Unit-I

Programming Paradigms:

*. Fundamental Style of Programming that defines how the Structure and basic elements of a computer program will be built.

Classification:

*. Monolithic Programming:- Emphasizes on finding a Solution (assembly language & basic)

*. Procedural Programming:- lays Stress on algorithms. (FORTRAN and COBOL)

*. Structured Programming:- focuses on modules (C, PASCAL)

*. Object-Oriented Programming - Emphasizes on classes and objects.

*. Logic - Oriented Programming - goals expressed in predicate calculus.

*. Rule: Oriented Programming - 'If-then-else' rules for computation.

*. Constraint - Oriented Programming - Invariant relationships to solve a problem.

Comparison of Procedural and object-Oriented Programming

=> Procedural Languages - Problems.

A Program is a set of Instructions.

*. Division into functions.

Large procedural programs are divided into functions.

function - clearly defined purpose & clearly defined I/f to other functions

Modules - grouping a number of functions.

Unrestricted acess:

Two kinds of data → local → used in specific functions
→ Global → accessed by any functions.

Real-World Modelling:

*. Arrangement of separate data and functions does a poor job modelling in the real world.

*. Complex real-world objects have both attributes and behaviour.

*. Attributes - chars with specific value.

*. Behaviour - Response to some Stimulus.

Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure.

Procedures, also known as routines, subroutines

or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself.

Languages used in Procedural Programming:

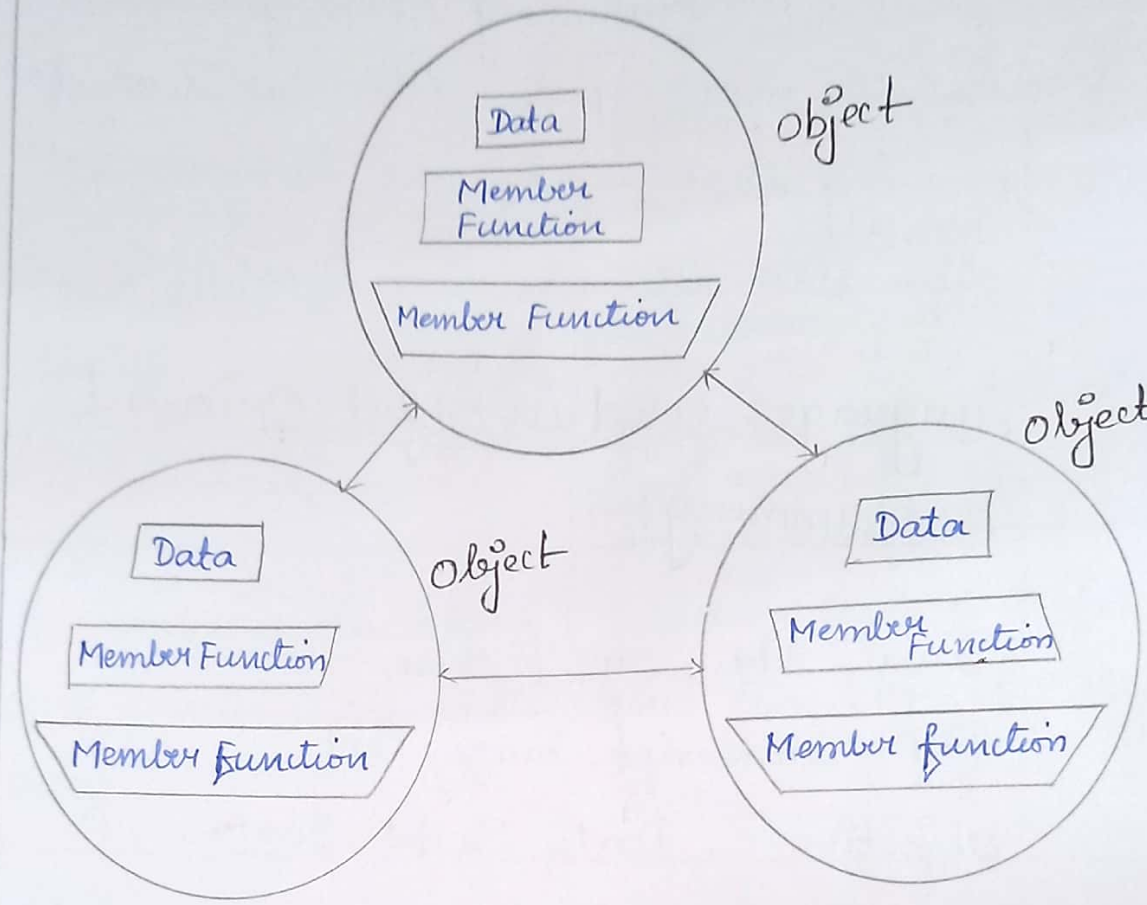
FORTRAN, ALGOL, COBOL,
BASIC, Pascal and C

Object - Oriented Approach:-

Object - data + functions that operate on data.

*. An object functions are called member functions.

