2 Concepts of Object Oriented Programming

Significant Learning Outcomes

After the completion of this chapter, the learner

- compares various programming paradigms.
- lists the features of procedure-oriented paradigm.
- lists the advantages of object-oriented paradigm.
- explains the concepts of data abstraction and encapsulation, citing examples.
- explains inheritance and polymorphism with the help of real life examples.

We learn a programming language to develop programs which make the computer a more useful machine. The bigger the program is, the more difficult it becomes to manage it. To overcome this difficulty there are a lot of tools like IDE, debugger, compiler, etc. that we can use, and approaches like structured, procedural, modular, object oriented, etc. that we can follow in software development. Implementation of these tools and approaches helps us to address a number of issues faced in software development, like maintainability, reusability, portability, security, integrity, and the user-friendliness of software products.

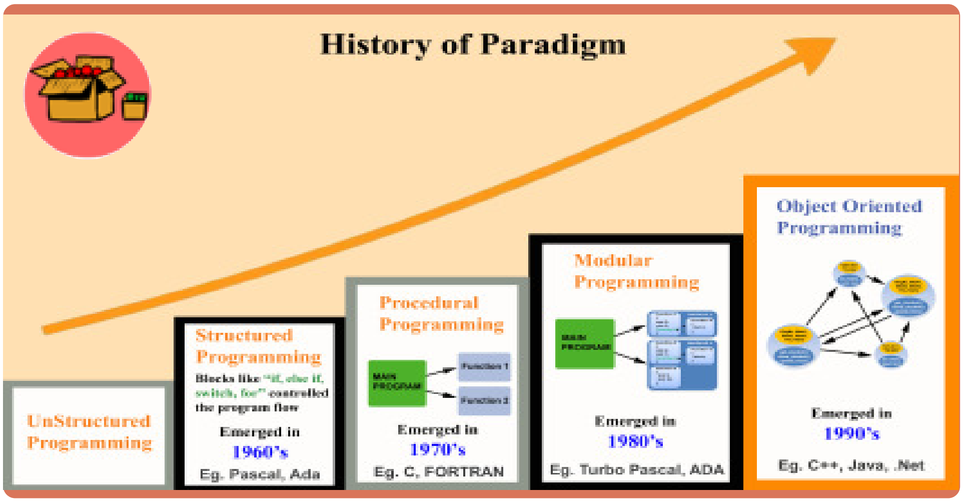
In the previous chapter, we learnt various programming concepts and developed programs in C++ language for solving problems. Our aim has been to process the inputs and produce the output. As the programs developed were small and less complex, we never thought of giving importance to the approaches we adopted or to the security of data used or in organising the processing steps. But, knowingly or unknowingly, we were following some approach while writing programs to solve problems. This chapter discusses the programming approach that we have been following so far and presents new approaches, so that we can choose a suitable approach for program development.

2.1 Programming paradigm

A programming paradigm denotes the way in which a program is organised. If the program is very small, there is no need to follow any organising principle. But, as it becomes larger, some measures have to be taken for managing it and for reducing its complexity.

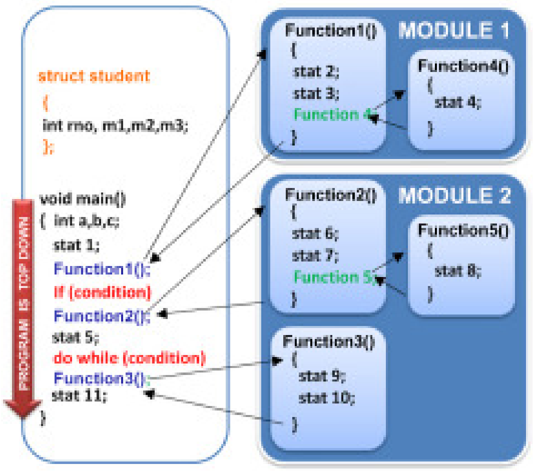
While some paradigms give more importance to procedures, others give more importance to data. Capabilities and styles of various programming languages are defined by the programming paradigms supported by them. The various approaches that have been tried are modular programming, top-down programming, bottom-up programming and structured programming. Each one of these approaches were used to reduce the complexity of programming and to create reliable and maintainable programs.

Some programming languages are designed to follow only one paradigm, while others support multiple paradigms. C++ is a multiple paradigm language. Using C++, we can implement two of the most important programming paradigms, the procedural paradigm and the object-oriented paradigm. Let us discuss each of these in detail.



