Module-2 (Functions with C++)

Introduction:

→ Dividing program into functions is one of the major principle of top-down, structure programming.

→ Other advantage is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Syntax:

```
void show( ); /* Function declaration
main()
{
   show();    // Function call
}
void show()   // Function definition
{
              / Function body
}
```

→ when function is called, control is transferred to the first statement in the function body.

→ The other statement in the function body are then executed and control returns to the main program when the closing brace is encountered.

→ Building blocks of C++ program

→ C++ has added many new features to functions to make them more reliable and flexible.

→ C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it.

→ We can declare a function with an empty argument list.

Ex:

```
void display();
```

→ This means that function does not pass any arguments

```
void display(void)
```

CALL BY REFERENCE:

→ Provision of the reference variables in C++ permits us to pass parameters to the functions by reference

→ When we pass argument by reference, the 'formal' argument in the called function becomes aliases to the actual arguments in the calling function.

→ This means that when the function is working with its own arguments, it is actually working on the original data.

Example

```
#include<iostream>
using namespace std;
void swap(int &a, int &b)
{   int t= a; // Dynamic initialization
    a=b;
    b=t;
}
int main()
{
```

```
int main()
{
    int x, y;
        x=10, y=20;
    swap (x, y);
  cout<< "x=" << x << "y=" << y << endl;

return 0;
}
```

Return by reference:

→ A function can also returns a reference

Ex:

```
int & max(int &x, int & y)
{
    if(x > y)
     return x;
    else
      return y;
}
int main()
{ int a=20, b=30;
  int z;
   z= max(a, b);
  cout<< z << endl;

return 0;
}
```

→ In the sample program

* return type of max() is int &, the function returns reference to x (or) y (and not the values)

→ Then a function call such as max(a, b) will yield a reference to either a (or) b depending on their values.

Ex: max(a, b) = -1

⇒ is legal and assigns '-1' to 'a' if it is larger, otherwise '-1' to b.

INLINE Function

→ one of the objective of using function in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.

→ Every time function is called, it takes registers, pushing arguments into the stack, and returning to the calling function.

→ when a function is small, a substantial percentage of execution time may be spent in such overheads.

→ C++ has a solution to this problem.

→ To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function

→ An inline function is a function, ~~when a function~~ that is expanded in inline when it is invoked.

→ That is the compiler replaces the function Call with the corresponding function code (Something similar to macros expansion).

Syntax:

```
inline function-header
{
    function body;
}
```

Example:

```
inline double cube(double a)
{   return (a*a*a);
}
int main()
{   double c, d;
    c= cube (3.0);          // c = 27
    d= cube (2.5+1.5); // d = 64
}
```

** → The speed benefits of inline functions diminish as the function grows in size.

~~→ At some~~

```
#include<iostream>
using namespace std;
inline float mul(float x, float y)
{
    return (x*y);
}
inline double div(double p, double q)
{
    return (p/q);
}
int main()
{
    float a = 12.345;
    float b = 9.82;
    cout << mul(a,b) << endl;
    cout << div(a,b) << endl;
    return 0;
}
```

Some of the situations where inline may not work

1) For function returning value if a loop, a switch, (or) a goto exists
2) For functions not returning values, if a return statement exits.
3) If function contains static variable.
4) If inline function are recursive

Default Arguments:

→ C++ allows us to call a function without specifying all its arguments.

→ The function assign a default value to the parameter which does not have a matching argument in the function call.

→ Default values are specified when the function declared

→ Compile looks at the prototype to see how many arguments a function uses and alters the program for possible default values.

Example

float amount (float P, int n, float r = 0.15);

→ default value is specified in a manner syntatically similar to a variable initilization.

→ above prototype declares a default value r=0.15.

→ A subsequent function call like

value = amount (5000, 7); // one argument missing.

→ passes the value of 5000 to 'p' and 7 to 'n' and then lets the function use default value of 0.15 for 'r'.

→ The call

value = amount (5000, 5, 0.12); // no arguments missing.

passes an explicit value of 0.12 to 'r'.

→ Default argument is checked for type at the time of declaration and evaluated at the time of call.

→ One important point to note is that only the Trailing arguments can have default values and therefore we must add defaults from Right to Left.

