RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL
New Scheme Based On AICTE Flexible Curricula

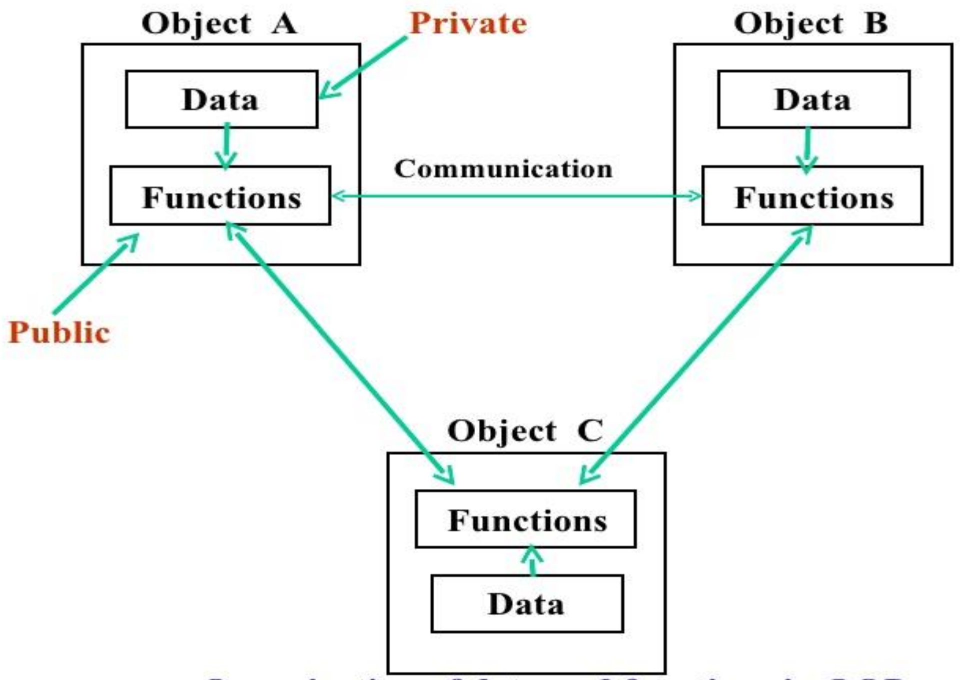
B. Tech. First Year
BT-205 Basic Computer Engineer

Topic Covered

Object & Classes, Scope Resolution Operator, Constructors & Destructors, Friend Functions, Inheritance, Polymorphism, Overloading Functions & Operators, Types of Inheritance, Virtual functions. Introduction to Data Structures.

Object Oriented Programming

Object Oriented Paradigm The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig.1.1. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



Organization of data and functions in OOP

Fig 1.1. Organization of data and function in OOP.

Some of the features of object oriented programming are:

- ❖ Emphasis is on data rather than procedure.
- ❖ Programs are divided into what are known as objects.
- ❖ Data structures are designed such that they characterize the objects.
- ❖ Functions that operate on the data of an object are ties together in the data structure.
- ❖ Data is hidden and cannot be accessed by external function.
- ❖ Objects may communicate with each other through function. • New data and functions can be easily added whenever necessary.
- ❖ Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

Basic Concepts of Object Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- ❖ Objects
- ❖ Classes
- ❖ Data abstraction and encapsulation
- ❖ Inheritance
- ❖ Polymorphism
- ❖ Dynamic binding
- ❖ Message passing

Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c. When a program is executed, the objects interact by sending messages to one another. For example, if “customer” and “account” are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other’s data or code. It is a sufficient to know the type of message accepted, and the type of response returned by the objects. Although different author represent them differently fig 1.2 shows two notations that are popularly used in object oriented analysis and design.

```
OBJECTS: STUDENT

DATA
        Name
        Date-of-birth
        Marks

FUNCTIONS
        Total
        Average
        Display
        .........
```

Fig. 1.2 representing an object

Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement Fruit Mango; will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation

The wrapping up of data and function into a single unit (called class) is known as encapsulation. Data and encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding. Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created. The attributes are some time called data members because they hold information. The functions that operate on these data are sometimes called methods or member function.