2.1.1 Procedure-Oriented Programming paradigm

Procedure-Oriented Programming specifies a series of well-structured steps and procedures to compose a program. It contains a systematic order of statements, functions and commands to complete a computational task or program. The statements may be to accept input, to do arithmetic or logical operations, to display result, etc. In this approach emphasis is on doing things. In this paradigm, when the program becomes larger and complex, the list of instructions is divided and grouped into **functions**. A function clearly defines the purpose, and interface to other functions in the program, thus reducing the complexity. To further reduce the complexity, the functions associated with a common task are grouped into **modules**. Figure 2.1 shows a procedural paradigm in which a large program is divided into five separate functions, and functions associated with a common task grouped into two modules.



```
struct student
{
int rno, m1,m2,m3;
};

void main()
{  int a,b,c;
   stat 1;
   Function1();
   If (condition)
   Function2();
   stat 5;
   do while (condition)
   Function3();
   stat 11;
}
```

```
Function1()
{
 stat 2;
 stat 3;
 Function 4;
}
```

```
Function4()
{
  stat 4;
}
```

```
Function2()
{
 stat 6;
 stat 7;
 Function 5;
}
```

```
Function5()
{
  stat 8;
}
```

```
Function3()
{
  stat 9;
  stat 10;
}
```

Fig. 2.1: Procedural paradigm

C, Pascal, FORTRAN, BASIC, etc. are procedural languages. In the C++ learning process, the organising principle we followed till now was the procedural paradigm. Procedural programming languages are also known as top-down languages.

But, when the program gets larger and more complex, the procedure-oriented programming approach is found to have several limitations. No matter how well this approach is implemented, large programs become excessively complex. The main reasons for the increasing complexity with procedural languages are:

a. Data is undervalued.

b. Adding new data element may require modifications to all/many functions.

c. Creating new data types is difficult.

d. Provides poor real world modelling.

Let us discuss each of them in detail.

a. Data is undervalued

In procedural language, emphasis is on doing things. Here data is given less importance. Let us explain this with the help of an example. Assume that we have to develop a software for automating the activities at our school. The activities may include admitting a new student, removing a student, recording fee collection details etc. Imagine we implemented the activities into the software using a function for each activity. The data that may be required by most of these functions for their functioning may be the student details stored in an array of structures. The easy way of making this data available to all these functions is to declare the student array as global (see section 10.5 of your Class XI textbook). Now the data is exposed and any function other than the function that requires this data may also access it and make changes to the data knowingly or unknowingly as we cannot imply any restrictions.

This is like leaving your assignment to be submitted the next day, over the dining table. There are chances of this document getting destroyed, as a small child may tear it or draw pictures on it or a cup of tea placed on the table may spill over it accidentally. This can happen as the assignment is kept exposed in a place where anybody can access it.

The arrangement of local variables, global variables and functions in a procedural programming paradigm is shown in Figure 2.2. The green lines show which functions require a particular data and red lines show illegal access of data by functions that do not require it.

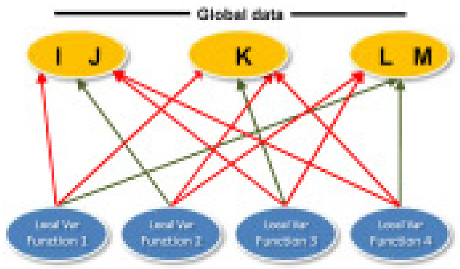


Fig. 2.2: Functions accessing data

b. Adding new data element may require modifications to all/many functions

Since many functions access global data, the way the data is stored is important. The arrangement of data cannot be changed without modifying all the functions that access it. If we add new data items, we will need to modify all the functions that access the data, so that they can also access the new items. It will be hard to find all such functions, and even harder to modify all of them correctly.

For example, assume that we need to add an item named '`age`' to the existing student structure in our school software. For the proper functioning of the software, we will have to find all functions that access the student data and incorporate proper changes.

c. Creating new data types is difficult

Computer languages typically have several built-in data types like integer, float, character and double. Certain programming language allows creation of data types other than the built-in data type, which means they are extensible. The ability of a program to allow significant extension of its capabilities, without major rewriting of code or changes in its basic architecture is called **extensibility.** This feature helps us in reducing the complexity of a program while extending its capabilities. Procedural languages are not extensible.

d. Provides poor real world modelling

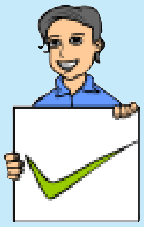
In procedural programming paradigm, functions and data are not considered as a single unit and are independent of each other. So neither data nor functions in procedure oriented paradigm, by themselves, cannot model real-world objects effectively.

For example, in our school software, we have student data and functions that access it. Similarly, assume that the software also maintains teacher data and functions accessing this data. In procedural programming we may not be able to club student data and the functions accessing it or teacher data with the functions accessing it.

Every real-world object that we deal with, have both characteristics and behaviour bundled in a single unit. If we take the object 'humans' as an example, its characteristics can be name, gender, nationality, etc. and behaviour can be speaking, laughing, etc. Even though behaviour may be implemented as a function and characteristics may be represented as data in a program, we are not able to club data and functions. Therefore modelling things in the real world is difficult in procedural programming.

In the next section, we will discuss Object-Oriented Paradigm and see how it tries to eliminate the limitations of Procedure Oriented paradigm.

