1. **Memory Allocation for Objects**

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the 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 the objects belonging to that class use the same member functions, no separate 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 variables will hold different data values for different objects.



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**2) 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. These 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. For example, a static data member can be used as a counter that records the occurrences of all the objects.

***Example Program***

class item

{

static int count;

int number;

public:

void getdata(int a)

{

number = a;

count ++;

}

void getcount(void)

{

cout « count;

}

} ;

int item :: count; *// count is initialized to zero*

void main ()

{

item a, b, c;

a.getcount(); *// display count*

b.getcount ();

c.getcount() ;

a.getdata(100); *// getting data into object* a

b.getdata(200); *// getting data into object* b

c.getdata(300);  *// getting data into object* c

a.getcount (); *// display count*

b.getcount ();

c.getcount();

}

**The output of the Program**

count: 0

count: 0

count: 0

After reading data

count: 3

count: 3

count: 3

Note that the type and scope of each **static** member variable must be defined outside the class definition. 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.*

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 5.4 shows how a static variable is used by the objects.



Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

int item :: count = 10;

**(OR) // WRITE THIS EXAMPLE**

#include <iostream>

using namespace std;

class Demo

{

private:

static int X;

public:

static void fun()

{

cout <<"Value of X: " << X << endl;

}

};

//defining

int Demo :: X =10;

int main()

{

Demo X;

X.fun();

return 0;

}

**Output**

**Value of X: 10**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3) Static Member Functions**

Like **static** member variable, we can also have **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 variables) 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;* |

**EX: class test**

{

int code;

static int count;

public:

void setcode(void)

{

code = ++count;

}

void showcode(void)

{

cout « "object number: " « code; *// static* member *variable*

}

static void showcount(void)  *// static member function*

{

cout « "count: " « count;

}

} ;

int test count;

void mian()

{

test t1, t2, t3;

t1.setcode();

t2.setcode ();

test :: showcount();  *// accessing static function*

t3.setcode() ;

test:: showcount();

t1.showcode();

t2 .showcode·();

t3.showcode();

}

**output**: count: 2

count: 3

object number: 1

object number: 2

object number: 3

The statement code=++count is executed whenever **setcode()** function is invoked and the current value of **count** is assigned to code. Since each object has its own copy of code, the value contained in **code** represents a unique number of its object.

**(OR) // WRITE THIS EXAMPLE**

**Consider the example:**

#include <iostream>

**using** **namespace** std;

**class** Demo

{

**private**:

//static data members

**static** **int** X;

**static** **int** Y;

**public**:

//static member function

**static** **void** Print()

{

cout <<"Value of X: " << X << endl;

cout <<"Value of Y: " << Y << endl;

}

};

//static data members initializations

**int** Demo :: X =10;

**int** Demo :: Y =20;

**int** main()

{

Demo OB;

//accessing class name with object name

cout<<"Printing through object name:"<<endl;

OB.Print();

//accessing class name with class name

cout<<"Printing through class name:"<<endl;

Demo::Print();

**return** 0;

}

**Output**

Printing through object name:

Value of X: 10

Value of Y: 20

Printing through class name:

Value of X: 10

Value of Y: 20

In above program X and Y are two static data members and print() is a **static member function**. According to the rule of static in C++, only static member function can access static data members. Non-static data member can never be accessed through static member functions.

|  |
| --- |
| Remember, the following function definition will not work:  static void showcount()  {  cout «code; // code is not static  } |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4) Arrays of Objects**

We can also have array of variables that are of the type class. Such variables are called *arrays of objects.* Consider the following class definition:

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

employee manager[3]; *// array of manager*

employee foreman[15]; *// array of foreman*

employee worker[75]; *// array of worker*

The array **manager** contains three objects (managers), namely, **manager[0], manager[1]** and **manager[2],** of type **employee** class, the **foreman** array contains 15 objects foremen. The Array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

manager[i].putdata();

will display the data of the ith element of the array **manager.** That is, this statement requests the object **manager[i]** to invoke the member function putdata().

An array of objects is stored inside the memory in the same way as a multi-dimensional array. Member functions are stored separately and will be used by all the objects.