Inheritance

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification. For example, the bird, 'robin' is a part of class 'flying bird' which is again a part of the class 'bird'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.3. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined feature of both the classes. The real appeal and power of the inheritance mechanism is that it

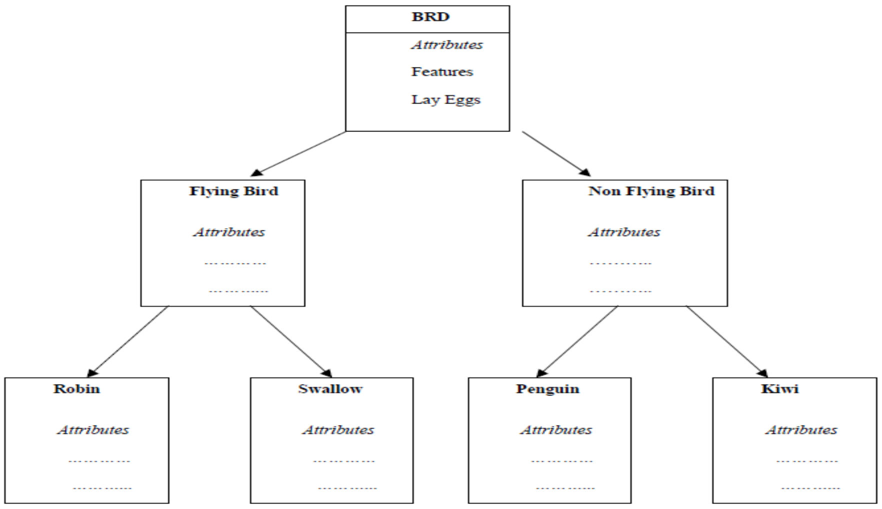


Fig 1.3. Property inheritances.

Allows the programmer to reuse a class i.e. almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduced any undesirable side-effects into the rest of classes.

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than on form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading. Fig. 1.4 illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending upon the context. Using a single function name to perform different type of task is known as function overloading.

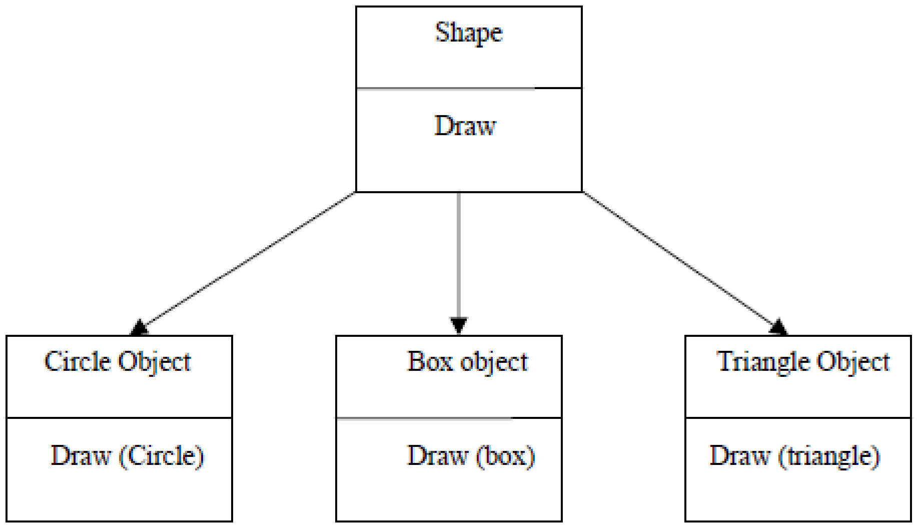


Fig 1.4. Polymorphism.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference. Consider the procedure “draw” in fig. 1.3. by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, involves the following basic steps:

1. Creating classes that define object and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real world counterparts. A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.

Class:

A class is a user defined data type which binds data and its associated functions together. It allows the data and functions to be hidden, if necessary from external use. Generally, a class specification has two parts.

i) Class declaration: it describes the type & scope of its members.

ii) Class function definitions: It describes how the class functions are implemented.

Class declaration.

The general form of a class is

```
class class-name
{
private:
variable declaration;
function declaration;
public:
variable declaration;
function declaration;
} ;
```

1) The class keyword specifies that what follows is an abstract data of type class name. The body of a class is enclosed within braces & terminated by semicolon.

2) The class body consists of declaration of variables & functions which are called as members & they are grouped under two sections i.e. private & public.

3) Private and public are known as visibility labels, where private can be accessed only from within the class where public members can be accessed from outside the class also. By default, members of a class are private.

4) The variable declared inside the class are known as data members & functions are known as member functions. Only the member function can have access to the private data members & private functions. However the public members can be accessed from outside the class.

Simple class example

```
class item
{
int number; variable
float cost; declaration
public :
void getdata (int a, float b); function
void putdata (void); declaration
};
```

In above class class-name is item. These class data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variable number & cost, and putdata() for displaying their values. These functions provide the only access to data members of the class.

