State identity and behavior of an object

What is an Object?

If you look around you, you will find many examples of real-world objects: your book, your computer, your pet, etc. In object-oriented programming, programs are viewed as collections of objects. Objects have three characteristics: state, behavior and identity. We can use a bank account to illustrate these concepts:

1. State: id, name, balance
2. Behaviour: deposit, withdraw, etc.
3. Identity: Joe's account is similar to Jane's, but its state has different values.

State of an object:

The state of an object consists of a set of data fields (its properties) with their current values. The state represents the cumulative results of an object's behavior.

Behavior of an object:

Behavior is how an object acts and reacts in terms of its state changes and message passing. When you send a message to an object, you actually invoke a method (i.e. execute some code). Invoking a method will cause certain well-defined behavior, and may change the object's state. The behavior may depend on the object's current state

A few kinds of operations that a client may perform on an object:

1. **Modifier:** alters the state of an object. E.g. karan.setNetWorth(25000);
2. **Selector:** accesses the state of an object, but does not alter it. E.g. age = s.getAge();
3. **Constructor:** creates an object and initializes its state.
4. **Destructor:** destroys an object (frees its memory).

Identity of an object:

Identity is that property of an object which distinguishes it from all others. Every instance of a class has its own memory to hold its state.

Reference Variables

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable, A reference variable is created as follows:

data-type & reference-name = variable-name
Example;
float total = 100;
float & sum = total;

total is a float type variable that has already been declared; sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

cout<< total;
and

cout << sum;
both print the value 100.
The statement
total = total + 10;

will change the value of both total and sum to 110.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here, & is not an address operator, The notation float & means reference to float. Other example are:

```
int n[10];
int & x=n[10]; // x is alias for n[10]
char & a= '\n'; //initialize reference to a literal
```

The variable x is an alternative to the array element n[10]. The Variable a is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant \n is stored.

A major application of reference variables is in passing arguments to functions. Consider the following'.

```
void f( int &x ) // uses reference
{
        x=x+10;         //x is incremented: so also m
}

int main( )
{
        int m=10;
        f(m);           //function call.
i
```

l
When the function call f(m) is executed, the following initialization occurs:

```
int & x = m;
```

Thus x becomes an alias of m after executing the statement f (m). Such function calls are known as call by reference. Since the variables x and m are aliases, when the function increments x, m is also incremented, the value of m becomes 20 after the function is executed.

The call by reference mechanism is useful in object oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth, It is also important to note that references can be created not only for built in data types but also for user-defined data types such as structures and classes.

Scope resolution operator

Like C, C++- is also a block structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration, A variable declared inside a block is said to be local to that block. Consider the following statement of the program:

```
…..
…..
{

        int x=10;
        …..
        …..
}
        ……
        ……
{
        int x=1;
        …….
        …….
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing new operator :: called the Scope resolution operator. This can be used to uncover a hidden variable. It takes the following form:

Functions

Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

When the Function in called, controlled is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered.

The Main Function

C does not specify any return type for the main() function which is the starting point for the execution of a program. The definition of main() would look like this:

```
main()
{
        // main program statements
}
```

This is perfectly valid because the main() in C does not return any value. In C++ the main() returns a value of type int to the operating system.

Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times, However, every time a function in called, it takes a lot of extra time in executing a 5eries of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function to small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macro. The major drawback with macros is that they are not really functions and therefore the usual error checking does not occur during compilation.

To eliminate the cost of calls to small functions, C + + proposes a new feature called inline function. An inline function is a function that is expanded inline when it is invoked- That is, the compiler replaces the function call with the corresponding code. It is easy to make a function inline. All we need to do is to prefix the keyword inline to the function definition. All inline functions must be defined before they are called. Usually, the functions are made inline when they are small enough to be defined in one or two lines, Example:

```
Inline double cube (double a)
{
        return (a* a *a) ;
}
```

Some of the situations where inline expansion may not work arc:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

Example:

```
#include<iostream.h>
Inline float mul( float x, float y)
{
        return(x*y);
```

```
}
Inline double div( double p, double q)
{
        return(p/q);
}
int main()
{
        float a= 12.34;
        float b= 9.82
        cout<<mul(a,b)<<”\n”;
        cout<<div(a,b)<<”\n”;

        return 0;
}
```

C Structure

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling group of logically related data items. It is a user-defined datatype with a template that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations, For example, consider the following declaration:

```
struct student
{
char name [20];
int ro11_number;
float total_marks;
}
```

The keyword struct declares student as a new data type that can hold three fields of different data types, These fields are known as structure members orelemente. The identifier student, which is referred to structure name or structure tag, can he used to create variables of type student.

Example;

```
struct student A; // c declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the dot or period operator as follows:

```
strcpy(A.name, "John");
A.ro1l_number = 999;
A.total_marks = 595.5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C Structure
The standard C does not allow the struct data type to be treated like built-in types.

```
struct complex
{
float x;
float y:
}
struct complex c1,c2, c3;
```

The complex numbers c1,c2 and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 =cl + c2;
```

is illegal in C.

Extensions to Structures
C++ supports all the features of structures as defined in C. But C++ has expanded the capabilities further to suit its OOp philosophy. it attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. Inheritance, a mechanism by which one type can inherit characteristics from other types, is also supported by C++. In C++, a structure tan have hath variables and functions as members. It can also declare some of its member as private so that they cannot be accessed directly by external functions.