*. Data can be accessed only through member functions. Since it is hidden.



Object - Oriented Paradigm

=> Deals with overall organization of the program.

Object oriented programming can be defined as a programming model which is based upon the concept of object. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept

of objects that interact with real world. Object Oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

Languages used in Object Oriented Programming:

Java, C++, C#, Python,
PHP, JavaScript, Ruby, Perl,
objective-C, Dart, Swift, Scala.

Difference between Procedural Programming and object Oriented Programming:

Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is ~~called~~ divided into small parts called Functions.	In Object oriented programming, Program is divided into small parts called Objects.

Procedural programming follows top down approach.	Object oriented programming follows bottom up approach.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc..
Adding new data and function is not easy.	Adding a new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is less secure.	Object oriented programming provides data hiding so it is more secure.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than functions.

In procedural Programming overloading is not possible.	Overloading is possible in object oriented programming.
Procedural programming is based on unreal world.	Object oriented programming is based on real world.
Examples: C, FORTRAN, Pascal, Basic etc.,	Examples: C++, Java, Python, C# etc.,

OOPS features:

Object oriented language must support mechanism to define, create, store, manipulate objects and allow communication b/w object's.

i) classes:

- user defined data types.
- Template or a blue print that describes the structure and behaviour of a set of similar objects.

```
class <class name>
{
Private :-
  <var type> <var-name>
Public :-
    function_name;
};
```

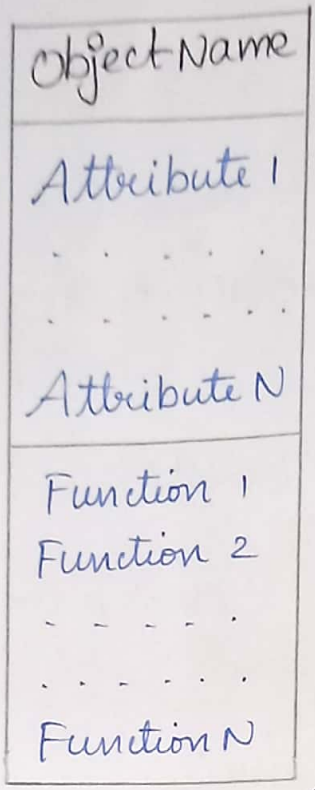
Syntax:

```
class_name  object_name;
```

ii.) Objects of Methods:

*. Every Object contains some data and functions.

*. Methods - Store data in variables and respond to messages that they receive from other objects.



Representation of an object.

Message passing:

2 distinguishable Object's can be have same set of values.

2 objects can be communicate with each other through messages.

Messages has 3 aspects.

- Receiver object.
- Name of the method that the receiver should invoke.

- Parameters to be used with the method.

Inheritance:-

*. Subclass contains the attributes and methods of the parent class

Parent, Base or Super class.

child, Derived or Subclass.

Polymorphism:

Assigning different functionality (or) usage of a variable, functions or an object in different contexts.

Containership:- Ability of a class to contain objects of one or more classes as member data.

Genericity:

Generic programs acts as a model of function or class that can be used to generate functions or classes.

Data Abstractions & Encapsulation:

Abstraction - only essential data are revealed and the implementation details are hidden.

I/O operations:-

i.) Console Output:-

```
/* Beginning of cout-cpp */
void main()
{
  int x;
  x=10;
  cout<<x;
}
/* End */
```

cout => object of the class ostream_with assign- alias for console O/P device.

<< => Insertion operator => Left -> object of ostream class

<iostream.h> => to be included.