Know your progress



1. State whether the following three statements are true or false:
   a. Global variables cannot be accessed in more than one function.
   b. Procedural programming resembles closeness to real world.
   c. In procedural programming paradigm, data and functions are not bound together.
2. Pick the procedural languages from the following.
   C, C++, Fortran, JAVA, Pascal.

2.1.2 Object-Oriented Programming (OOP) paradigm

Object-oriented paradigm eliminates the problems in the procedural paradigm by clubbing data and functions that operate on that data into a single unit. In OOP, such a unit is called an **object**.

For example, by implementing OOP in our school software, we can create a `Student` object by clubbing student data and its functions as well as `Teacher` object by clubbing teacher data and its functions. Now the functions of one object will not be able to access the data of other object without permission.

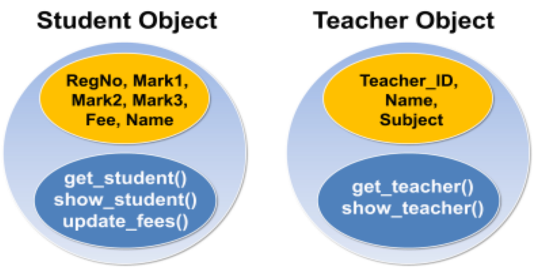


Fig. 2.3: Objects containing data and functions

Advantages of using OOP are:

a. OOP provides a clear modular structure for programs.

b. It is good for defining abstract data types.

c. Implementation details are hidden from other modules and have a clearly defined interface.

d. It is easy to maintain and modify the existing code as new objects can be created without disturbing the existing ones.

e. It can be used to implement real life scenarios.

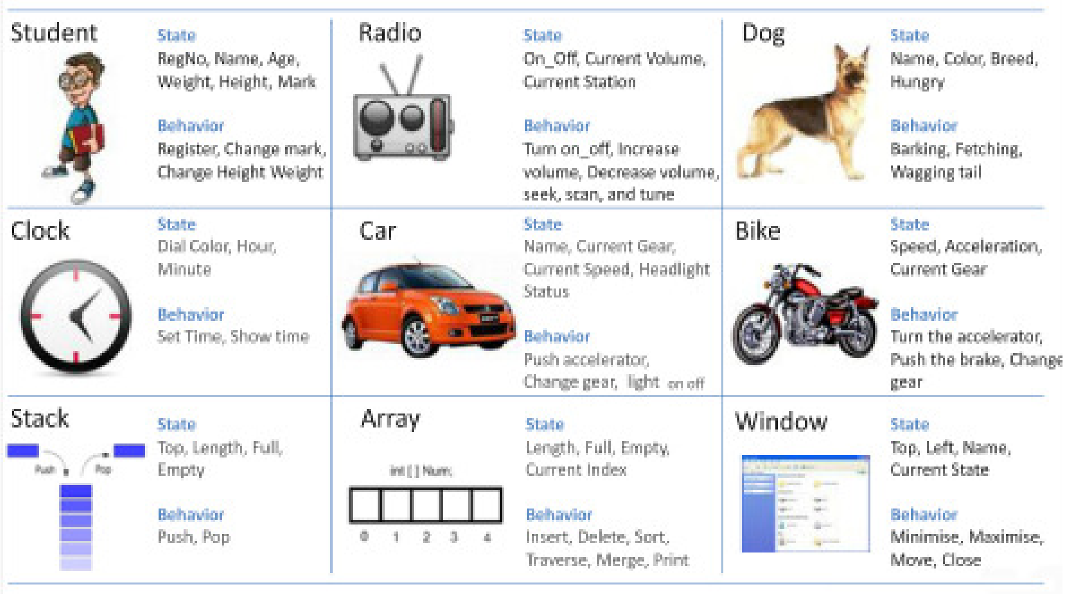
f. It can define new data types as well as new operations for operators.

2.2 Basic concepts of OOP

Object Oriented Programming simplifies the software development and maintenance by providing some concepts such as objects, classes, data abstraction, data encapsulation, modularity, inheritance, polymorphism. Let us discuss these concepts in detail.

2.2.1 Objects

Anything that we see around us can be treated as an object and all these objects have properties (also called member/data/state) and behaviour (also called methods/member functions). Some examples of objects are listed in Figure 2.4 with their properties and methods.



	State	Behavior
Student	RegNo, Name, Age, Weight, Height, Mark	Register, Change mark, Change Height Weight
Radio	On_Off, Current Volume, Current Station	Turn on_off, Increase volume, Decrease volume, seek, scan, and tune
Dog	Name, Color, Breed, Hungry	Barking, Fetching, Wagging tail
Clock	Dial Color, Hour, Minute	Set Time, Show time
Car	Name, Current Gear, Current Speed, Headlight Status	Push accelerator, Change gear, light on off
Bike	Speed, Acceleration, Current Gear	Turn the accelerator, Push the brake, Change gear
Stack	Top, Length, Full, Empty	Push, Pop
Array	Length, Full, Empty, Current Index	Insert, Delete, Sort, Traverse, Merge, Print
Window	Top, Left, Name, Current State	Minimise, Maximise, Move, Close

Fig. 2.4: Real world objects with their properties (state) and methods (behavior)

Let us do

Observe some real world objects around you, identify properties each of them posseses and the behaviours eacht exibits. Write your findings in the following table:

Object Name	Properties	Behaviour

When OOP is to be implemented to solve a programming problem, instead of dividing the problem into functions we will have to think about dividing it into objects. When thinking in terms of objects rather than functions, program designing becomes easier, as there is a close match between objects in the program and objects in the real world.