Creating objects

Object is an instance or variable of the class. Once a class has been declared. We can create variables of that type by using the class name.

Ex . item x;

Here class variables are known as object therefore, x is called an object of type item and necessary memory space is allocated to an object.

Example of class and object

```
# include<iostream>
using namespace std;
class item // class declaration
{
int number; // private by default
float cost;
public :
void getdata(int a float b);
void putdata(void) // function defined here
{
cout <<.number .<< number << .\n.;
cout << cost: << cost << .\n.
}
};
// member functions definition
void item :: getdata(int a, float b)
{
number = a;
cost = b;
}
Int main()
{
item x; // create object x
x.getdata(100, 20.3);
x.putdata();
}
```

In the above program we have shown that one member functions is inline and other is external member function. Here is the output of program.

OUTPUT

number : 100

cost : 20.3

Member Function Definition

The class specification can be done in two part :

(i) Class definition. It describes both data members and member functions.

(ii) Class method definitions. It describes how certain class member functions are coded.

We have already seen the class definition syntax as well as an example. In C++, the member functions can be coded in two ways :

(a) Inside class definition

(b) Outside class definition using scope resolution operator (::) The code of the function is same in both the cases, but the function header is different as explained below :

Inside Class Definition:

When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as inline functions. In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster but memory penalty is there.

Outside Class Definition Using Scope Resolution Operator (::) :

In this case the function's full name (qualified_name) is written as shown:

Name_of_the_class :: function_name

The syntax for a member function definition outside the class definition is :

```
return_type name_of_the_class::function_name (argument list)
{
body of function
}
```

Here the operator::known as scope resolution operator helps in defining the member function outside the class.

SCOPE RESOLUTION OPERATOR

Member functions can be defined within the class definition or separately using scope resolution operator (::). Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. Defining a member function using scope resolution operator uses following declaration

```
return-type class-name:: func-name(parameter- list)
{
 // body of function
 }
```

Here the class-name is the name of the class to which the function belongs. The scope resolution operator (::) tells the compiler that the function func-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified.

```
Class myclass
 {
 int a;
 public:
 void set_a(int num);  //member function declaration
int get_a( ); //member function declaration
 };  //member function definition outside class using scope resolution operator
void myclass :: set_a(intnum)
{
 a=num;
}
 int myclass::get_a( )
{
 return a;
 }
```

Another use of scope resolution operator is to allow access to the global version of a variable. In many situation, it happens that the name of global variable and the name of the local variable are same .In this while accessing the variable, the priority is given to the local variable by the compiler. If we want to access or use the global variable, then the scope resolution operator (::) is used.

Constructor:

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the value data members of the class.

- ❖ The constructor functions have some special characteristics.
- ❖ They should be declared in the public section.
- ❖ They are invoked automatically when the objects are created.
- ❖ They do not have return types, not even void and therefore, they cannot return values.
- ❖ They cannot be inherited, though a derived class can call the base class constructor.
- ❖ These cannot be static.
- ❖ Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
- ❖ These can have default arguments as other C++ functions.

Example:

```
#include< iostream>
Using namespace std;
class myclass
{ // class declaration
int a;
public:
myclass( ); //default constructor
void show( );
};
myclass :: myclass( )
 {
```

```
cout <<"In constructor\n"; a=10;
}
myclass :: show( )
{
 cout<< a;
}
int main( )
{
int ob; // automatic call to constructor
ob.show( );
return0;
}
```

In this simple example the constructor is called when the object is created, and the constructor initializes the private variable a to10.

Default constructor

The default constructor for any class is the constructor with no arguments. When no arguments are passed, the constructor will assign the values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses. Above program uses default constructor. If it is not defined explicitly, then it is automatically defined implicitly by the system.

Parameterized Constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
#include <iostream>
Using namespace std;
class myclass
```

```
{
int a, b;
public:
myclass (int i, int j) //parameterized constructor
{
a=i;
b=j;
}
void show()
{
cout << a << " " << b;
}
};
int main()
{
myclass ob(3, 5); //call to constructor
ob.show();
return 0;
}
```

C++ supports constructor overloading. A constructor is said to be overloaded when the same constructor with different number of argument and types of arguments initializes an object.

Copy Constructors

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. If class definition does not explicitly include copy constructor, then the system automatically creates one by default. The copy constructor is used to:

Initialize one object from another of the same type.

Copy an object to pass it as an argument to a function.

Copy an object to return it from a function.

The most common form of copy constructor is shown here:

```
classname (const classname &obj)
{
// body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object. The keyword const is used because obj should not be changed.