ii.) console Input:-

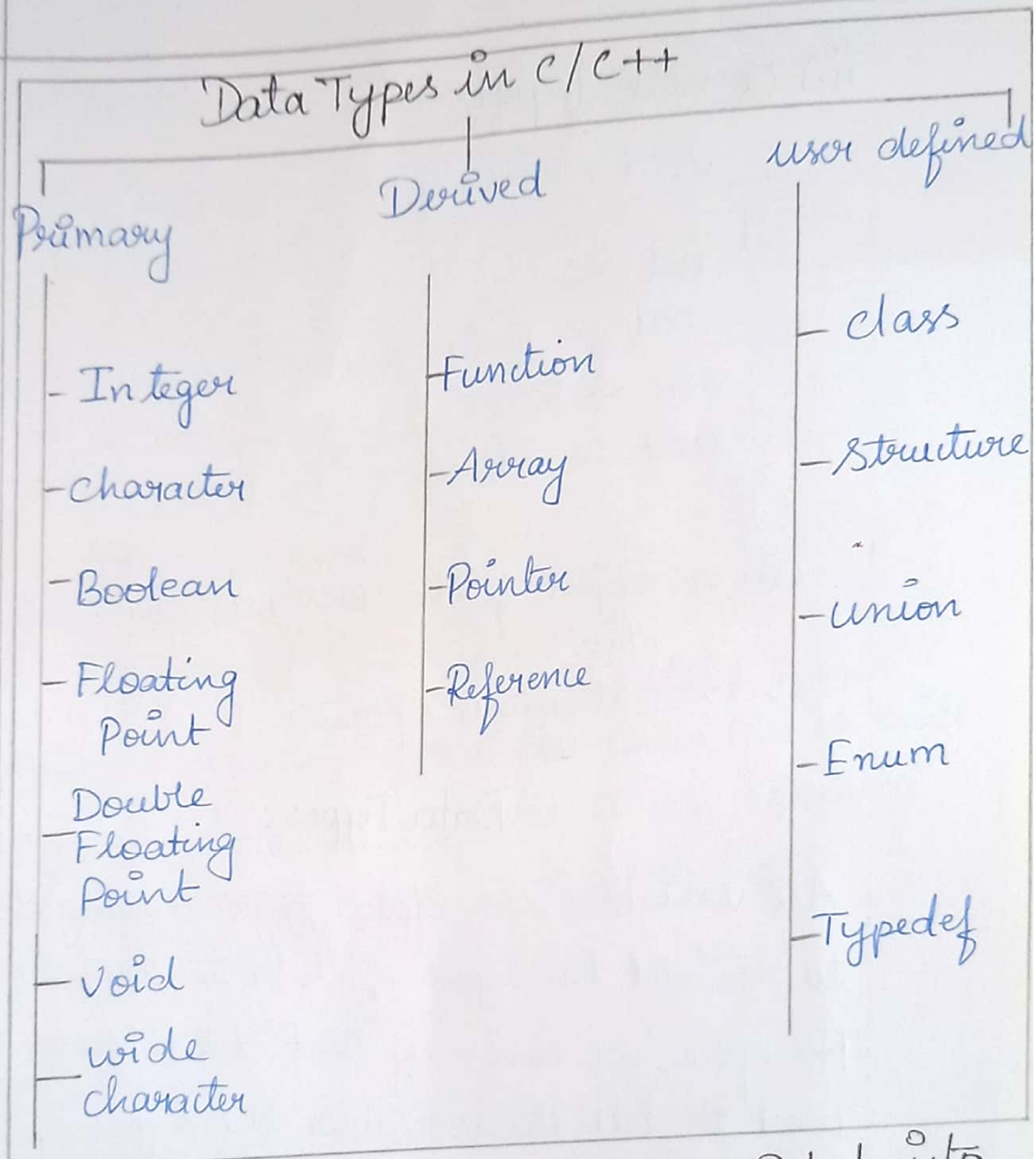
```
Void main ( )
{
int x;
cout << Enter a number " ;
cin >> x
cout << "Value : "<< x;
}
```

cin => object of the class istream-with assign class

>> => Extraction operates

Data Types:

All variables use data-type during declaration to restrict the types of data to be stored. Therefore, we can say that data types are used to tell the variables the types of data it can store. Whenever a variables is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data types requires a different amount of memory.



Data types in C++ is mainly divided into Three types:

1. Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. Eg: int, char, float, bool etc. Primitive

data types available in c++ are:

*. Integer
*. character
*. Boolean
*. Floating point
*. Double floating point
*. Valueless or Void
*. wide character.

2. Derived Data Types: The data-types that are derived from the primitive or built-in datatypes are referred to as Derrived Data Types. These can be of four types namely:

*. Function
*. Array
*. Pointer
*. Reference.

3. Abstract or User-Defined Data Types: These data types are defined by user itself. Like, defining a class in c++ or a structure, c++ provides the following user-defined data-types:

*. Class
*. Structure
*. Union
*. Enumeration
*. Typedef defined Data types.

This article discusses primitive data types available in C++.

*. Integer: Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

*. Character: Character data type is used for storing characters. Keywords used for character data type is char. characters typically requires I byte of memory space and ranges from -128 to 127 or 0 to 255.

*. Boolean: Boolean data type is used for storing boolean or logical values. A boolean

Variable can store either true or false. Keywords used for boolean data type is bool.

*. floating Point: floating point data type is used for storing single precision floating point values or decimal values. Keywords used for floating point data type is float. Float variables typically requires 4 byte of memory space.

*. Double floating Point: Double floating Point data type is used for storing double precision floating point values or decimal values. Keywords used for double floating point data type is Double. Double variables typically requires 8 byte of memory space.

*. void: void means without any value. void data type represents a valueless entity. void data type is used for those functions which does not returns a value.

*. Wide character:

wide character data types is also called as character data type but this data type has size greater than the normal 8-bit datatype. Represented by Wchar_t. It is generally 2 or 4 bytes long.

Datatype Modifiers:

As the name implies, data type modifies are used with the built-in data types to modify the length of data that a particular data type can hold.

Modifiers in C++

Signed	unsigned	Long	Short.
Integer	Integer	Integer	Integer
char	char	Double.	
Long-Prefix.	Short-Prefix		

Data type modifiers available in C++ are:

*. Signed
*. Unsigned
*. Short
*. Long.

