)
      cin >> a[i]
}
void average (vector v)            // friend fn definition
{ float sum = 0.0;
  for (int i=0; i<10; i++)
  { sum += v.a[i]; }
    v.avg = sum/10;
    cout << " average = " << v.avg;
  }
}
int main ()
{ vector obj;
  cout << " enter 10 integer";
  obj.getdata ();                  // function call
  average (obj);                   // friend fn call passing obj
  getch ();                          as argument */
}
```

Friend Function as a bridge

→ A function can be declared as friend in two or more classes so that it can operate on private members of those classes.

→ In this case friend function take object of two or more classes as argument.

* WAP to add data members of two different classes with the help of the function i.e. friend with the both of the classes.

```
class mango;          // class declaration
class apple
{ int x;
  public: void get_data1()          //fn defn.
          { cout<<"enter value of x";
            cin>>x;
          }
    friend void add(apple, mango); // fn declaration
};
  class mango
{ int y;
  public : void get_data2()  // fn defn  // class defn
          { cout<<"y = ? "<<endl;
            cin>>y;
          }
    friend void add (apple, mango); // fn decl
};
  void add(apple, a1, mango m1) // friend
{ int sum = a1.x + m1.y;        function defn x)
   cout<<" sum = "<<sum;
}
  int main ()
{ apple obj 1;
```

```
mango obj2;
obj1.get_data();          // function call
obj2.get_data2();
add(obj1, obj2);          // friend function call
getch();
}
```

Friend Class

- Entire class can be declared as friend with another class
- When a class is declared as a friend it can access private as well as public members of a class.
- We declare friend class when its has to share its all members functions with another class.

```
class abc
{ - - -
}
public: - - -
};
class xyz
{ - - -
public: - - -
friend class abc;
};
```

e.g.

```
class B;
class A
{ int x,y;
public: void enter()
{cin >> x >> y }
friend class B;
};
class B.
{public: void display(A l)
{cout << "x = " << l.x << endl;
cout << "y = " << l.y << endl;
}
};
```

```
int main
{ A obj1;
  B obj2;
  obj1.enter();
  obj2.display(obj1);
  getch() return 0;
}
```

Nesting of class

```
class student
{ int roll;
  char name[20];
public : void enter()
  {cout << "enter name and roll";
     gets(name);
      cin >> roll;
  }
  void display ()
{ cout << endl << "name = " << name << "roll = " << roll;
}
    class date
  { int day, month, year;
    public : void get data();
             void showdata();
  };
};
```

```
void student :: date :: getdata ()
{ cout << "enter date";
  cin >> day >> month >> year;

void student :: date :: showdata()
{ cound << day << month << year ; }

int main()
{ student ram;
  student :: date d1;
    ram.enter();
    d1.getdata();
    ram. display ();
    d1 . showdata();
  } return 0;
```

Returning Object From Function

```
class coordinate
{ int x, y;
public : void input ()
 { cin >> x >> y; }
void show ()
{ cout << x << y; } ;
friend coordinate sum (coordinate, coordinate); };
coordinate sum (coordinate a, coordinate b)
{ coordinate z;
  z.x = a.x + b.x ;
  z.y = a.y + b.y;
  return (z)
}
```

```
int main()
{ coordinate i,j,k;
  i.input();
  j.input();
  k=sum(i,j);
  i.show();
  j.show();
  k.show();
  return 0;
}
```

Constructor

- These are the special type of function that are used to construct the object.
- Constructor do not have return type not even void.
- The name of constructor is same as that of the class which declared or defined.
- Constructors are defined or declared on public section once it is declared inside the class it must be define outside of the class
- Since it construct the abstracted data of a class so it is called constructors

Characteristics of Constructor

- Constructors are called automatically when objects of relevent class are relevant.
- Constructors cannot be static.

- Constructors can call the member function of class
- Constructors cannot be virtual.
- Constructors call the operators new and delete implicitly when memory allocation and deallocation is required.

Types of Constructor
1) Default Constructor
2) Parametirized Constructor
3) Copy Constructor

1) Default Constructor
- The constructor that do not take any arguments is called default constructor.
- If there is no any default constructor is class, then the default constructor is supplied by compiler itself.

e.g.
```
class abc
{
    data members
    public:
        abc(); //default constructor declaration
        abc(parameters); //parameterized constructor de
};
abc::abc() //default constructor defn
{---}
abc::(parameters) //parametarized constructor defn.
{...
...
}
```

Paratmelarized Consbructor

- The constructor that takes argument is called parametarized constructor.
- Parametarized constructor is called automatically when an object is initialized by parameter value during its creation.
- Constructors never take the argument of type class but they may have the parameters that are reference to the class.