<u>DESTRUCTOR</u>

A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be preceded by the character Tilde (~).

A destructor takes no arguments and has no return value.

Each class has exactly one destructor.

If class definition does not explicitly include destructor, then the system automatically creates one by default. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

```
// A Program showing working of constructor and destructor //
#include <iostream>
Using namespace std;

class Myclass
{
public:
int x;
Myclass()   //Constructor
{
x=10;
}
~Myclass() //Destructor
{
cout<<“destructing…”;
```

```
}
int main (}
{
Myclass obj1, obj2;
cout<<ob1.x<<obj2.x;
return 0;
```

output

```
10 10
Destructing……..
```

Friend function

In general, only other members of a class have access to the private members of the class. However, it is possible to allow a nonmember function access to the private members of a class by declaring it as a friend of the class. To make a function a friend of a class, you include its prototype in the class declaration and precede it with the friend keyword. The function is declared with friend keyword. But while defining friend function, it does not use either keyword friend or :: operator. A function can be a friend of more than one class. Member function of one class can be friend functions of another class. In such cases they are defined using the scope resolution operator.

A friend, function has following characteristics.

- It is not in the scope of the class to which it has been declared as friend.
- A friend function cannot be called using the object of that class. If can be invoked like a normal function without help of any object.
- It cannot access the member variables directly & has to use an object name dot membership operator with 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 object as arguments.

Program to illustrate use of friend function

```
#include <iostream>
Using namespace std;
class A
{
int x, y;
public:
friend void display(A &obj);
void getdata()
{
cin>>x>>y;
}
};
void display(A &obj)
{
cout<<obj.x<<obj.y;
}
Int main ()
{
A a;
a.getdata();
display(a);
return 0;
}
```

DATA STRUCTURES

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use. An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored. To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be linear if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Classification of Data Structure.

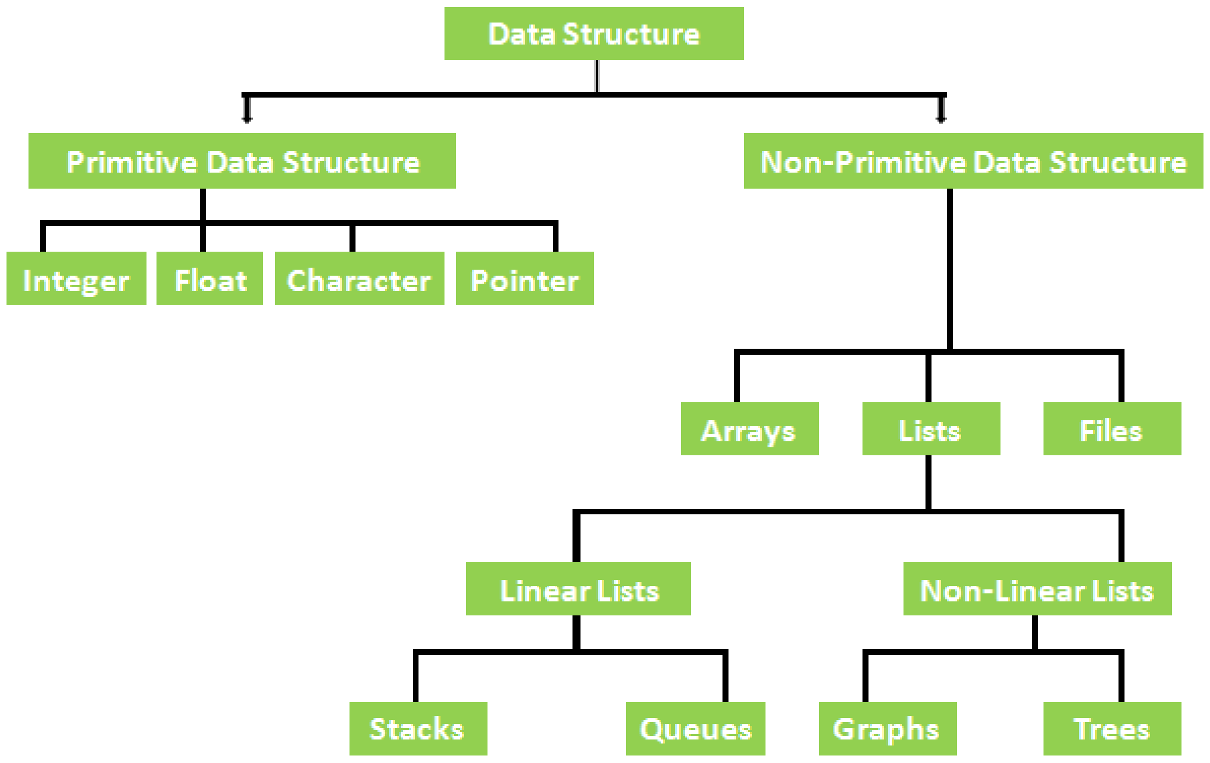


Fig 1.5: Classification of Data Structures.