Example

```
int mul (int i, int j=5, int k=10); // legal
int Mul (int i=5, int j); // illegal.
int mul (int i=0, int j, int k=10); // illegal
int mul (int i=2, int j=5, int k=10); // legal.
```

Ex: Bank intrest may remain the same for all customers for a particular period of deposit.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    float amount;
    float value(float p, int n, float r=0.15); //prototype
    void printline(char ch='*', int len=40); //prototype
    printline(char ch='*', int len=40);
    printline();
    amount = value (5000.00, 5)
    cout << "Final value=" << amount << endl;
    printline('=');
    return 0;
}
```

```
amount = value (10000.0, 5, 0.30);
cout << "Final value=" << amount << endl;
```

```
float value (float p, int n, float r)
{
  int year=1;
  float sum=0, float Pf;
  Pf= P
  while (year<=n)
  {
    sum= Pf * pow((1+r), n);
    year++;
    Pf= sum;
  }
  return (sum);
}
void printline (char ch, int len)
{ for (i=1; i<=len; i++)
    cout<< ch;
}
```

Recursion:

→ Recursion is a situation where a function calls itself.

→ It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, a recursive function also possesses a base case which returns the program control

- from the current instance of the function to call back to the calling function.

Ex:① Calculating factorial of a Number

```
#include<iostream>
using namespace std;
long int fact (int n)
{
    if (n==0)
    return 1;
    else
    return (n * fact (n-1)); //recursive function call
}
int main()
{
    int num;
    cout<< "Enter a positive number"<<endl;
    cin>> num;
    cout<<" factorial of "<< num <<" is "<< fact (num);
    return 0;
}
```

Ex ① Solving Tower of Hanoi problem

```
#include<iostream>
using namespace std;
void toh(int d, char t1, char t2, char t3)
{
  if(d==1)
  { cout<<"Shift to disk from tower"<<t1<<"to
    tower"<<t2<<endl;
    return 0;
  toh(d-1, t1, t3, t2);
  cout<<"Shift top disk from tower"<<t1<<"to
  tower"<<t2;
  toh(d-1, t3, t2, t1);
}
int main()
{ int disk;
  cout<<"Enter the number of disks"<<endl;
  cin>>disk;
  if(disk<=1)
    cout<<"There are no disks to shift"<<endl;
  else
    cout<<"There are"<<disk<<"In tower1";
  toh(disk, '1', '2', '3');
  cout<<"\n\n"<<disk<<"disks in t1 are shifted to
  t2"<<
}
```

Function overloading

→ This means that we can use the same function name to create functions that perform a variety of different tasks.

→ using the concept of function overloading; we can design a family of functions with one function name with different argument lists.

→ function would perform different operations depending on the argument list in the function call.

→ The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example

declaration

```
int add(int a, int b);    // prototype 1
int add (int a, int b, int c); // prototype 2
double add (double x, double y); // prototype 3
double add (int p, double q); // prototype 4
double add (double p, int q); // prototype 5.
```

Function calls

```
cout<< add(5,10): // uses prototype 1
cout<< add(15, 10.0): // uses prototype 4
cout<< add(12.5, 7.5); // prototype 3
cout<< add (5, 10.15); // uses prototype 2
cout<< add (0.75, 5); // uses prototype 5
```

→ A function call first matches the prototype having the same number and type of arguments and then call the appropiate function for execution.

→ A best match must be unique.

→ The function selection involved the following steps:

1) The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.

2) If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

char to int
float to double to find a match.

3) When either of them fails, the compiler tries to use the built-in conversion to the actual arguments and then uses the function whose match is unique.

If the conversion is possible to have multiple matches then the compiler will generates an error message

Eg:

```
long square (long n)
double square (double x)
```

A function call is

```
Square (10);
```

→ will cause an error because int argument can be converted to either long (or) double. Thus creating an ambiguous situation as to which version of Square() should be used.

4) If all steps fail, then the compiler will try the user-defined conversions in combination with Integral promotion and built-in conversions to find a unique match.

---

① write a function to read matrix of size mxn from keyboard.

② write a program to read a matrix of size mxn from the keyboard and display the same on the screen using functions.

③ write a macro that obtains the largest of three numbers.

④ write a function power() to raise a number 'm' to a power 'n'. The function takes a double value for 'm' and 'int' value for 'n', and returns the result correctly. Use a default value of '2' for 'n' to make the function to calculate Squares when this

arguments is omitted. Write a main that gets the values of m and n from the user to test the function.

④ Write a program to compute the area of a triangle and a circle by overloading the area() function.

Specifying a class

→ A class is a way to bind the data and its associated functions together.

→ It allows the data and functions to be hidden

→ when we defining a class, we are creating a new abstract data type that can be treated like any other built-in data type.

→ Generally, a class specification has two parts:

1. class declaration
2. class function definitions.

→ class declaration describes the type and scope of its members.

→ The class defination describes how the class functions are implemented.

General form of class declaration