In OOP an **object** is obtained by combining data and functions acting upon the data into a single unit. After combining, the functions inside an object are called **member functions** and data is called **member** (see Figure 2.6).

2.2.2 Classes

An object is defined via its class which determines everything about an object. A **class** is a prototype/blue print that defines the specification common to all objects of a particular type. This specification contains the details of the data and functions that act upon the data. Objects of the class are called individual **instances** of the class and any number of objects can be created based on a class (see Figure 2.5).

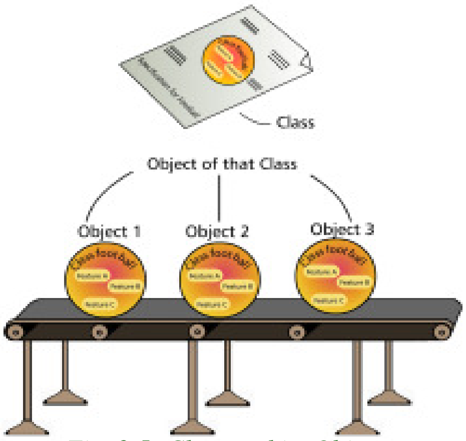
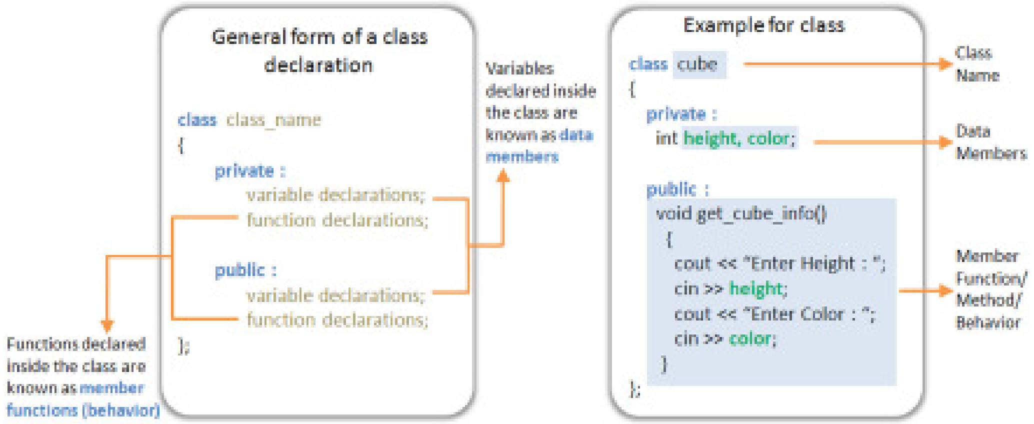


Fig. 2.5: Class and its Objects



General form of a class declaration

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

    public :
        variable declarations;
        function declarations;
};
```

Example for class

```
class cube
{
  private :
   int height, color;

  public :
   void get_cube_info()
    {
     cout << "Enter Height : ";
     cin >> height;
     cout << "Enter Color : ";
     cin >> color;
    }
};
```

Variables declared inside the class are known as data members
Functions declared inside the class are known as member functions (behavior)
Class Name
Data Members
Member Function/ Method/ Behavior

Fig. 2.6: General form of declaring a class with example

The declaration and usage of class is almost similar to that of a structure. A structure includes specifications regarding data, where as a class includes specification regarding both data and functions that use the data (See Figure 2.6). A structure is declared using the keyword '`struct`', where as class is declared using the keyword '`class`'.

If `Student` is the name of a C++ class, to create two objects named '`S1`' and '`S2`', (as in Figure 2.7) declaration will be as follows:

```
Student S1,S2;
```

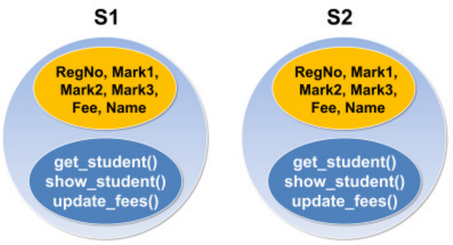


Fig. 2.7: Objects of Student class

Objects can be declared and created from a class using the statement.