Data structures are divided into two types:

• Primitive data structures.

• Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.5 shows the classification of data structures.

Data structures: Organization of data

The collections of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:

• Contiguous

• Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.6 below illustrates the difference between contiguous and non-contiguous structures.

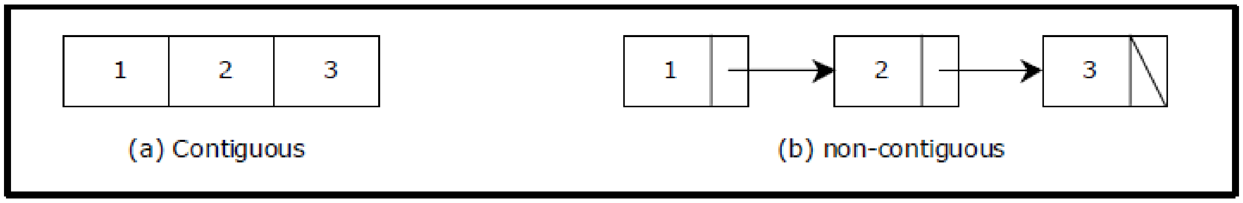


Fig 1.6: Contiguous and Non-contiguous structures compared.

Contiguous structures:

Contiguous structures can be broken drawn further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.6 shows example of each kind. The first kind is called the array. Figure 1.7 (a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.7(b) shows a simple structure consisting of a person’s name and age. In a struct, elements may be of different data types and thus may have different sizes.

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the "mortar" you need to built more exotic form of data structure, including the non-contiguous forms.

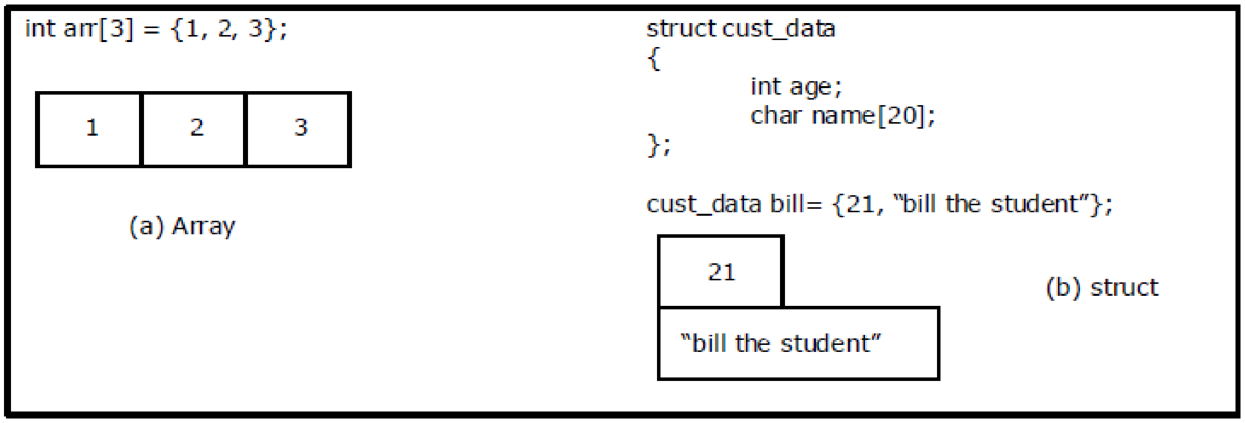


Fig 1.7: Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards.

A tree such as shown in figure 1.8(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right.

In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.8(c) is an example of a graph.