```
class class_name
{   private:
        variable declaration;
        function declaration;
    public:
        variable & declaration;
        function };
```

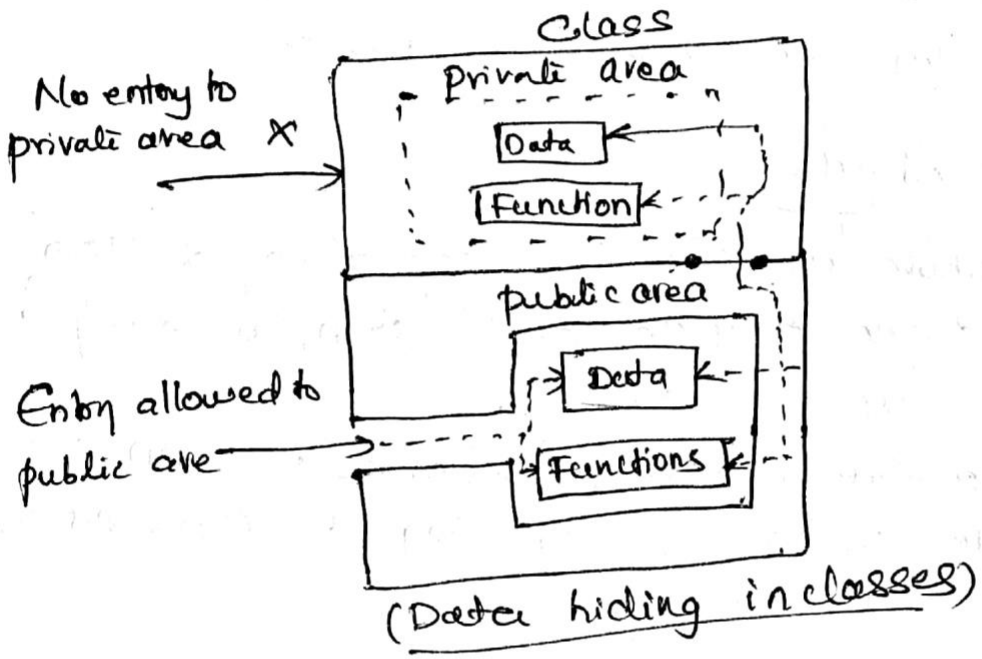
→ body of a class is enclosed within braces and terminated by a semicolon.
→ class body contains the declaration of variable and functions. (class members)

→ Class members grouped under two sections, namely private and public to denote which of the members are private and which of them are public
→ These keywords are called as access specifiers followed with colon (:)

→ ~~The class members~~
→ Private class members can be accessed only within the class
→ Public: class members can be accessed outside the class.
→ Data hiding (using private declaration) is the key feature of object-oriented programming
→ use of the keyword private is optional. By default the members of a class are private.
→ The variables declared inside the class are known as data members and the functions are known as member functions.
→ only member functions can have access to the private data members and private functions

→ public members (both function and data) can be accessed from outside class.

→ The binding of data and functions together into a single class-type variable is refered to as encapsulation.



(Data hiding in classes)

A Simple Class Example

```
class item
{
    int number;          // data members
    float cost;
public:
    void getdata(int a, float b); // member functions
    void putdata(void);
};
```

item → class type

item → consists of two data members and two member functions.

getdata() → used to assign values to the data members

putdata() → display their values.

→ These functions provide the only access to the data members from outside class.

→ data members declared as private.

→ member functions declared as public.

Creating object:

→ from above example the declaration of 'item' does not define any objects of 'item', but only specifies what they will contains.

→ once a class has been declared, we can create variable of that type by using the class name.

Example:

item x; // memory for x is created

→ creates a variable 'x' of type 'item', the class variable is known as objects.

→ where 'x' is called an object of type 'item'

→ we can declare more than one object in one statement

Eg: item x, y, z;

→ object can also be created when a class is defined by placing their names immediately after closed brace.

Ex:

```
class item
{
    ....
    ....
    ....
} x, y, z;
```

Accessing class members

→ private data of a class can be accessed only through the member functions of that class.

→ main() cannot contain statements that access number and cost directly.

Syntax:

object_name . functionname (actual_arguments);

Eg: function call statement is

x.getdata(100, 75.5); // is valid assignment

where number = 100 and cost = 75.5 will be assigned.

Example:

x.putdata(); // in main function.

→ will displays values of data members

→ member function can be invoked only using an object (or) class variable.

Eg:
```
main()
{
    getdata(100, 75.5) // is illegal. gives error.
    item x;
    x.number // error ⇒ number is private
    x.putdata // valid.
    x.getdata // valid;
}
```

Example:

```
class xyz
{
    int x;
    int y;
  public:
    int z;
};
int main()
{
  xyx p;
  p.x=0;// invalid (error) wher 'x' is private.
  p.z = 10;// valid 'z' is public
}
```

DEFINING MEMBER FUNCTION

member function can be defined in two places:

1) outside the class defination.
2) Inside the class defination.

→ ~~It is~~ Outside the class defination:

→ member functions that are declared inside a class have to be defined seperately outside the class

→ Difference blw a member function and a normal function is that a member function incorporates a membership "identity label" in the header.

→ This label tells the compiler which class the function belongs to

Syntax:

```
return_type class_name :: function_name (arguments declaration)
{
  function body
}
```

→ class_name:: → tells the compiler that the function "function_name" belongs to the class "class_name".

:: ⟶ scope resolution operator.

Example:

```
class item
{   int number;
    float cost;
  public:
    void getdata(int a, float b);
    void putdata();
};
void item :: getdata(int a, float b)
{   number = a;
    cost = b;
}
void item :: putdata()
{   cout << "number: " << number << endl;
    cout << "cost = " << cost << endl;
}
```

The member functions have some special characteristics that are often used in the program development.
These are:

* Several different classes can use the same function name. The membership level will resolve their scope.
* Member functions can access the private data of the class. A nonmember function cannot do so.
* A member function can call another member function directly, without using dot operator.

Inside the class definition:

→ Example;

```
class item
{   int member;
    float cost;
public:
    void getdata (int a, float b);
    {   number = a;
        cost = b;
    }
    void putdata ( )
    {   cout << number << endl;
        cout << cost << endl;
    }
main( )
{   item x;
    x.getdata(100, 75.5);
    x.putdata( );
}
```

A C++ program with class

```
#include<iostream>
using namespace std;
class item
{
    int number;
    float cost;
  public:
    void getdata(int a, float b);
    void putdata()
    {
      cout << "number=" << number << endl;
      cout << "cost=" << cost << endl;
    }
};
void item::getdata(int a, int b)
{
    number = a;
    cost = b;
}
int main()
{
    item x, y;
    x.getdata(100, 299.95);
    x.putdata();
    y.getdata(200, 175.50);
    y.pudata();
  return 0;
}
```

→ This class contains two private variables and two public functions

→ The member function getdata() which has been defined outside the class supplies values to both variables.

→ Note the use of statements such as

number = a;

in the function definition of getdata();

* This shows that the member functions can have direct access to private variable data item.

→ Above program creates two objects, x and y in two different statements. This can be combined in one statement.

item x, y

o/p: object x
number : 100
cost : 299.95
object y
number : 200
cost : 175.5

---

* Making outside function inline.

Nesting of Member Functions

Ex:

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;
class binary
{ string s;
  public:
    void read()
    {
      cout<<"Enter a binary number"<<endl;
      cin>>s;
    }
void check_bin(void)
{ for(int i=0; i<s.length(); i++)
  { if((s.at(i)!='0')&&(s.at(i)!='1'))
    { cout<<"Incorrect binary number formats"<<endl;
      exit(0);
    }
  }
}
void ones(void)
{ check_bin();
  for(int i=0; i<s.length(); i++)
    { if(s.at(i)=='0'
```

```
        s.at(i) = '1';
    else
        s.at(i) = '0';
    }
}
void display_comp(void)
{
    ones();
    cout << "The 1's complement of the above binary number is " << s << endl;
}
};
int main()
{
    binary b;
    b.read()
    b.disp_comp(void)
    return 0;
}
```

* private member function

Arrays within a class

→ The arrays can be used as member variables in class. The following class defination is valid.

Syntax:

```
const int size=10; // array size.
class array
{   int a[size]; // array of type int
```

```
public:
    void setval( );
    void display( );
};
```

→ array variable a[] declared as a private member of the class array can be used in the member functions, like any other array variable. we can perform any operations on it.