**Write a program to calculate total marks and average of n student*(\* Refer lab program no : 2 to write as example***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**5.) Objects as Function Arguments**

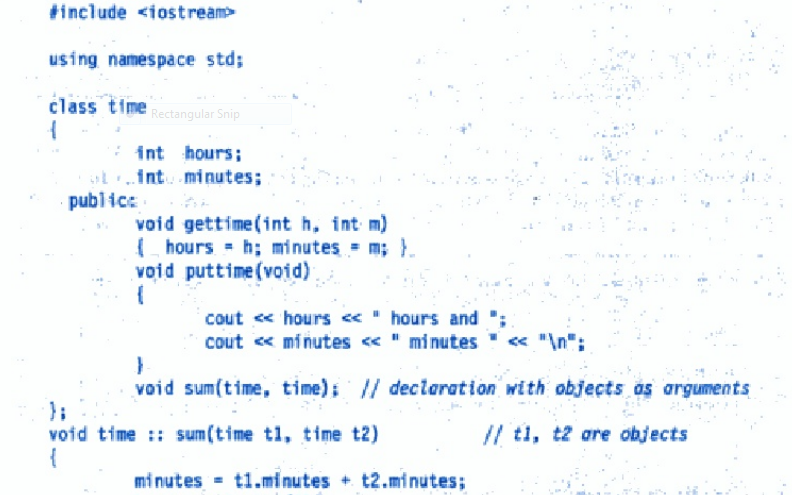
Like any other data type, an object may be used as a function argument. **This can be done in two ways:**

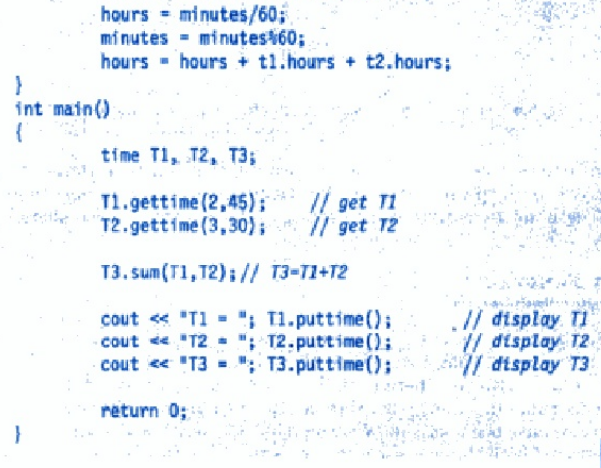
* A copy of the entire object is passed to the function.
* Only the address of the object is transferred to the function.

***pass-by-value*** : 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 not affect the object used to call the function.

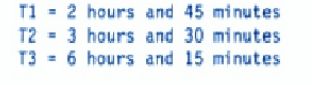
***pass-by-reference :*** The second method is called ***pass-by-reference.***When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object

Ex: Illustrates the use of objects as function arguments. **Write a program to addition of time in the hour and minutes format using object as function argument**





Output:



In this example the member function sum() is Invoked by the object T3, with the objects T1 and T2 as arguments, it can directly access the hours and minutes variables of T3. But, the members of T1 and T2 can be accessed only by using the dot operator (like T1.hours and T1.minutes).

Illustrates how the members are accessed inside the function sum ().



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**6) Friend Function's**

A non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function.

C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

|  |
| --- |
| class ABC  {  public:  **friend** void xyz(void);  *//declaration*  }; |

* The function declaration should be preceded by the keyword **friend.**
* The function is defined elsewhere in the program like a normal C++ function.
* The function definition does not use either the keyword **friend** or the scope operator **::**
* The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

**A friend function possesses certain special characteristics:**

* It is not in the scope of the class to which it has been declared as **friend.**
* Since it is not in the scope of the class, it cannot be called using the object of that class.
* It can be invoked like a normal function without the help of any object.
* Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name
* It can be declared either in the public or the private part of a class without affecting its meaning.
* Usually, it has the Objects as arguments.

**Example :**

class sample

{

int a, b;

public: void setvalue()

{

a=25; b=40;

}

friend float mean (sample s);

};

float mean(sample s)

{

return float(s.a + s.b)/2.0;

}

void main ()

{

sample X; // object X

X.setvalue();

cout « "Mean value "« mean(X) ;

}

Output: Mean value = 32.5

The friend function accesses the class variables a and b by using the dot operator and The function call mean(X) passes the object X by value to the friend function.

**Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator as shown below:**

class X

{

int funl(); // member function of X

};

class Y

{

friend int X :: funl (); // fun1 () at X is friend of Y

};

The function fun1() is a member of class X and a friend of class Y.