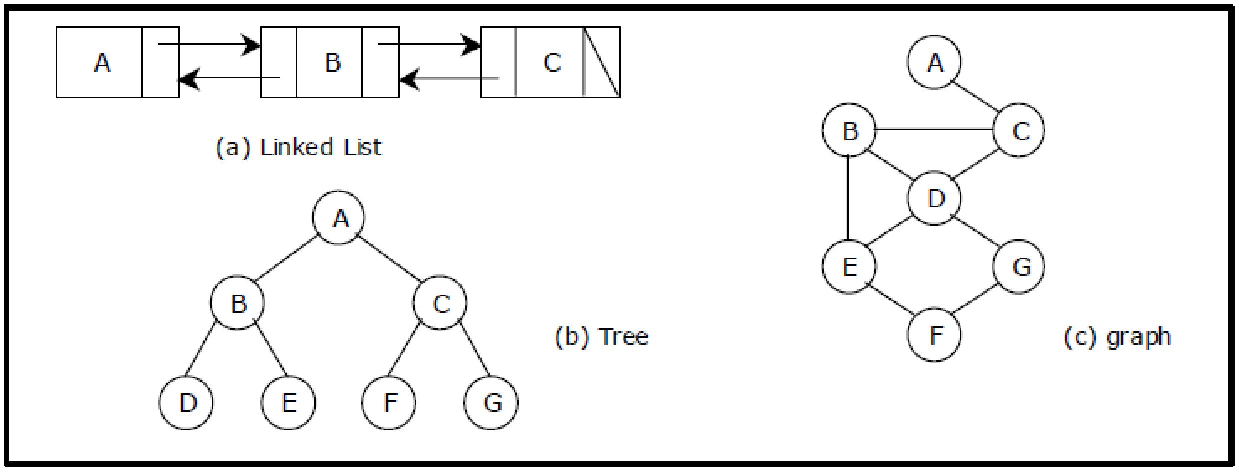


Fig1.8: Examples of non-contiguous structures.

Difference between Linear and Non Linear Data Structure

Linear Data Structure	**Non-Linear Data Structure**
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

OPERATION ON DATA STRUCTURES

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types:-

- **Create:-** The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.
- **Destroy:-** Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
- **Selection:-** Selection operation deals with accessing a particular data within a data structure.

- **Updation:-** It updates or modifies the data in the data structure.
- **Searching:-** It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
- **Sorting:-** Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
- **Merging:-** Merging is a process of combining the data items of two different sorted list into a single sorted list.
- **Splitting:-** Splitting is a process of partitioning single list to multiple list.
- **Traversal:-** Traversal is a process of visiting each and every node of a list in systematic manner.

Bibliography

- http://www.ddegjust.ac.in/studymaterial/mca-3/ms-17.pdf
- https://www.softwaretestinghelp.com/object-oriented-programming-in-cpp/
- https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/
- https://ambedkarcollegevasai.com/wp-content/uploads/2019/03/CPP.pdf
- https://www.programmingsimplified.com/cpp/source-code/scope-resolution-operator

Inheritance:

Inheritance is the most powerful feature of Object Oriented programming. Inheritance is the process of creating new classes, called derived classes from existing or bases classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. A class, called the derived class, can inherit the features of another class, called the base class. Generally every base class has a list of qualities and features. The main theme in this inheritance is to share all the common characteristics of base class to derived classes.

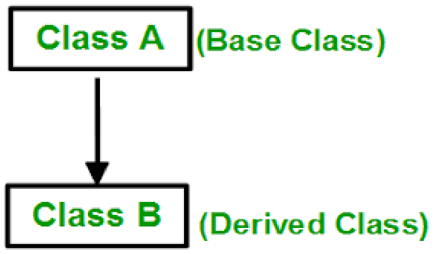


Figure-1.9 Inheritance Overview

Inheritance has an important feature to allow reusability. One result of reusability is the ease of distributing class libraries. A programmer can use a class created another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>
 using namespace std;
// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }
  protected:
     int width;
     int height;
};
// Derived class
class Rectangle: public Shape {
   public:
     int getArea() {
        return (width * height);
     }
};
int main(void) {
   Rectangle Rect;
   Rect.setWidth(5);
   Rect.setHeight(7);
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.

- The friend functions of the base class.

Advantages of Inheritance

- The main advantage of the inheritance is that it helps in the reusability of the code. The codes are defined only once and can be used multiple times. In java, we define the superclass or base class in which we define the functionalities and any number of child classes can use the functionalities at any time.
- Through inheritance a lot of time and efforts are being saved.
- It improves the program structure which can be readable.
- The program structure is short and concise which is more reliable.
- The codes are easy to debug. Inheritance allows the program to capture the bugs easily
- Inheritance makes the application code more flexible to change.
- Inheritance results in better organization of codes into smaller, simpler and simpler compilation units.