→ The member function **setval()** sets the values of elements of the array a[]

→ **display()** function displays the values of a[],

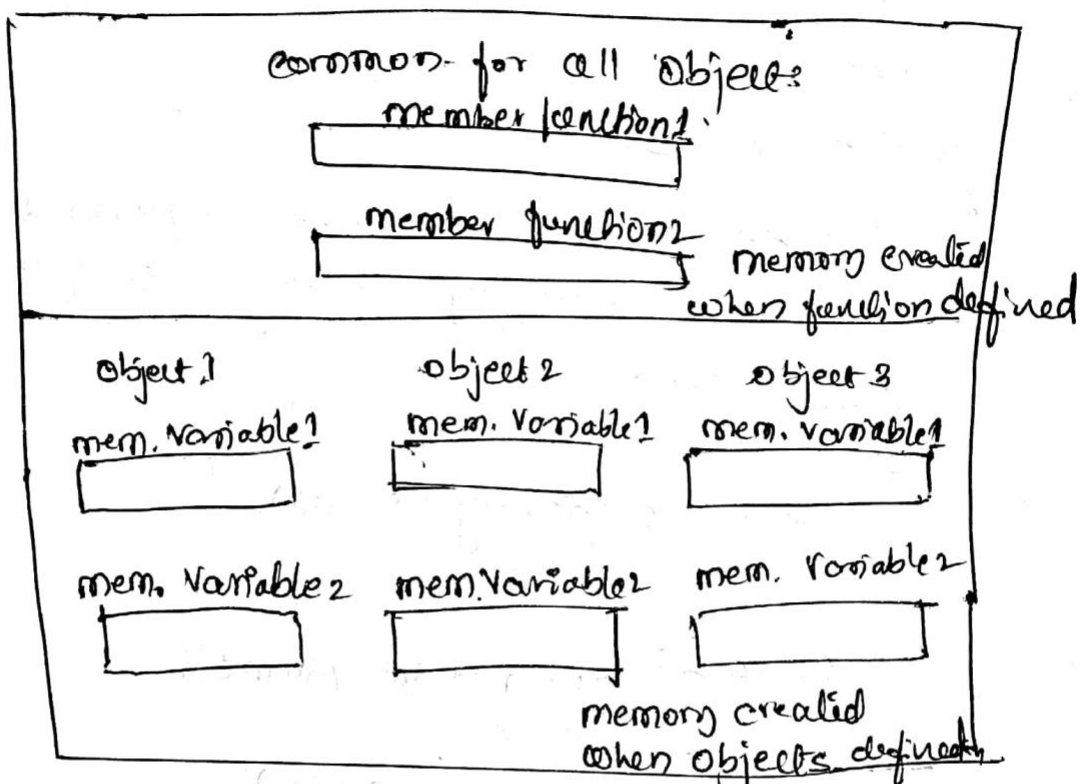
Example: processing shopping list

```
#include<iostream>
using namespace std;
const m=50;
class item
{   int itemcode[m];
    float itemprice[m];
    int count=0;
public:
    void cnt(void) { count=0;}
    void getitem(void);
    void display_sum(void);
    void remove(void);
    void display_item( );
};
```

```
void item :: getitem (void)
{ cout << "Enter item code :" << endl;
  cin >> itemcode [count];
  cout << "Enter item price :" << endl;
  cin >> itemprice [count];
  count++;
 }
void item :: displaysum (void)
{ float sum=0
  for(int i=0; i<count ; i++)
  {   sum = sum + itemprice [i];
      cout << "Total value :" << sum << endl;
}
void item :: remove ( )
{ int a;
  cout << "Enter item code" << endl;
  cin>>a;
  for(int i=0 ; i<count ; i++)
    if (itemcode [i] == a)
        itemprice [i] = 0;
  }
void item :: displayItem (void)
 {  cout << " Code price" << endl;
   for(int i=0; i<count << i++)
    { cout << itemcode [i] << endl;
      cout << itemprice [i] << endl;   } }
```

```
int main()
{
    item order;
    order.cnt();
    int x;  ------> cin>>x;
  while (x!=5)
  { cout<< "1 - get item, 2- Display Sum, 3-> Remove,
            4,- display all item,  5- quit"<<endl;

    Switch (x)
    {
      case 1 ; order.get item():
                 break;

       case 2 : order.displaysum();
                 break;

       case 3 : order.remove();
                 break;

       case 4 : order.displayitem();
                 break;
        case 5 : break;
        default : cout<<"Invalid i/p"<<endl;

    }
  }
 return 0;
}
```

Memory allocation for object



→ Member functions are created and placed in the memory space only once when they are defined as a part of a class specification.