```
class constructor
{ int a,b;
  public : constructor();
           constructor(int, int);
           void display()
    { cout<<" a = "<<a<<endl;
      cout<<" b = "<<b;
    }
};
  constructor :: constructor()
{ a = 0;
  b = 0;
}
  constructor :: constructor(int x, int y)
{ a = x;
  b = y;
}
```

```
int main()
{ constructor C1, C2;
  constructor C3(2,9), C4(7,3);
  C1.display();
  C2.display();
  C3.display();
  C4.display();
    getch();
}
```

Note: Default constructor are called automatically when we create object of a class and the variables are initialized to dummy values by the compiles

Copy Constructor

- The constructor that takes refeuence to the object as a argument when copies a values of one object to another object.

```
class constructor
{ : : : };
int main()
{ constructor C1, C2, C3;
  C1 = C2;
  C3 = C2;
  : : :
}
```

- Copy constructor are called automatically when an object is initialized to another object.
- If a copy constructor is not created then compiler automatically creates it. and copies the values of one object to another.

```
class student
{ int id;
 public:
        student () { }
       student (int a): id (a) { } // student (int a) { id = a; }
        student ( student & s )
       { id = s.id; }
    int display ()
{ return (id);
}
   int main ()
{ student  s1(200);
   student  s2(s1); // s2 = s1 , copy constructor call
   student  s3 = s2;
   student  s4 = s1;
cout << " id of student s1 = " << s1.display ();
cout << " id of student s2 = " << s2.display ();
cout << " id of student s3 = " << s3.display ();
cout << " id of student s4 = " << s4.display ();
getch ();
}
```

If two or more objects are involved in function call, then left most object calls the function.

Destructor

- Destructor is special type of function that do not have a return type (like constructor) and is called automatically when an object goes out of scope.
- It has the same name as that of class.
- Destructors do not have arguments.
- Destructors are called in reverse order of constructors.

Syntax

```
class abc
{ ...
  ...
public: ...
    ~abc() // desctructor
  { ...
    ...
  } ...
```

- Destructor can be declared or defined inside or outside of the class like constructor.

• eg.

```
class dept-store
{ static int count ;
   int id ;
   float price ;
   public : dept-store (float p)
        { price = p
          count ++ ;
            id = count ;
  cout << "object with id = " << id << "created" << endl;
  } ~ dept-store ()
  { cout << "object with id = " << id << "destroyed" << endl;
  }
};
   int dept-store :: count = 0 ;
   int main
{
  { dept-store obj 1 (1.2), obj 2 (2.7), obj 3 (10.3);
    { dept-store obj 4 (9.2);
    }
  } getch ();
}
```

that pointer to address of variable

Pointer to object and member access

→ To create object pointer first we create the pointer variable of type class of which the object is to be pointed out.
→ Then we create object and put the address of object on the pointer variable.
→ To access the members (data or function) we use '→' operator instead of dot (.) operator with the function or data which is to be accessed.

```
class_name *ptr;
class_name obj1, obj2;
ptr = &obj1;
  - - -
ptr → function_name(); // obj1.function_name();
   or
(*ptr).function_name();
```

Example given,

```
class BEE
{ int roll, mark;
  public: void getdata()
          { cin >> roll >> mark; }
  void showdata()
  { cout << roll << mark; }
int main()
{ BEE *ptr, obj1;
  ptr = &obj1;
  ptr → getdata(); //(*ptr).getdata();
  ptr → showdata();
  getch();
}
```

Dynamic Memory Allocation For Objects and Object Array