Below table summarizes the modified size and range of built-in datatype when combined with the type modifiers:

Data Type	Size (in bytes)	Range:
Short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
Long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int.	8	0 to 4,294,967,295
Long long int	8	-(2^63) to (2^63)-1

unsigned long long int	8	0 to 18,446,744,073,709,551,615
Signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	-
double	8	-
long double	12	-
wchar_t	2or4	1 wide character.

Note: Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.

We can display the size of all data types by using the sizeof() operator and passing the keyword of the datatype as argument to this function as shown below:

//C++ program to sizes of data types.

```
#include <iostream>
using namespace std;

int main()
{
cout << "Size of char: " << sizeof(char)
 << " byte" << endl;
cout << "Size of int: " << sizeof(int)
 << " bytes" << endl;
cout << "Size of short int: " << sizeof(short int)
 << " bytes" << endl;
cout << "Size of long int: " << sizeof(long int)
 << " bytes" << endl;
cout << "Size of signed long int: " << sizeof(signed long int)
 << " bytes" << endl;
cout << "Size of unsigned long int: " << sizeof(unsigned long int)
 << " bytes" << endl;
cout << "Size of float: " << sizeof(float)
 << " bytes" << endl;
cout << "Size of double: " << sizeof(double)
 << " bytes" << endl;
cout << "Size of wchar_t: " << sizeof(wchar_t)
 << " bytes" << endl;
    return 0;
}
```

Output:

Size of char : 1 byte
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes.
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes.

Variables:

Variables are containers for ~~(short)~~ storing data values.

In C++, there are different types of variables (defined with different keywords), for example:

int - stores integers (whole numbers), without decimals, such as 123 or -123
double - stores floating point numbers, with decimals, such as 19.99 or -19.99

Char – Stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes.

String - Stores text, such as "Hello World". String values are surrounded by double quotes.

bool - Stores values with two states; true or false.

Declaring (creating) Variables.

To create a variable, you must specify the type and assign it a value.

Syntax

```
type variable = value;
```

where type is one of c++ types (such as int) and variables is the name of the variable (such as x or myName). The equal sign is used to assign values to the variables.

To create a variables that should store a number, look at the following examples:

Example:

Create a variable called myNum of type int and assign it the value 15:

```
int myNum = 15;
cout << myNum;
```

You can also declare a variable without assigning the value, and assign the value later:

Example:

```
int myNum;
myNum = 15;
cout << myNum;
```

Note that if you assign a new value to an existing variable, it will overwrite the previous value:

Example:

```
int myNum = 15; // myNum is 15
myNum = 10; // Now myNum is 10
cout << myNum; // output 10.
```

Other Types:

A demonstration of other data types:

Example

```
int myNum = 5;  // Integer (whole Number without decimals).

double myFloatNum = 5.99;  // Floating point Number (with decimals).

char myLetter = 'D';  // character
String myText = "Hello";  // string (Text).
bool myBoolean = true;  // Boolean (true or false).
```

Static

Static is a keyword in c++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Since keyword can be used with following:

1. Static variables in functions.
2. Static class objects.

3. Static member Variable in class.
4. Static Methods in class.

1. Static Variables inside functions;

Static Variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on Static storage area, not in Stack.

```
void counter()
{
  static int count=0;
  cout << count++;
}
int main()
{
  for(int i=0;i<5;i++)
  {
    counter();
  }
}
```

output

01234

Let's see the same program's output without using static variable.

```
void counter()
{
   int count = 0;
   cout << count++;
}
   int main()
{
   for (int i=0; i<5; i++)
   {
      counter();
      }
}
```

Output
00000

If we do not use static keyword, the variable count, is reinitialized every time when counter() function is called, and gets destroyed each time when counter() functions ends. But if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

If you don't initialize a static variable, they are by default initialized to zero.

2. Static class objects.

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructions like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
  int i;
  public:
  ABC()
  { i = 0;
    cout << "constructor";
  }
  ~ABC()
```

```
    {
        cout << "destructor";
    }
};
void f()
{
    static Abc obj;
}
int main()
{
    int x = 0
    if (x == 0)
    {
        f();
    }
    cout << "END";
}
```

Output

Constructor END destructor

You must be thinking, why was the destructor not called upon the end of the scope of if condition, where the reference of object obj should get destroyed. This is because object was static, which has scope till the program's lifetime, hence destructor for this object was called when main() function exists.

3. Static Data Member in class.

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each ~~other~~ object, like other non-static data members.

Static member variables (data members) are not initialized using constructor, because these are not dependent on object initializations.

Also, it must be initialized explicitly, always outside the class. if not initialized, Linker will give error.