→ Since all objects belonging to that class use the same member functions, no seperate space is allocated for member functions when the objects are created.

→ only space for member variables is allocated separately for each object.

→ Separate memory locations for the objects are essential, because the member variable will hold different data values for different objects.

Static Data members

→ a data member of a class can be qualified as static.

→ The properties of a static member variable are similar to that of a 'C' static variable

A static member variable has certain special characteristics. There are:

* It is initialized to zero when the first object of its class is created. No other initialization is permitted.
* only one copy of that member is created for the entire class and is shared by all the objects of that class. no matter how many objects are created.
* It is visible only within the class, but its lifetime is the entire program.

→ static variables are normally used to maintain values common to the entire class.

Example:

* Static data member can be used as a counter that records the occurrences of all the objects.

```
#include <iostream>
using namespace std;
class item
{
    static int count;
    int number;
public:
    void getdata(int a)
    {
        number = a;
        count++;
    }
void getcount(void)
{
    cout<<"count = "<<count<<endl;
}
};
int item::count;
int main()
{
    item a, b, c;
    a.getcount();
    b.getcount();
    c.getcount();

    a.getdata(100);
    b.getdata(200);
    c.getdata(300);
    cout<<"After reading data"<<endl;
    a.getcount();
    b.getcount();
    c.getcount();
    return 0;
}
```

o/p:

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

* The static variable count should be defined outside the class defination.

* This is necessary because the static data members are stored separately rather than as a part of an object.

* Since they are associated with the class itself rather than with any class object, they are also known as class variables.

* Static variable count=0 when objects are created.

* count is incremented whenever the data is read into an object. since the data is read into object three times, the variable count is incremented three times.

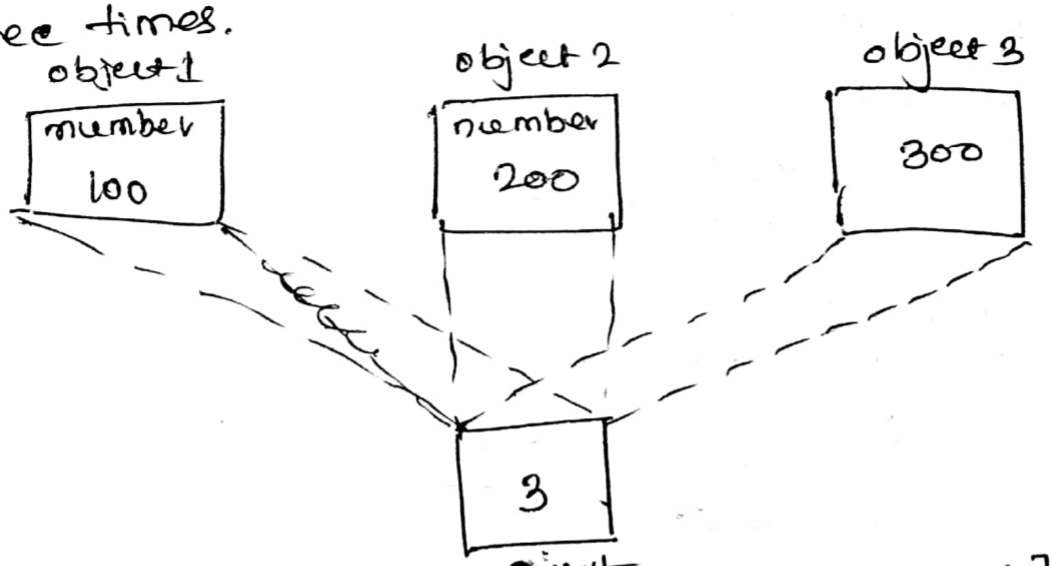


fig: Sharing of a static data member

Static member functions

→ A member function that is declared static has the following properties:

* A static function can have access to only other static members (functions or variable) declared in the same class

* A static member function can be called using the class name (instead of its objects) as follows:

```
class_name :: function name;
```

Example:

```
#include<iostream>
using namespace std;
class test
{   int code;
    static int count;
  public:
    void setcode( )
    {  code = ++count;
    }
    void showcode( )
    {  cout<<" object number="<< code<<endl;
    }
```

```
static void showcount( )
 { cout << "count=" << count << endl;
  }
};
int test :: count;
int main()
{
  test t1, t2;
  t1. setcode();
  t2. setcode();
  test :: showcount();
  test t3;
  t3. setcode();
  test :: showcount();
  t1. showcode();
  t2. showcode();
  t3. showcode();
  return 0;
}
```

o/p:
count=2
count=3
object number=1
object number=2
object number=3

* where the static function show_count() displays the number of objects created till that moment
* A count of number of objects created is maintained by the static variable count.