```
Cube S;    Or    Cube *C;
                 C = new Cube;
```

where `Cube` is the name of the class and `C` is the name of the object.

Objects can communicate with each other by passing message, which is similar to people passing message with each other. This helps in building systems that simulate real life. In OOP, calling member function of an object from another object is called **passing message**. Message passing involves specifying the name of object, the name of the member function, and the information to be sent.

For example, in our school software, the `Teacher` object to update the fees of a student, will pass a message to `Student` object by calling the `update_fee()` member function (See Figure 2.8) like

```
S1.update_fees("Rahul", 1000);
```

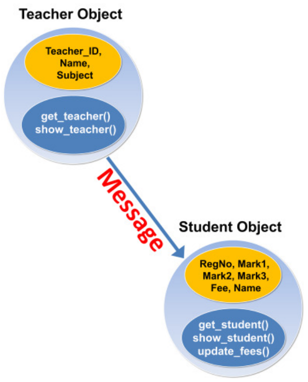


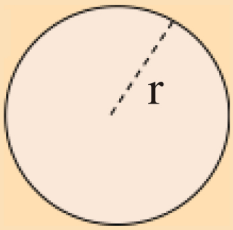
Fig. 2.8: Objects passing message

where `S1` is an object of `Student` object. The `Student` object on reciving the message will update the fees of the student with the data provided by the `Teacher` object.

Know your progress

1. OOP stands for _________.
2. A blueprint for an object in OOP is called a __________.
3. The functions associated with the class are called __________.
4. The variables declared inside the class are known as __________.
5. What is the difference between structure and class?

The following program implements a circle as an object. The class 'Circle' declares the only data required - the radius and the member functions for accepting the radius and displaying the area.



```
#include<iostream>
using namespace std;
class Circle
 {
   private:
      float r;   } Member
   public:
   void get_radius()
   {
     cout << "Enter Radius :";
     cin >> r;
   }
   void display_area()
   {
    cout<< "Area:"<< 3.14*r*r;
   }
   };
   int main()
   {
   Circle C1;
   C1.get_radius(); //Sending message
   C1.display_area(); //Sending message
   }
```

Member functions

Class Declaration

Class

Object

Main program

Output :