```
class x
{
  public:
  Static int i;
  X()
  { // constructor
  };
};
```

```
int x::i=1;
int main()
{
    x obj;
    cout << obj.i; // prints value of i
}
```

Output

1

Once the definition for static data member is made, user cannot redefine it. Through, arithmetic operations can be performed on it.

4. Static Member functions:

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access. Operator. But, its more typical to call a static member function by itself, using class name and scope resolution:: operator.

For Example:

```
class x
{
  public:
  static void f()
  {
    // statement
  }
};
int main()
{
   x::f(); // calling member function directly
              with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

Constants:

Constants refers to fixed values that the program may not alter and they are called "literals" (in Bold)

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Use const prefix to declare constants with a specific type as follows:-

const type variable = value;

following example explains it in detail-

```
#include <iostream>
using namespace std;
int main(){
   const int LENGTH = 10;
   const int WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

When the above code is complied and executed, it produces the following result - 50

Pointer

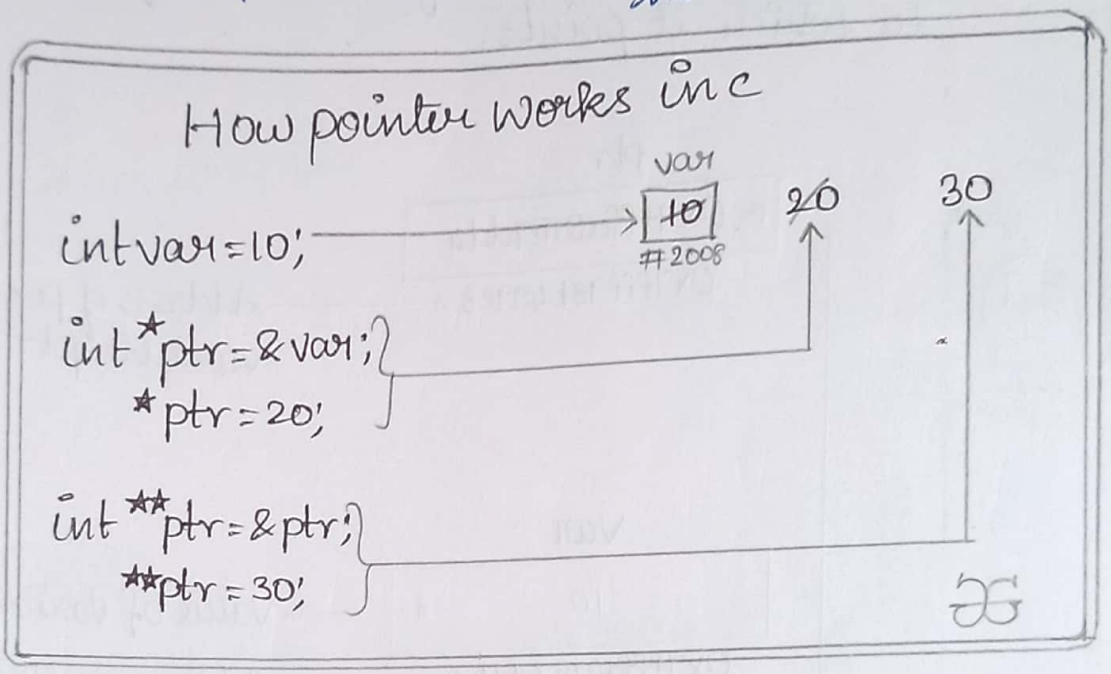
Pointers are Symbolic representation of addresses. They enable programs to stimulate call-by-reference as well as to create and manipulate dynamic data structures. It's general declaration in C/C++ has the format:

[int **ptr = &ptr; *ptr = 30] - face colour.

Syntax:

datatype *var_name;

int *ptr; //ptr can point to an address which holds int data.

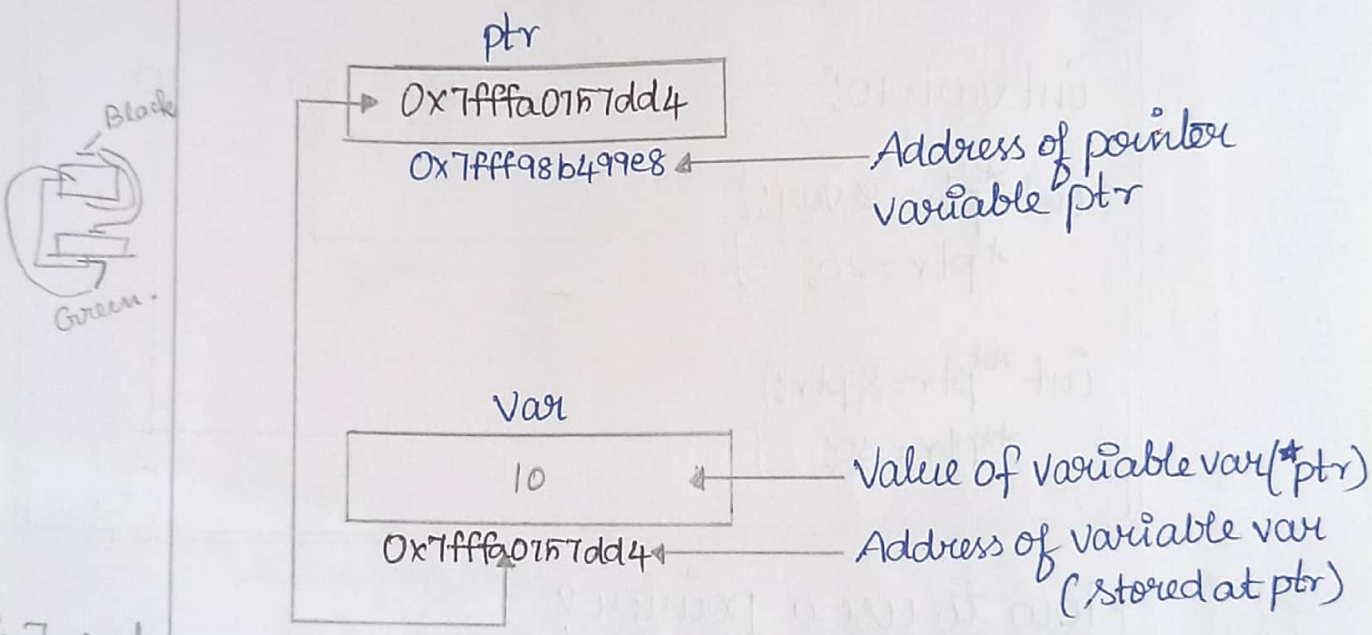


How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

int *ptr=&var; [2of-7 orange] *ptr=20; [line, arrow mark, yellow] [Box, #2008, var, 7, #2008, white.] [Diagram Background [dark green] [title in white] [int var=10;] - Blue.

The reason we associate data type to a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of data type to which it points.



Black

Green.

[10]-red.

