CLASSES & METHODS AND MESSAGE, INSTANCE & INITIALIZATION

Limitation of C Structure

The standard C does not allow the **struct** data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
        float real;
        float imag;
};
struct complex c1,c2,c3;
```

We cannot add/subtract two complex numbers one from other. For example:

```
c3 = c1 + c2;
```

is illegal in C.

They do not permit data hiding. Structure members can be directly accessed by structure variables by any function anywhere in their scope. The structure members are *public members*.

Classes, Objects, Member Functions and Data Members

- A car typically begins as engineering drawings, like the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster.
- In a sense, the pedal “hides” the complex mechanisms that make the car go faster, just as the brake pedal “hides” the mechanisms that slow the car, the steering wheel “hides” the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly “interfaces” to the car’s complex internal mechanisms.
- Unfortunately, you cannot drive the engineering drawings of a car—before you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster. But even that’s not enough—the car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

Classes, Objects, Member Functions and Data Members

- Performing a task in a program requires a function (such as *main*). The function describes the mechanisms that perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster.
- In C++, we begin by creating a program unit called a class to house a function, just as a car's engineering drawings house the design of an accelerator pedal. A function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class's tasks.
- For example, a class that represents a bank account might contain one-member function to deposit money into the account, another to withdraw money from the account and a third to inquire what the current account balance is.

Classes, Objects, Member Functions and Data Members

- Just as you cannot drive an engineering drawing of a car, you cannot “drive” a class.
- Just as someone must build a car from its engineering drawings before you can drive the car, you must create an object of a class before you can get a program to perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language. Note also that just as many cars can be built from the same engineering drawing, many objects can be built from the same class.
- When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, make the car go faster. Similarly, you send messages to an object—each message is known as a member-function call and tells a member function of the object to perform its task. This is often called requesting a service from an object.

Classes, Objects, Member Functions and Data Members

- Thus far, we've used the car analogy to introduce classes, objects and member functions.
- In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading).
- Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated with the car.
- Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.
- Similarly, an object has attributes that are carried with the object as it's used in a program. These attributes are specified as part of the object's class.
- For example, a bank account object has a balance attribute that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class's data members.

- The class is the group of objects which can share common properties and established the relationship among them. The class is user defined data type that contains member variables and member functions.
- A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type.
- The general form of class declaration is:

```
class class_name
{
        private:
                variable declarations;
        public:
                function declarations;
};
```

- The class body contains the declaration of variables and functions. These functions and variables are collectively known as class members. The keywords private and public is known as visibility labels or access specifiers.
- The class member that has been declared as private can be accessed only from within the class while public member can be accessed from outside the class also. The data hiding is the key features of OOP.

- The variables declared inside the class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members. Example:

```
class Item
{
        private:
                int number;
                float cost;
        public:
                void setdata(int, float);
                void showdata();
};
```

- ***Item*** becomes a new data type that can be used to declare instances of the class type. The function *setdata()* can be used to assign values to the member variables number and cost and *showdata()* displays their values. These functions can only access to the private data members.

Creating Objects

- Once a class has been declared, we can create any number of variables(objects) like built-in data type variables.
- Syntax:

```
class_name object_name;
```

- For example:

```
Item x; //memory for x is created
```

- We may create more than one object in one statement as:

```
Item x, y, z;
```

Accessing Class Members

- Syntax:

```
object_name.function_name(actual arguments);
```

- For example:

```
x.setdata(100, 10.5);/*assigns number=100 and cost=10.5*/
```

- Similarly, the statement

```
x.showdata();
```

would display the value of data members.

- The statement like

```
setdata(100, 10.5);
```

has no meaning.

- Objects communicate by sending and receiving messages. This is achieved through the member functions. For example:

```
x.putdata();
```

sends a message to object ***x*** requesting it to display its contents.

Note:

- A method that does nothing more than return the value of a data field is termed an *accessor*, or sometimes a *getter*. The accessor methods should begin with the word *get*.
- Methods whose major purpose is simply to set a value are termed *mutator methods or setters*. As the name suggests, a setter most generally is named beginning with the word *set*.

Defining Member Functions

- Small methods can be defined in the class, while larger methods are defined outside.
- Member functions can be defined in two places:
    a. Outside the class definition
    b. Inside the class definition

a. Outside the class definition

General form:

```
class class_name
{
   private:
        //...
   public:
        return_type member_function(args...);
};
return_type class_name::member_function(args...)
{
   //body of function
}
```

Example

```
class Item
{
        private:
                int number;
                float cost;
        public:
                void setdata(int,float);
                void showdata();
};
```