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, 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, (generally, a class specification has two parts;

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.
The general form of a class declaration is:

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

};

The class declaration is similar to a struct declaration, The keyword class specifies, that what follows is an abstract data of type cla8ft_name. The body of a class is enclosed within braces and terminated by a semicolon, The class body contains the declaration of variables and functions. These functions and variables are collectively called class members. They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords private and public are known as visibility labels.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration] is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are private. If both the labels are missing, then by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared, inside the class are known as data members and the functions are known as member function. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data)can be accessed from outside the class.The binding of data, and functions together into a single class type variable is referred to as encapsulation.

A Simple Class Example

A typical class declaration would look like:

```
class Item
{
        int number;
        float cost;
public:
        void getdata(int a,float b);
        void putdatafvoid);
};
```

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare instances of that class type, The class item contains two data members and two function members. The data members are private by default while both, the function are public by declaration. The function getdata() can be used to assign values to the member variables number and cost, and putdata() is for displaying their values. These functions provide the only access to the data members from outside the class This means that the data cannot be accessed by any function that is not a member of the class Item.

Creating Objects

Remember that the declaration of item as shown above does not define any object of item but only specifies what they will contain. Once a class has been declared, we can create variables of that type by using the class name.

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

creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item, We may also declare more than one object in one statement, Example:

Item x,y;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory apace is allocated to aft object at this stage.

Accessing Class Members

As pointed not earlier, the private data of a class can be accessed only through the member functions of that class. The main() cannot contain statements that access number and cost directly. The following is the format for calling a member function:

object-name.function- name (actual -arguments);

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
        int x;
        int y,
public:

        int z;
};
Void main()
{

        Xyz p;
        p.x=o           // error, x is private
        p.z=10          // OK, it is public
}
```

Defining Member Functions

Member functions can be defined in two places:
Outside the class definition.
Inside the class definition.
It is obvious thatH irrespective of the place of definition, the function should perform the same task.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body.

An important difference between 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. The general form of a member function definition is:

```
return- type class_name :: function_name (argument declaration)
{
        Function body
}
```

The membership label class_name :: tells the compiler that the function function- name belongs to the class class_name. That is in the scope of the function is restricted to the class_name specified in the header line. The symbol :: is called the scope resolution operator.

For instance, consider the member functions getdata() and putdata() as discussed above. They may be coded as follows:

```
Void  item :: getdata(int a, float b)
{
        number = a;
        cost = b;
}

void item::putdata(void)
{
        cout<<"Number is:"<<number<<"\n";
        cout<<"Cost :" <<cast<<"\n";
}
```

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

1. Several different classes can use the same Function name. The membership label will resolve their scope.
2. Member Junctions can access the private data of the class. A non- member function Cannot do so.{However, an exception to this rule- ls a friend function).
3. A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class . For example, we could define the item class as follows:

```
class item
{
        int number;
        float cost;
public:
```

```
        void getdata(int a, float b); // declaration If inline function
        void putdata(void) // definition inside the class
        {
                cout <<numher << "\n";
                cout <<cost <<"\n";
        }
};
```

When a function is defined inside the class, it is treated as inline function. Therefore, the restrictions and limitations that apply to an inline function are also applicable here.

Making an Outside Function Inline

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition, Example:

```
class item
{
        Public:
                void getdata(int a, float b); // declaration
}
 Inline void item :: getdata(int a, float b) // definition
{
        number = a;
        cost = b
}
```

Nesting of Member Functions

A member function of a class can be called only by an abject of that class using a dot operator. However, there is an exception to this, A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions

```
#include <iostream.h>

class set
{
        int m, n;
puhlic:
        void input(void);
        void display (void);
        int largest (void);
};

int set:: largest(void)
{
if(m >n)
        return (m);
```

```
else
        return (n);
}
void set:: input (void)
{
        cout <<"Input values of m and n <<"\n":
        cin>> m>>n;
}
void set :: display(void)
{
        cout<<"Largest value =”<<largest ();
}
int main( )
{
set A;
a.input( );
A.display( );
return 0;
}
```

Function Overloading

overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

We can design a family of functions with one function name but with different argument lists. The 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. For example, an overloaded add() function handles different types of data as shown below;

```
// Declarations
int add(ini a. int b);                  // prototype 1
int add(int a. int b. int c)            // prototype Z
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);                  // uses prototype 3-
cout « add(5,10. 15);                    // uses prototype 1
cout « add(0.5,5);                      //uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution, A best match must be unique, The function selection involves 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
   flort to double
   to find a match.

3. When either of them fails, the compiler tries to use the built-in. conversions (the implicit assignment conversions) 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 generate an error message. Suppose we use the following two functions:
   long square (long n)
   double square (double x)
   A function call such as
   square (10)
   will cause an error because int argument can be converted to either longer double, hereby creating an ambiguous situation as to which version of square should be used.

4. If all of the 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, User-defined conversions are often used in handling class objects.

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks.

```
// Function volume() is overloaded three times
# Include <iostream.h>

int voluire(int);
double volume(double, int);
long volune(long, int, int);
int main ()
{
        cout<< volume(10)<< "\n";
        cout<<volume(2.5,8)<<"\n";
        cout<<volume(100L,75,15)<<"\n";
        return 0;
}
//Function definitions
int volume (int a) // cube
```

```
{
        return (s*s*s)
}

double volume(double r, int h) // cylinder
{
        return(3.14*r*r*h);
}
        long volume{long l, int b, int h} // rectangular box
        return (l*b*h);
}
```