```
//C++ program to illustrate Pointers in c++
# include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;
                // declare pointer variable
    int *ptr;
                // note that data type of ptr and var
                must be same:

    ptr = &var;
```

```
    // assign the address of a variable to a pointer
    cout << "value at ptr = " << ptr << "\n";
    cout << "value at var = " << var << "\n";
    cout << "value at *ptr = " << *ptr << "\n";

}
// Driver program
int main()
{
    geeks();
}
```

Output:

```
Value at ptr = 0x7ffcb9e9ea4c
value at var = 20
Value at *ptr = 20.
```

Type Conversions.

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion Also known as 'automatic type conversion'.

- Done by the complier on its own, without any external trigger from the user.

- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double

- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit conversion:

```
// An example of implicit conversion
#include <iostream>
using namespace std;
int main()
```

```
{
    int x=10; // integer x
    Char y='a'; // character c
    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x=x+y;
    // x is implicitly converted to float
    float z=x+1.0;
    count<<"x="<<x<<endl
         <<"y="<<y<<endl
         <<"z="<<z<<endl;
    return 0;
}
```

Output:

```
x = 107
y = a
z = 108
```

2. Explicit Type Conversion:

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:
Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

(type) expression

where type indicates the data type to which the final result is converted.

Example:

```
//C++ program to demonstrate
// explicit type casting
# include <iostream>
using namespace std;
int main()
{
    double x = 1.2;
    // Explicit conversion from double to int
    int sum = (int)x + 1;
    cout << "Sum =" << sum;
    return 0;
}
```

Output:

Sum = 2

Conversion using cast operator: A cast operator is an unary operator which forces one data type to be converted into another data type. C++ supports four types of casting:

1. Static cast
2. Dynamic cast
3. const cast
4. Reinterpret cast

Example:

```
# include <iostream>
using namespace std;
int main()
{
    float f = 3.5;
    // using cast operator
    int b = static_cast<int>(f);
    cout << b;
}
```

Output:

3

Advantages of Type conversion:

This is done to take advantage of certain features of type hierarchies or type representations.

It helps to compute expression containing variables of different data types.

classes and objects.

Class: A class in C++ is the building block, that leads to object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. for Example: consider the class of cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed limit, Mileage range etc., So here, car is the class and wheels, Speed limit, mileage are their

Properties.

A class is a user defined data-type which has data members and member functions

Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behaviour of the objects in a class.

In the above example of class car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.,

An object is an instance of a class. when a class is defined, no money is allocated but when it is instantiated (i.e. an object is created) money is allocated.

Defining class and Declaring Objects.

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at

the end.

```
Class ClassName
{Access Specifier: //can be private, public or protected
Data members;  //variables to be used
Member Function() {} //Methods to access data members
                     //class name ends with a semicolon
};
```

Declaring objects: When a class is defined, only the specification for the object is Defined; no memory or space storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Accessing data members and member functions.

The data members and member function of class can be accessed using the dot ('.') operator with the object.

For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName().

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in c++. There are three access modifiers: public, private and protected.