```
 Enter Radius : 2.0
 Area: 12.56
```

2.2.3 Data Abstraction

Data abstraction refers to showing only the essential features of the application and hiding the details from outside world.

Let us take a real life example of a television which we can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, but we do not know about its internal details, that is, we do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and we can use its interfaces like the power button, channel selector, and volume control without having any knowledge about its internal features.

Like this, C++ classes provide great level of data abstraction. They provide public methods/functions to the outside world to use the functionality of an object and to manipulate object data. These methods helps to manipulate objects from outside without actually knowing how object has been implemented internally.

Data abstraction separates interface and implementation. **Implementation** denotes how variables are declared, how functions are coded, etc. It is through **interface** (function header/function signature) a communication is send to the object as messages.

For example, in our school software, a member function e.g. `show_student()` of `Student` object may be called without actually knowing what algorithm the function uses internally to display the given values. At any time we can change the implementation of `show_student()`, and the function call will still work as long as long there is no change in the interface.

Data abstraction provides two important advantages:

- Class internals are protected from accidental user-level errors, which might corrupt the state of the object.
- Any change in class implementation may be done over time in response to changing requirements, without changing the statements of the class..

2.2.4 Data Encapsulation

All C++ programs are composed of two fundamental elements, functions and data. **Encapsulation** is an OOP concept that binds together the data and functions that manipulate the data, and keeps both data and function safe from outside interference and misuse.

C++ implements encapsulation and data hiding through the declaration of a class. A C++ class can contain `private`, `protected` and `public` members (See Figure 2.6). By default, all items defined in a class are `private`. Members declared under `private` section are not visible outside the class. Members declared as protected are visible to its derived class also (explained in Section 2.2.6) , but not outside the class.

In Student class of our school software, the variables Regno, Name, Mark1, Mark2, Mark3 and Fee are to be declared private. This means that they can be accessed only by member functions of the Student class, and not by any other part of our program. Here data is hidden and encapsulation is achieved.

To make parts of a class accessible to other parts of our program, we must declare them under the public section. All variables or functions defined after the public access specifier are accessible anywhere in our program. For example, in Student class of our school software, all member functions in it that need to be called by other objects are to be declared public, in order to make them visible outside the class.

2.2.5 Modularity

When we write a program, we try to solve a problem by decomposing the problem into small sub-problems and then try to solve each sub-problem separately. Solution to each sub-problem is a separate component that includes interface, specification and implementation.

Modularity is a concept through which a program is partitioned into modules that can be considered and written on their own, with no consideration of any other module. These modules are later linked together to build the complete software. These modules communicate with each other by passing messages.

We have already studied in detail about implementing modularity using functions. (See Chapter 10, Functions in class XI text book). In object oriented-programming, modularity is implemented with the help of class. For example, in our school software we can separate everything related to students and teachers into two separate modules (See Figure 2.9) using the concept of class.

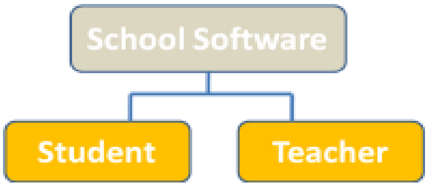


Fig. 2.9: Modularity

2.2.6 Inheritance

Inheritance is the process by which objects of one class acquire the properties and functionalities of another class. Inheritance supports the concept of hierarchical classification and reusability.

Let us take a real life example to explain the scenario. In Figure 2.10, land vehicle and water vehicle acquires the properties (i.e. data members and member functions) of vehicle. Again car and truck acquires the properties of land `vehicle` (i.e. `car/truck = vehicle + land vehicle`) and `boat` acquires the properties of `water vehicle` (i.e. `boat = vehicle + water vehicle`). In the case of `hovercraft` which travels both in land and water, it acquires the properties of both `land vehicle` and `water vehicle` (i.e. `hovercraft = vehicle + land vehicle + water vehicle`). A `car` can have further classification such as hatchback, sedan, SUV etc., which will acquire the properties from `car`, `land vehicle` and `vehicle`, but will still have some specific properties of its own. Thus, the level of hierarchy can be extended to any level.

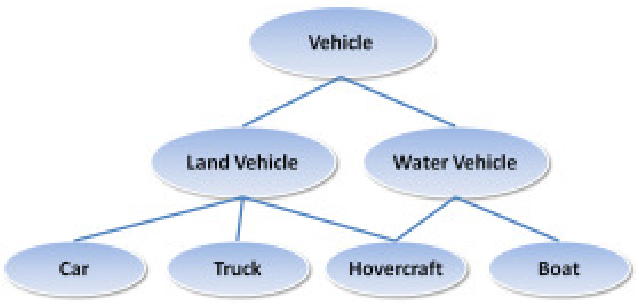


Fig. 2.10: Inheritance in real world

In the above example, if the common properties and functionalities of both land vehicle and water vehicle were not separated and placed in vehicle, then it would have to be repeated in both the classes. This would have increased the size of the program and the time taken for coding and debugging. Keeping it in mind, if we observe from top to bottom of the chart, we can understand how complexity is reduced greatly through the introduction of inheritance.

Once a class is written, created and debugged, if needed, it can be distributed for use in other programs. This is called **reusability**. In OOP, the concept of inheritance provides an important extension to the idea of reusability. Through this we can add additional features to an existing class without modifying it. This is made possible through deriving a new class from the existing one. The new class will inherit the capabilities of the old one, and can add features of its own. The existing class is called the **base class**, and the new class is referred to as the **derived class**. The derived class will have combined features of both the classes. Any number of classes can be derived from an existing class. Figure 2.11 shows the concept of derivation of new classes.

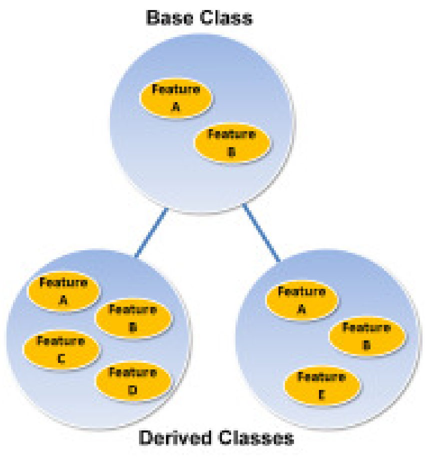
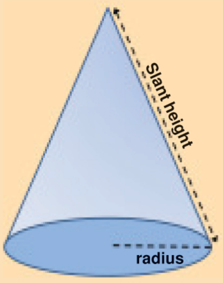


Fig. 2.11: Inheritance in OOP

The following program derives a class 'Cone' from class 'Circle' through inheritance. As radius(r) is already declared in the base class 'Circle', the derived class 'Cone' needs to declare only one data member the slant height(s). Two member functions, one to accept data and other to display the area of cone are also declared. The data member of the class 'Circle' are to be declared 'protected' so that they can be accessed by the derived class 'Cone'.



```
 #include <iostream>
using namespace std;
class Circle
{ protected:
     float r;
   public:
     void get_radius(){
       cout << "Enter Radius :";
       cin >> r;
     }
     void display_area(){
      cout<< "Area:"<< 3.14*r*r;
     }
};
```

Base Class

```
class Cone : public Circle
{ private:
     float s;
```

New member

```
   public:
     void get_cone_data() {
       get_radius();
       cout << "Enter slant height:";
       cin >> s;
     }
     void display_cone_area(){
     cout << "Area :" << 3.14*r*(s+r);
     }
};
```

New member functions

Derived class