***We can also declare all the member functions of one class as the member functions ao another class. In such cases, the class is called*** a “ **friend class. “**

class Z

{

friend class X; *// all member functions of* X *are friends* to *Z*

};

**Ex; demonstrates how friend functions works between the 2 classes.**

class ABC; *//Forward declaration*

class XYZ

{

int x;

public: void setvalue(int i)

{

x = i;

}

friend void max(XYZ, ABC);

} ;

class ABC

{

int a;

public: void setvalue(int i)

{

a = i;

}

friend void max(XYZ, ABC);

} ;

void max(XYZ m, ABC n) // *Definition of friend*

{

if(m.x >= n.a)

cout « **m.x;**

else

cout « n.a;

}

int main ()

{

ABC abc;

abc.setvalue(10);

XYZ xyz;

xyz.setvalue(20);

max (xyz , abc);

}

**The output: 20**

The function max() has arguments from both XYZ and ABC. When the function max() is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

class ABC;

This is known as 'forward' declaration.

Friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

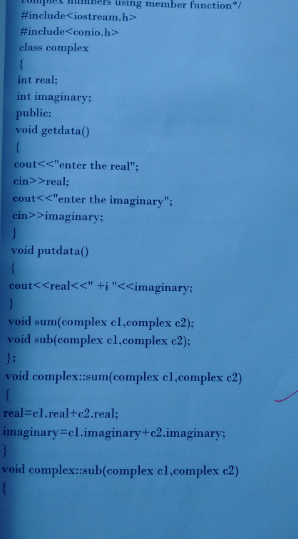
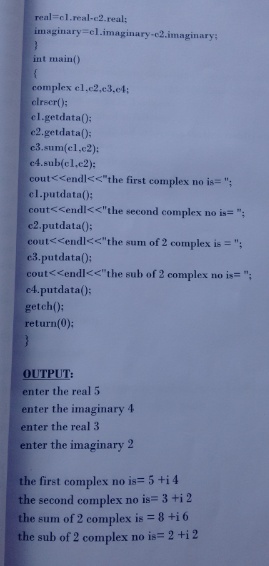
Write a program, common friend function to exchange the private values of two classes using called by reference. (\* refer class notes)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**7) Returning Objects**

A function cannot only receive objects as arguments but also can return them. Illustrates how an object can be created (within a function) and returned to another function**.**

**Write a program to add and subtract a two complex number**

** **

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**8) const Member Functions**

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

|  |
| --- |
| void mul(int, int) const;  double get\_balance() const; |

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**9) Pointers to Members**

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::\* with the class name.

**For example:** class A

{

int m;

public: void show();

} ;

We can define a pointer to the member m as follows:

int A:: \* ip = &A :: m;

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object.

* + A::\* means "pointer-to-member of class".
  + &A::m means the "address of the m member of A class".

**Remember, the following statement is not valid**:

int \*ip = &m;  *//won't work*

This is because m is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member m inside member functions (or friend functions). Let us assume that a is an object of A declared in a member function. We can access m using the pointer **ip** as follows:

cout « a.\*ip; *// display*

cout « a.m; *//* same as *above*

Now, look at the following code:

ap = &a; *// ap is pointer* to *object* a

cout « ap -> \*ip; *// display* m

cout « ap -> m; *// same* as *above*

The ***dereferencing operator* --- >\*** is used to access a member when we use pointers to both the object and the member.

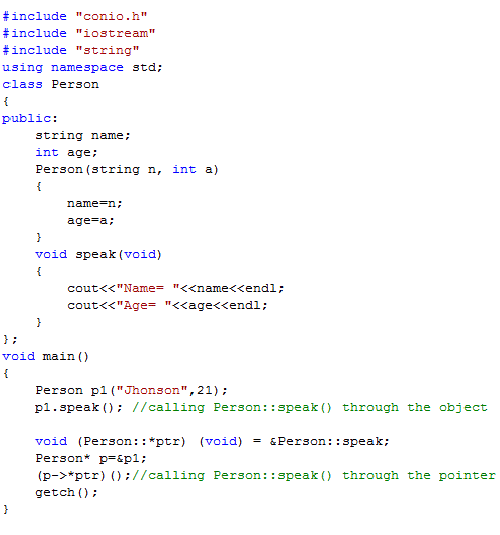
The ***dereferencing* operator .\*** is used when the object itself is used with the member pointer. Note that \*ip is used like a member name.

We can also design pointers to member functions which , then, can be invoked using the dereferencing operators in the main as shown below:

***(object-name .*\* *pointer-to-member function) (10);***

***(pointer-to-object* ->\* *pointer-to-member function) (10)***

**Write a Program the use of dereferencing operators to access the class members** .



Output:

Name= Jhonson

Age= 21

Name= Jhonson

Age= 21