Type of Inheritance based on access types:

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied −

- Public Inheritance − When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
- Protected Inheritance − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
- Private Inheritance − When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Types of Inheritance based on derived classes:

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

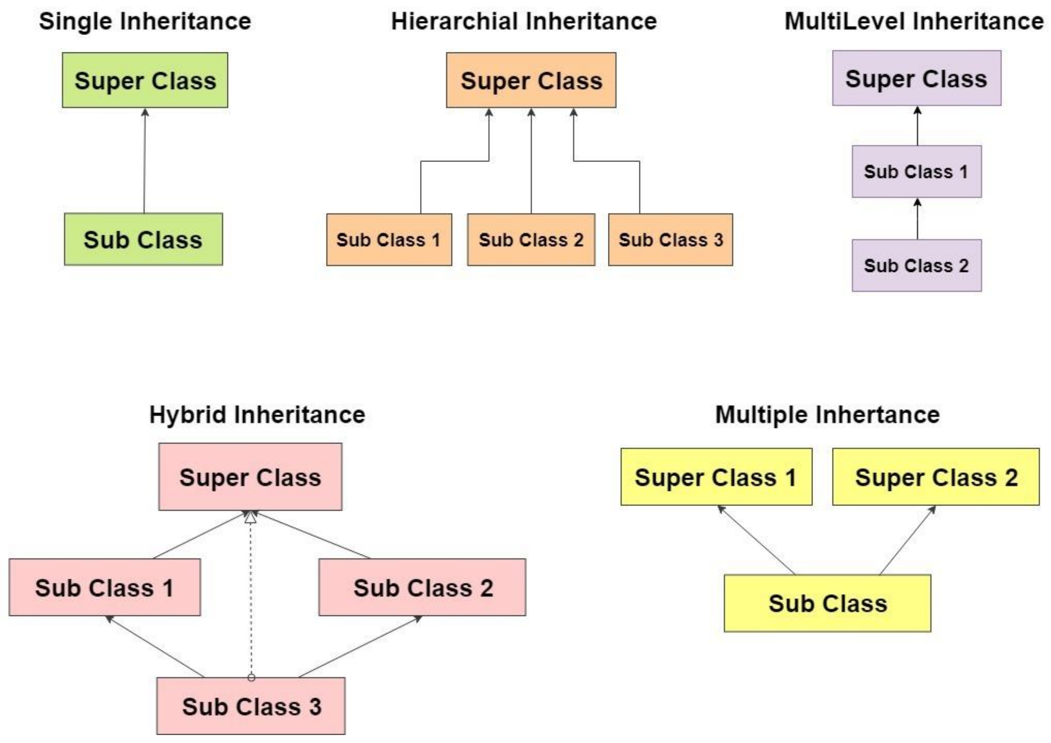


Figure-1.10- Types of Inheritance

1) Single Inheritance

```
 #include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class";
  }
};
int main() {
  //Creating object of class B
  B obj;
  return 0;
}
```

Output:

```
Constructor of A class
Constructor of B class
```

2) Multilevel Inheritance

In this type of inheritance one class inherits another child class.

```
C inherits B and B inherits A
```

Example of Multilevel inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
    cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
    cout<<"Constructor of B class"<<endl;
  }
};
class C: public B {
public:
  C(){
    cout<<"Constructor of C class"<<endl;
  }
};
int main() {
  //Creating object of class C
  C obj;
  return 0;
}
```

Output:

```
Constructor of A class
Constructor of B class
Constructor of C class
```

3) Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.
For example:

```
C inherits A and B both
```

Example of Multiple Inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A, public B {
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
Constructor of A class
Constructor of B class
Constructor of C class
```

4) Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class. For example:

```
Class B and C inherits class A
```

Example of Hierarchical inheritance:

```
#include <iostream>
using namespace std;
class A {
public:
  A(){
     cout<<"Constructor of A class"<<endl;
  }
};
class B: public A {
public:
  B(){
     cout<<"Constructor of B class"<<endl;
  }
};
class C: public A{
public:
  C(){
     cout<<"Constructor of C class"<<endl;
  }
};
int main() {
   //Creating object of class C
   C obj;
   return 0;
}
```

Output:

```
Constructor of A class
Constructor of C class
```

5) Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.

Polymorphism:

Polymorphism is a feature of OOPs that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:

1)Compile time Polymorphism – This is also known as static (or early) binding.
2) Runtime Polymorphism – This is also known as dynamic (or late) binding.