```
void Item::setdata(int id, float price)
{
        number = id;
        cost = price;
}
void Item::showdata()
{
        cout<<number<<endl
                <<cost<<endl;
}
```

b. Inside the class definition

General form:

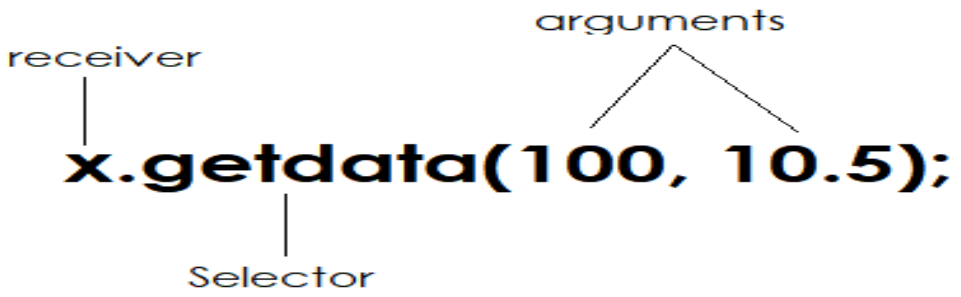
```
class class_name
{
   private:
        //...
   public:
        return_type member_function(args...)
        {
             //body of function.
        }
};
```

Example

```
class Item
{
        private:
                int number;
                float cost;
        public:
                void setdata(int id, float price)
                {
                        number = id;
                        cost = price;
                }
                void showdata()
                {
                        cout<<number<<endl
                                <<cost<<endl;
                }
};      //program1
```

Message Passing

- The term message passing (sometimes also called method lookup) to mean the dynamic process of asking an object to perform a specific action.
- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed but may differ depending upon the class of the receiver. That is, different objects may accept the same message, and yet perform different actions.
- There are three identifiable parts to any message-passing expression. These are the *receiver* (the object to which the message is being sent), the message *selector* (the text that indicates the particular message being sent), and the *arguments* used in responding to the message. Message passing uses a period to separate the receiver from the message selector.



Array as data member

//program2

Object as Function Arguments

- Like other variables, any number of objects can be passed to the function as arguments.
- While passing object, all its data members are passed as a single unit like structure.

//program3

Returning Objects from Functions

- Like other data types objects can also be returned from function.

//program4

Home Assignment

1. Create a class ***distance***; a *setdist(int ft, float in)* is used to take feet and inches as arguments and initialize its member variables inches and feet. Another function *getdist()* is used to inches and feet from user then *add* function having two class objects as parameter which add these values, respectively. (***12' = 1 feet***)

//program5

2. Add two times (returning and without returning objects)

3. Add two dates (returning and without returning objects)

Reference parameters

- An alias for its corresponding argument in a function call & placed after the parameter type in the function prototype and function header
- Example

  `int &count` in a function header
- Pronounced as “count is a reference to an int”
- Parameter name in the body of the called function actually refers to the original variable in the calling function

//program12

//program13

Friend Function

- A non-member function cannot have 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.
- For example, consider a case where two classes, programmer and system analyst, have been defined. We would like to use a function income_tax() to operate on the object of both these classes. In such situation, C++ allows a 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 member of any these classes.
- To make outside function friendly, we must declare this function as a friend of the class as shown as:

```
class class_name
{
        private:
                //...
        public:
                //friend function declaration
                friend return_type function_name(arguments);
                //...
};
```

- The function definition does not use either the keyword friend or the scope resolution operator (::).

Characteristics of friend functions

- It is not in the scope of the class to which it has been declared as friend function. Thus it cannot be called using the object of the class. It can be invoked like a normal function without the help of any object.
- It cannot access the data member directly and has to use an object name with each data member. (for e.g. T.hour)
- It can be declared either in the public or private section of a class.
- Usually, friend function has objects as parameters.

//program6

//program7

//program8

Static Data

- The modifier static means that there exists only one instance of a variable, regardless of how many instances of the class are created.
- Memory is allocated for all the members of the objects. Storage space for the data members which are declared as static is not allocated when the object is created.
- For different objects of the same class, automatic data members of each object are stored in separate memory location, whereas static data members of all objects are stored in the same memory location.
- Static data members do not belong to a specific object but to a class in general. So static data members are also called 'class variables'.
- The static data members are declared in the class and is defined outside the class. To make a data member static, the keyword static is used.
- General Form:

```
class class_name
{
        private:
                //...
                static data_type data_member;
        public:
                //...
};
data_type class_name::data_member = initial value;
```

- The static variable exists throughout the program's lifetime but are limited to the scope of the class. A static data item is useful when all objects of same class must share common information.

Static Function

- Static function belongs to a class not to any object like static data. However, the objects can access static functions. Static function cannot access non static member data since non static members belong to object and they can be only accessed through objects.
- Static functions can access only static data. But non-static member function can access non-static as well as static data.

//program9

Inline Function

- When the compiler sees function call there must be
  - An instruction for jump to the function
  - instructions for saving registers
  - instruction for pushing arguments onto the stack in the calling program and removing them from the stack in the function (if there are arguments),
  - instruction for restoring registers
  - and an instruction to return to the calling program. The return value if any must be dealt with.
- All the instructions slow down the program. To save execution time in short function, put the code in the function body directly inline with the code in the calling program.
- That is, each time there is function call in the source file, the actual code from the function is inserted, instead of a jump to the function.
- Functions that are very short, say one or two statements, are candidates to be inline function.
- The inline keyword is just a request to the compiler. Sometimes the compiler will ignore the request and compile the function as a normal function.
- The inline function is declared as:

```
inline return_type function_name(args)
{
        //function body
}                    //program10
```

Default Argument

- When a function is called, the number of actual argument and formal argument must be same.
- In C++, there is a provision of supplying a smaller number of arguments than the actual number of parameters. This mechanism is supported by default argument.
- If we do not supply any argument, the default value is used for the absent argument.
- The default values are specified when function is declared.
- For example,

```
float total(float m1 = 40, float m2 = 40, float m3 = 40)
{
    return (m1 + m2 + m3);
}
```

- Consider the following function calls:

```
total();              //40 + 40 + 40 = 120
total(50);            //50 + 40 + 40 = 130
total(80, 60);         //80 + 60 + 40 = 180
total(35, 45, 55);    //35 + 45 + 55 = 135
```

- All the default arguments must be in the right part of the parameter list.

```
void total(float m1, float m2, float m3 = 40);      //valid
void total(float m1 = 40, float m2, float m3);      //invalid
void total(float m1, float m2 = 40, float m3);      //invalid
```

//program11

Constructor

- It is a special type of member function. Its name is same as class name.
- It does not have return type but may have arguments.
- Its task is to initialize the object of its class. It is called automatically when object is created.
- It is used to initialize the value of data member when object is created. Also, it allocates memory for the object.
- Moreover, constructor cannot be inherited and cannot be virtual.
- There are three types of constructors: *default, parameterized* and *copy*.

1. Default Constructors

- The constructor having no parameter is called default constructor. It is called by creating object without parameter.

```
class class_name
{
        private:
                //...
        public:
                class_name()
                {
                        //...
                }
};
```

//program14

2. Parameterized Constructor

- It should have parameter. It is called when object is created with parameter.

```
class class_name
{
    private:
        //...
    public:
        class_name(args)
        {
            //...
        }
};
```

//program15

3. Copy Constructor

A kind of parameterized constructor which accepts another object as parameter. It copies values of an object to another object.

```
class class_name
{
        private:
                //...
        public:

class_name(&obj)
                {
                        //...
                }
};
```

```
class integer
{
        private:
                int m, n;
        public:
                integer(integer &i)
                {
                        m = i.m;
                        n = i.n;
                }
};
```

- `integer i3(i2);`

would automatically invoke the third constructor which copies the values of i2 to i3.

Multiple Constructors/Constructor Overloading

- C++ allows us to use both default and parameterized constructors in the same class.

//program16

Destructor

- Like constructor, its name is same as class name but preceded by *tilde (~).* Moreover, it does not have any parameter and return type.
- It performs the reverse operations performed by constructor
- It is called automatically when object is going to be destroyed.
- It releases the memory assigned by the object.
- There is no meaning of destructor overloading.