Arrays of objects:

→ Arrays of variables that are of the type class. Such variables are called arrays of objects.

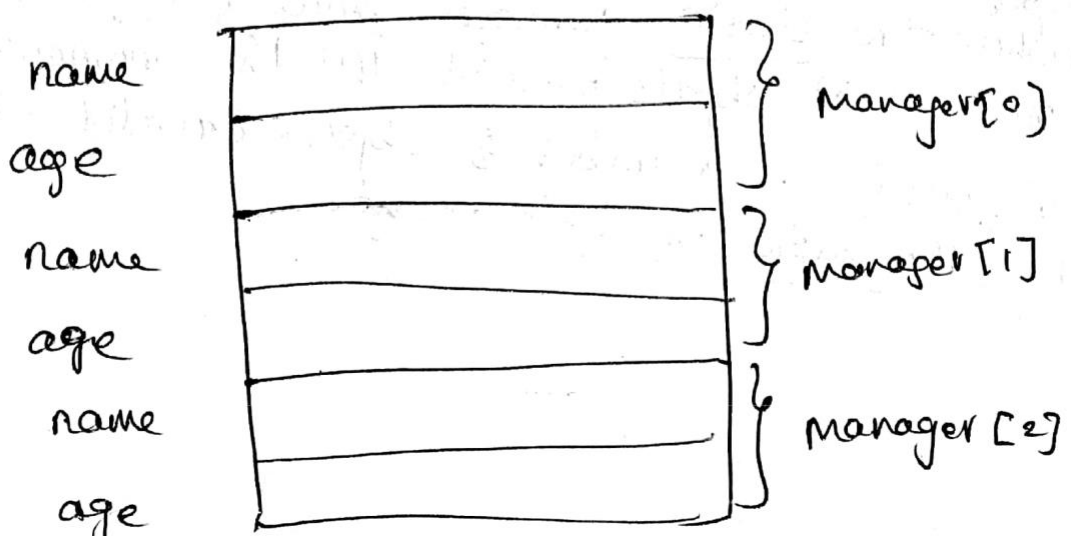
Eg:

```
class employee
{
    char name[30];
    float age;
  public:
    void getdata();        // employee is user defined
    void putdata();          data type.
};
int main()
{
    employee manager[3];
    employee foreman[15];
    employee worker[75];
return 0;
}
```

→ manager contains 3-objects manager[0], manager[1], manager[2]

→

→ Array of objects stored inside the memory in the same way as a multi-dimensional array.



Ex:

```
#include<iostream>
using namespace std;
class employee
{
    char name[30];
    float age;
  public:
    void get_data();
    void put_data();
};
void employee :: get_data()
{
    cout<<"Enter name="<<endl;
    cin >>name;
    cout<<"Enter age"<< endl;
    cin>>age;
}
void employee :: putdata()
{
    cout<<" name="<< name<<endl;
    cout<<" age="<< age << endl;
}
int main()
{
    int size=3;
    employee manager[size];
    for(int i=0; i<size; i++)
    {
        cout<<" Details of Manager"<< i+1 <<endl;
        manager[i].getdata();
    }
    cout<<"display details"<<endl;
    for(int j=0; j<size; j++)
    {
        cout<<"dddd
        Manage[i].putdata()
    }
    return 0;
}
```

object as function arguments

This can be done in two ways:

* A copy of the entire object is passed to the function
* only the address of the object is transferred to the function.

→ The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do effect the object used to call the function.

→ Second method is pass by reference:

* when an address of the object is passed the called function works directly on the actual object used in the call.
* changes made to the object inside the function will reflect for the actual object.