```
//c++ program to demonstrate
//accessing of data members
#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    // Access Specifier
    public:
    //Data Members
    String geekname;
```

```
    //Member Functions()
    void printname()
    {
       cout << "Geekname is: " << geekname;
    }
};
int main(){
    // Declare an object of class geeks
    Geeks obj1;
    // accessing data member
   obj1.geekname = "Abhi";
   // accessing member function
   obj1.printname();
   return 0;
}
```

Output:

```
Geekname is: Abhi
```

Member functions in classes

There are 2 ways to define a member function:

Inside class definition

Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
//C++ program to demonstrate function
// declaration outside class
# include < bits/stdc++.h>
using namespace std;
class Geeks
{
  public:
  String geekname;
  int id;
  // printname is not defined inside class definition
  void printname();
  // printid is defined inside class definition
  void printid()
  {
    cout << "Geek id is:" << id;
  }
};
```

```
// Definition of printname using scope resolution operator ::
void Geeks :: printname()
{
    cout << "Geekname is: " << geekname;
}
int main(){
    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id = 15;
  // call printname()
   obj1.printname();
   cout << endl;

   // call printid()
   obj1.printid();
   return 0;
}
```

Output

```
Geekname is: xyz
Geek id is: 15
```

Note that all the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere

during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note: Declaring a friend function is a way to give private access to a non-member function.

More about classes

1. Class name must start with an uppercase letter (Although this is not mandatory). If class name is made of more than one word, then first letter of each word must be in uppercase. Example,

Class Study, class StudyTonight etc.,

2. Classes contain, data members and member functions, and the access of these data members and variable depends on the access Specifiers.

3. Classes member functions can be defined inside the class definition or outside the class definition.

4. Class in C++ are Similar to Structures in C, the only difference being, class defaults to private access control, where as Structure defaults to public.

5. All the features of OOPs, revolve around classes in C++ inheritance, Encapsulation, Abstraction etc.,

6. Objects of class holds separate copies of data members. we can create as many objects of a class as we need.

7. Classes do posses more characteristics, like we can create abstract classes, immutable classes.

Objects of classes

Classes is mere a blueprint or a template. NO storage is assigned when we define a class. Objects are instance of class, which holds the data variables declared in class and the member functions work on these class objects.

Each object has different data variables. Objects are initialised using Special class functions called constructors. And whenever the object is out of its scope, another Special class member function called Destructor is called

, to release the memory reserved by the object. C++ doesn't have Automatic Garbage collector like in JAVA, in C++ Destructor performs this task.

```
class Abc
{
  int x;
  void display()
  {
    // some statement
  }
};
int main()
{
  Abc obj; // Object of class Abc created
}
```

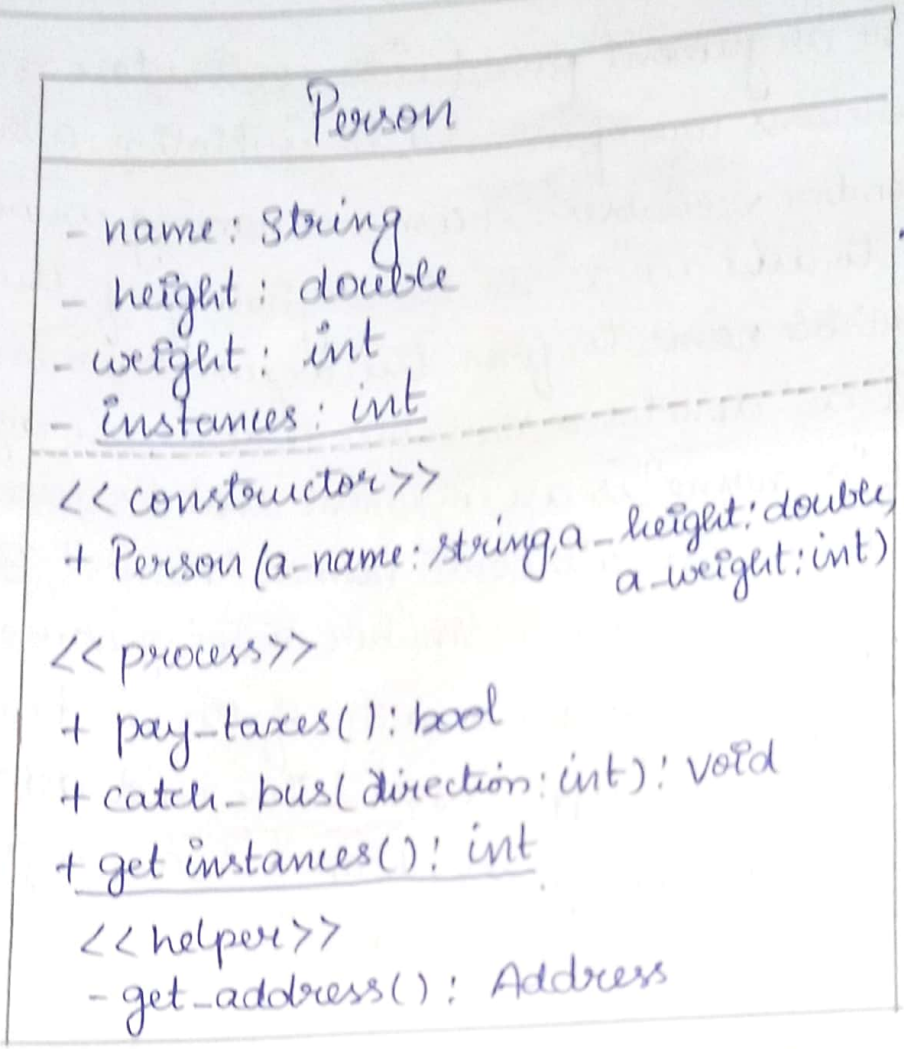
UML class diagram and its components.