1) Compile time Polymorphism:

Function overloading and Operator overloading are perfect example of Compile time Polymorphism.

Compile time Polymorphism Example

In this example, we have two functions with same name but different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time thats why it is called compile time polymorphism.

```
#include <iostream>
using namespace std;
class Add {
public:
  int sum(int num1, int num2){
     return num1+num2;
  }
  int sum(int num1, int num2, int num3){
     return num1+num2+num3;
  }
};
int main() {
  Add obj;
  //This will call the first function
  cout<<"Output: "<<obj.sum(10, 20)<<endl;
  //This will call the second function
  cout<<"Output: "<<obj.sum(11, 22, 33);
  return 0;
}
```

Output:

```
Output: 30
Output: 66
```

Operator Overloading:

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

Example:

```
 #include<iostream>

using namespace std;

class Complex {

private:

        int real, imag;

public:

        Complex(int r = 0, int i =0) {real = r; imag = i;}


        // This is automatically called when '+' is used with

        // between two Complex objects

        Complex operator + (Complex const &obj) {

                Complex res;

                res.real = real + obj.real;

                res.imag = imag + obj.imag;

                return res;

        }

        void print() { cout << real << " + i" << imag << endl; }

};

int main()

{

        Complex c1(10, 5), c2(2, 4);

        Complex c3 = c1 + c2; // An example call to "operator+"
        c3.print();
}
```

Output

```
12 + i9
```

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

4) Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed

2) Runtime Polymorphism

Function overriding is an example of Runtime polymorphism. **Function Overriding**: When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class.

In case of function overriding we have two definitions of the same function; one is parent class and one in child class. The call to the function is determined at **runtime** to decide which definition of the function is to be called, that's the reason it is called runtime polymorphism.

Example of Runtime Polymorphism

```
#include <iostream>
using namespace std;
class A {
public:
  void disp(){
    cout<<"Super Class Function"<<endl;
  }
};
class B: public A{
public:
  void disp(){
    cout<<"Sub Class Function";
  }
};
int main() {
  //Parent class object
  A obj;
  obj.disp();
  //Child class object
  B obj2;
  obj2.disp();
  return 0;
}
```

Output:

```
Super Class Function
Sub Class Function
```

Virtual Function:

When we declare a function as virtual in a class, all the sub classes that override this function have their function implementation as virtual by default (whether they mark them virtual or not). **Why we declare a function virtual?** To let compiler know that the call to this function needs to be resolved at runtime (also known as **late binding** and dynamic linking) so that the object type is determined and the correct version of the function is called.

Let's take an example to understand what happens when we don't mark an overridden function as virtual.

Example 1: Overriding a non-virtual function

See the problem here. Even though we have the parent class pointer pointing to the instance (object) of child class, the parent class version of the function is invoked.

You may be thinking why I have created the pointer, I could have simply created the object of child class like this: Dog obj; and assigned the Dog instance to it. Well, in this example I have only one child class but when we a big project having several child classes, creating the object of child class separately is not recommended as it increases the complexity and the code become error prone. More clarity to this after this example.

Output:

```
This is a generic Function
```

Example 2: Using Virtual Function

See in this case the output is Woof, which is what we expect. What happens in this case? Since we marked the function animalSound() as virtual, the call to the function is resolved at runtime, compiler determines the type of the object at runtime and calls the appropriate function.

```
#include<iostream>
using namespace std;
//Parent class or super class or base class
class Animal{
public:
  virtual void animalSound(){
    cout<<"This is a generic Function";
  }
};
//child class or sub class or derived class
class Dog : public Animal{
public:
  void animalSound(){
    cout<<"Woof";
  }
};
int main(){
  Animal *obj;
  obj = new Dog();
  obj->animalSound();
  return 0;
}
```

Output:

```
Woof
```

Bibliography

http://www.cplusplus.com/doc/tutorial/polymorphism/

https://beginnersbook.com/2017/08/cpp-function-overloading/

https://www.geeksforgeeks.org/operator-overloading-c/

https://web.cs.dal.ca/~jin/3132/lectures/OOP-07.pdf

RGPV Questions	
What is Object oriented Programming? Explain its Features.	RGPV Nov 2019

What are virtual Functions? What is use of it? Give an example.	RGPV May 2019
What is the use of Operator Overloading? Write a Program to overload Pre and post Operator	RGPV May 2018
What is the benefits of copy constructor? Explain with Suitable Example	RGPV Nov 2018
Explain Following • Friend Function • Scope Resolution Operator	RGPV Nov 2019