```
int main()     {
     Cone C1;
     C1.get_cone_data(); //Sending message
     C1.display_cone_area(); //Sending message
}
```

Main function

Output :
```
Enter Radius : 2.0
Enter slant height: 5.0
Area :43.96
```

Different forms of Inheritance are Single Inheritance, Multiple Inheritance, Multilevel Inheritance, Hierarchical Inheritance, and Hybrid Inheritance (see Figure 2.12).

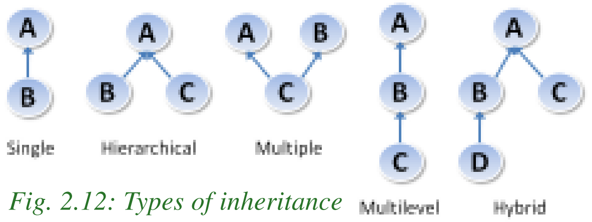


Fig. 2.12: Types of inheritance

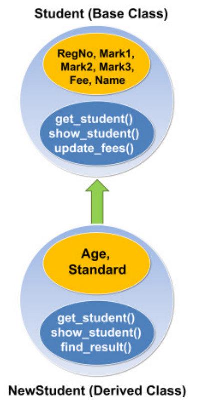


Fig.: 2.13 Sample Inheritance

Let us try to implement the concept of inheritance in our school software. Assume that in addition to existing data and function, we need to add new data - age of the student, and standard in which the student is studying and a function to find the examination result, to `Student` class. Instead of modifying the existing `Student` class, we can derive a class named `NewStudent` from the existing `Student` class, so that the `Student` class remains undisturbed. Here `Student` is the base class and `NewStudent` is the derived class (see Figure 2.13).

The syntax for declaring a derived class is as follows:

```
class derived_class: AccessSpecifier base_class
{
   //declaration of members and member functions
 };
```

where `derived_class` is the name of the derived class and `base_class` is the name of the class on which it is based. The `AccessSpecifier` may be `public`, `protected` or `private`. This access specifier describes the access level for the members that are inherited from the base class.

2.2.7 Polymorphism

'Poly' means many. 'Morph' means shapes. So polymorphism can be defined as the ability to express different forms. This is demonstrated in Figure 2.14. Here the same command "Now Speak" is issued to all objects, but each object responds differently to the same command.

In object-oriented programming, polymorphism refers to the ability of a programming language to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.

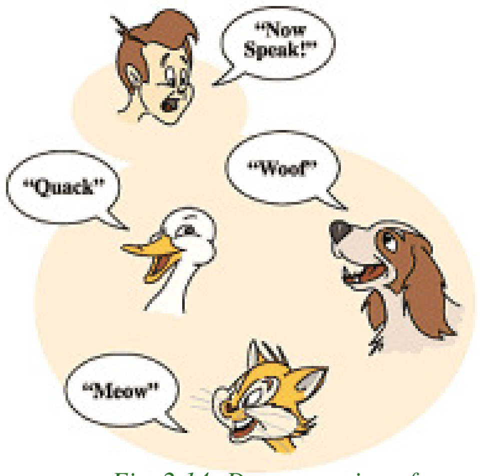


Fig. 2.14: Demonstration of Polymorphism

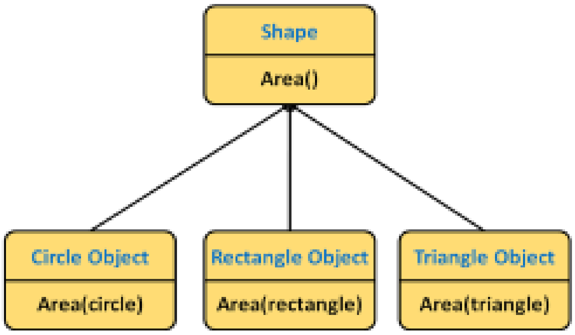


Fig.: 2.15 Example of Polymorphism

For example, given a base class `Shape`, polymorphism enables the programmer to define different `area` methods for any number of derived classes such as `circle`, `rectangle` and `triangle`. No matter what shape an object has, applying the `area` method to it will return the correct results.

There are two types of polymorphism.

a. Compile time (early binding/static) polymorphism

b. Run time (late binding/dynamic) polymorphism

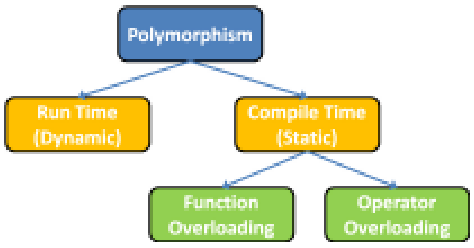


Fig.: 2.16 Classification of Polymorphism

a. Compile time polymorphism

Compile time polymorphism refers to the ability of the compiler to relate or bind a function call with the function definition during compilation itself. Function overloading and operator overloading comes under compile time polymorphism.

Function Overloading: Functions with the same name, but different signatures can act differently. For example `area(int)` can be used to find the area of a square whereas `area(int, int)` can be used to find the area of a rectangle. Thus, the same function `area()` acts in two different ways depending on its signature. Defining multiple functions with the same name and different function signatures is known as function overloading.

Operator overloading: Operator overloading is the concept of giving new meaning to an existing C++ operator (like +, -, =, * etc.). It makes it possible to use the ordinary operator to exhibit different behaviors on different objects of a class, depending on the types of operands it receives. To overload an operator, we need to write a member function for the operator we are overloading.

For example, the + (plus) operator in C++ is already overloaded as it can do integer addition (`4 + 5`) and floating point addition (`3.14 + 2.6`). If needed, we can add additional functionality to it and make it add two objects. For example `T1 = T2 + T3`, where `T1, T2` and `T3` are all objects of a class named '`time`'. Here '+' may be used to add two time sequences represented in HH:MM:SS format.

b. Run time polymorphism

Run time polymorphism refers to the binding of a function definition with the function call during runtime. It uses the concept of pointers and inheritance.

The following program implements function overloading, to find the area of a square and a rectangle. It defines two functions, one to find the area of a square and the other to find the area of a rectangle. Both functions are given the same name 'area' but have different signatures.

Function names are same

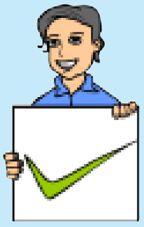
Signatures are different