Example:

```
#include <iostream>
using namespace std;
class time
{   int hours;
    int Min;
public:
    void gettime(int h, int m)
    {  hours=h;  Min=m;
    }
```

```
void put time()
{ cout<< hours<< min<<endl;
}
void sum (time, time);
};
void time :: sum (time t1, time t2)
{ min = t1.min + t2.min;
  hour = min/60;
  min = min%60;
  hours = hours+ t1.hours +t2.hours;
}
int main()
{ time T1, T2, T3;
  T1. gettime (2.45);
  T2. gettime (3.30);
  T3. sum (T1, T2);
  cout <<"T1 = "<<endl;
 T1.puttime();
  cout<<"T2="<<endl;
  T2. puttime();
  cout<<"T3 ="<<endl;
  T3. puttime();
return 0;
}
```

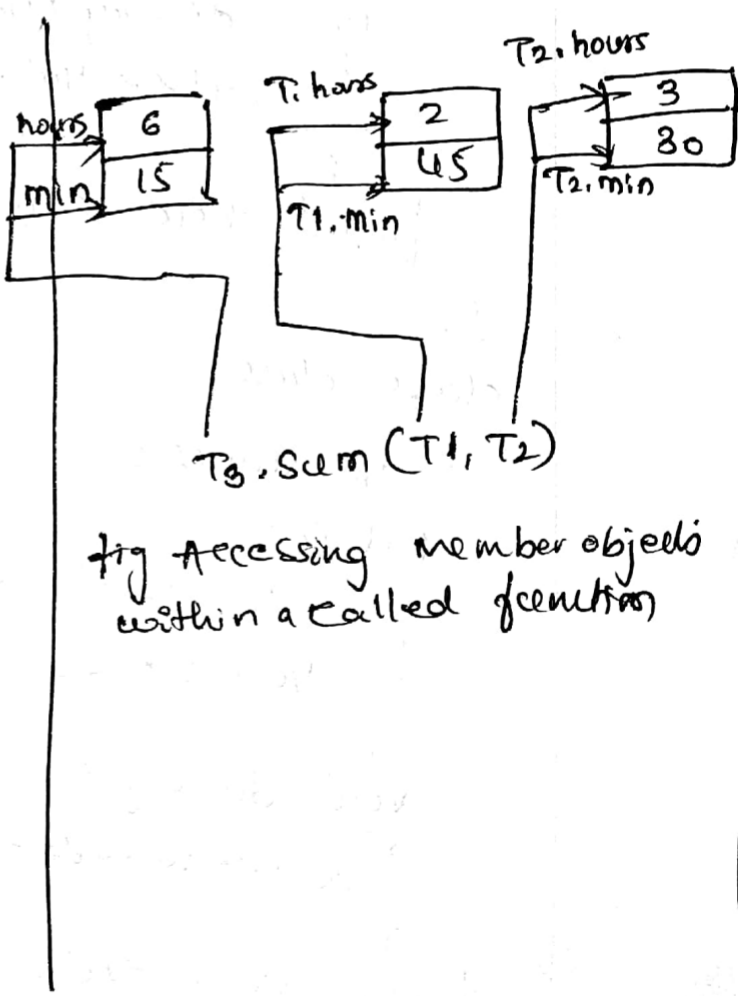


fig Accessing member objects within a called function

Example for friend function

(1) Swapping private Data of classes

```
#include<iostream>
using namespace std;
class class_2;
class class_1
{
    int value1;
  public:
    void indata(int a) {
    { value1 = a;
     }
    void display (void)
     { cout<<"value1 ="<<value1<<endl;
     }
  friend void exchange (class1 &, class_2 &);
};
class class_2
{   int value2;
  public:
  void indata (int a)
  {  value2 = a;
   }
  void display()
  { cout<<"value2="<<value2<<endl;
   }
 friend void exchange (class_1 &, class_2 &);
};
```

```
void exchange(class_1 &x, class_2 &y)
{ int temp = x.value1;
  x.value1 = y.value2;
  y.value2 = temp;
}
int main()
{ class_1 c1;
  class_2 c2;
  c1.indata(100);
  c2.indata(200);
  cout<<"values before exchange"<<endl;
  c1.display();
  c2.display();
  exchange(c1, c2);
  cout<<"values after exchange"<<endl;
  c1.display();
  c2.display();
  return 0;
}
```

Const Member function

→ If a member function does not alter any data in the class, then we may declare it as const member function as follows;

```
void mul(int, int) const;
double get_balance() const;
```

→ the qualifier const is appended to the function prototype (both in declaration and defination).