```
        class class_name
        {
                private:
                        //...
                public:
                        calss_name(){}//constructor
                        //....
                        ~class_name();//destructor
        };
//program17
```

Constructor vs Destructor

	Constructor	Destructor
Purpose	Constructor is used to **initialize the instance of a class**.	Destructor **destroys the objects** when they are no longer needed.
When Called	Constructor is Called when new instance of a class is created.	Destructor is called when instance of a class is deleted or released.
Memory Management	Constructor **allocates the memory**.	Destructor **releases the memory**.
Arguments	Constructors can have arguments.	Destructor can not have any **arguments**.
Overloading	**Overloading** of constructor is possible.	Overloading of Destructor is not possible.
Name	Constructor has the same name as class name.	Destructor also has the same name as class name but with tilde (~) **operator**.

Syntex	ClassName(Arguments)
	{
	//Body of Constructor
	}
		~ ClassName()
		{
		}

C++ DYNAMIC MEMORY

- Memory in your C++ program is divided into two parts:
  - **The stack**: All variables declared inside the function will take up memory from the stack.
  - **The heap**: This is unused memory of the program and can be used to allocate the memory dynamically when program runs.
- Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.
- You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.
- If you are not in need of dynamically allocated memory anymore, you can use delete operator, which de-allocates memory previously allocated by new operator.

The new and delete Operators

- There is following generic syntax to use new operator to allocate memory dynamically for any data-type.

```
new data-type;
```

- Here, data-type could be any built-in data type including an array or any user defined data types include class or structure.
- For example we can define a pointer to type **double** and then request that the memory be allocated at execution time. We can do this using the new operator with the following statements:

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double; // Request memory for the variable
```

- The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue = NULL;
if( !(pvalue = new double ))
{
    cout << "Error: out of memory." <<endl;
    exit(1);
}
```

The new and delete Operators cont.

- At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the **delete** operator as follows:

```
delete pvalue; // Release memory pointed to by pvalue
```

//program18

Dynamic Memory Allocation for Arrays

- Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue = NULL; // Pointer initialized with null
pvalue = new char[20]; // Request memory for the variable
```

- To remove the array that we have just created the statement would look like this:

```
delete[] pvalue; // Delete array pointed to by pvalue
```

- Following the similar generic syntax of new operator, you can allocate for a multidimensional array as follows:

```
double** pvalue = NULL; // Pointer initialized with null
pvalue = new double [3][4]; // Allocate memory for a 3x4 array
```

- However, the syntax to release the memory for multi-dimensional array will still remain same as above:

```
delete[] pvalue; // Delete array pointed to by pvalue
```

Dynamic Memory Allocation for Objects

//program19

Memory Recovery

- Memory created using the **new** operator is known as heap-based memory, or simply heap memory. Memory is always a finite commodity, and hence some mechanism must be provided to recover memory values.
- Memory that has been allocated to object values is then recycled and used to satisfy subsequent memory requests.
- There are two general approaches to the task of memory recovery.
- Some languages (such as C++ and Delphi Pascal) insist the programmer indicate when an object value is no longer being used by a program, and hence can be recovered and recycled. The keywords used for this purpose vary from one language to another.
- In C++, the keyword is ***delete*.**

```
delete pointer_name;
```

- When an array is deleted a pair of square braces must be placed after the keyword:

```
delete [] pointer_name;
```

- The alternative to having the programmer explicitly manage memory is an idea termed garbage collection. A language that uses garbage collection (such as Java, C#, Smalltalk) monitors the manipulation of object values, and will automatically recover memory from objects that are no longer being used.

Memory Recovery cont.

- Generally garbage collection systems wait until memory is nearly exhausted, then will suspend execution of the running program while they recover the unused space, before finally resuming execution.
- Garbage collection uses a certain amount of execution time, which may make it more costly than the alternative of insisting that programmers free their own memory. Automatic garbage collection can be expensive, as it necessitates a run-time system to manage memory.
- But garbage collection prevents several common programming errors:
- It is not possible for a program to run out of memory because the programmer forgot to free up unused memory.
- It is not possible for a programmer to try to use memory after it has been freed. Freed memory can be reused, and hence the contents of the memory values may be overwritten. Using a value after it has been freed can therefore cause unpredictable results.

```
PlayingCard *aCard = new PlayingCard;
...
delete aCard;
...
cout << aCard.rank(); // attempt to use after deletion
```

Memory Recovery cont.

- It is not possible for a programmer to try and free the same memory value more than once. Doing this can also cause unpredictable results.

```
Playingcard * aCard = new PlayingCard(Space, 1);
...
delete aCard;
...
delete aCard; // deleting already deleted value
```

- When a garbage collection system is not available, to avoid these problems it is often necessary to ensure that every dynamically allocated memory object has a designated owner. The owner of the memory is responsible for ensuring that the memory location is used properly and is freed when it is no longer required.
- When a single object cannot be designated as the owner of a shared resource, another common technique is to use reference counts. A reference count is a count of the number of pointers that reference the shared object. Care is needed to ensure that the count is accurate; whenever a new pointer is added the count is incremented, and whenever a pointer is removed the count is decremented. When the count reaches zero it indicates that no pointers refer to the object, and its memory can be recovered.