```
#include<ciostream>
using namespace std;
int area(int s){ //To find the area of a square
return s * s;
}
int area(int s1, int s2){ //To find area of a
return (s1 * s2);          //rectangle
}
int main()
{
cout << "Area of Square:" << area(5); << endl;
cout << "Area of Rectangle:"<< area (7,2);
}
```

Output:

```
   Area of Square: 25
   Area of Rectangle: 14
```

Know your progress



1. The wrapping up of data and functions into a single unit is referred to as __________.
2. Access to data is restricted by the feature known as __________.
3. Objects normally communicate with each other through __________.
4. C++ supports __________ and __________ binding.
5. Late binding is also called __________.
6. Early binding is also called __________.
7. What are the different types of inheritance?

Let us conclude

As software makes the computer a useful machine, the development and maintenance of software requires special consideration. In order to make software development more productive and to reduce maintenance costs, various methods/paradigms have been tried. They are Structured paradigm, Procedural paradigm, Modular paradigm and Object-oriented paradigm (OOP). Most of the latest and widely-used programming languages follow OOP. OOP implements a problem using objects that can communicate with each other. Here, data is given more importance than in previous paradigms. To give maximum protection to data from unauthorised access they are defined using various access specifiers and kept along with functions that operate on the data. OOP also provides effective modularisation and features to improve reusability and extensibility of a code, and can implement static and dynamic polymorphism.

Let us assess

1. Protecting data from unauthorised access is ___________.

   a. Polymorphism b. Encapsulation c. Data abstraction d. Inheritance

2. A base class may also be called ___________.

   a. Child class b. Subclass c. Derived class d. Parent class

3. Which of the following is not a type of inheritance?

   a. Hybrid b. Multiple c. Multilevel d. Multiclass

4. Subclass is the same as:

   a. Derived class b. Super class c. Base class d. None of these

5. Default access specifier is :

   a. public b. private c. protected d. none

6. Which of the following is not an OOP concept?

   a. Overloading b. Procedural programming

   c. Data abstraction d. Inheritance

7. The ability of a message or data to be processed in more than one form is called

   a. Polymorphism b. Encapsulation c. Data hiding d. Inheritance

8. C++ is a ___________ language.

   a. Object based b. Non-procedural c. Object oriented d. Procedural

9. Which of the following is not a characteristic of OOP?
   a. It emphasizes procedure more than data.
   b. It offers good real world modelling.
   c. It wraps up related data items and associated functions in the unit.
   d. None of these.
10. Which among the following is true about OOPs?
   a. It supports data abstraction
   b. It supports polymorphism
   c. It supports structured programming
   d. It supports all of these
11. What do you mean by programming paradigm? Name the programming paradigms.
12. What are the limitations of procedural programming approaches?
13. What is object oriented-programming paradigm? List the basic concepts of OOP.
14. How does OOP implemented in C++?
15. What is encapsulation?
16. Distingush between an object and a class?
17. What is a base class and a sub class? What is the relationship between base class and subclass?
18. Explain the concept of data abstraction. Give an example.
19. Write a short note on inheritance.
20. To operate a car, we use behaviors such as steering, brakes, accelerator etc. All we know is how to use these. We need not know what happens internally when we apply these. Can you connect this with any of the OOP concept? Explain?
21. What do you mean by inheritance? How does this support 'reusability'?
22. What is polymorphism? Give an example to illustrate this feature.
23. Explain the concept of OOP with examples.
24. There is a plug point with a switch. What a switch "does", depends on what is connected to the plug point, and the context in which it is used. Can you connect this with any of the OOP concepts? Explain?
25. Let us assume there is a base class named '`LivingBeings`' from which the subclasses '`Horse`', '`Fish`' and '`Bird`' are derived. Let us also assume that the `LivingBeings` class has a function named '`Move`', which is inherited by all subclasses mentioned. When the `Move` function is called in an object of the `Horse` class, the function might respond by displaying trotting on the screen. On the other hand, when the same function is called in an object of the `Fish` class, swimming might be displayed on the screen. In the case of a `Bird` object, it may be flying. Can you connect this with any of the OOP concept? Explain?