"The Unified Modelling Language (UML) is a graphical language for visualizing, Specifying, constructing, and documenting the artifacts of a Software-intensive System". The UML is the result of the unification of the Booch Method, Rumbaugh's object Modelling Technique (OMT),

and Jacobson's Object-Oriented Software Engineering (OOSE). The UML currently consists of thirteen different diagrams consisting of a set of abstract symbols and rules that that represent the various elements of an object-oriented software system.

The UML Class Symbol

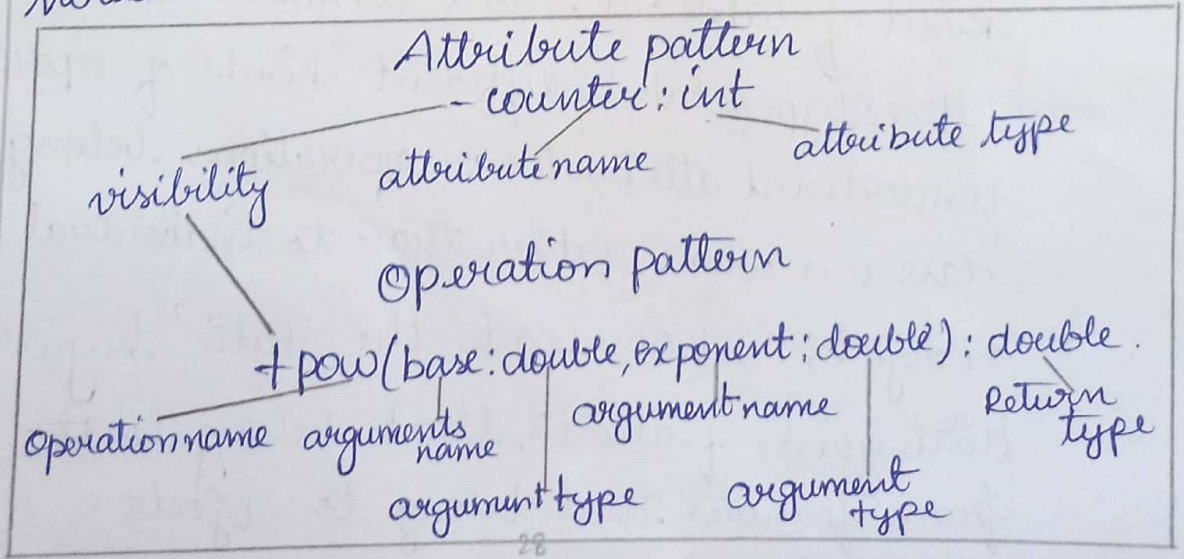
The UML class symbol is formed by a rectangle that is divided into three sections. The name of the class appears in the top section, the attributes are placed in the middle section, and the operations are placed in the bottom section. The individual class symbols are "semantically rich," meaning that they encode a great deal of information about each class. For example, the class name seems straightforward but the other two sections include some rather strange symbols. Fortunately, decoding those symbols a fairly straightforward. The following UML class diagram illustrates the UML class diagram symbols.



A UML class diagram is a rectangle divided into three sections. The << and >> symbols define an optional Stereotype that is a kind of label or short comment. Standard stereotypes label different kinds of operations. Underlined attributes or operations belong to the class as a whole rather than to individual instances or objects; C++ overloads the "static" keyword to distinguish features that belong to the class from features that belong to objects.

The argument passed into constructors and setter functions are often used to initialize attributes or member variables. A common naming conventing is to add an "a_" to the beginning of the attribute variable name to form the argument variable name. The "a_" denotes a variables that is an agruments, so, "a_name" is an argument that is used to initialize an attribute named "name." Some Java programmers use a similar naming convention: Change the first character of the instance field name to an upper case letter and add an "a" at the beginning. So, "aName" is an agrument used to initialize "Name".

The attributes and operations sections follow a rigid but fairly simple notation. The following illustration summarizes the notation as two patterns.



The UML notation for specifying attributes and operations. The notation is a well-defined language, which means that when coupled with a specific programming language, there is only one way (ignoring the order of features within the class) to translate the UML into a programming language. There is also only one way to translate code into a UML class. Going from UML to code is called forward engineering while going from code to UML is called reserve engineering.

Translating UML Attributes to C++ code

Translating UML attributes to C++ code is actually quite simple. A step-by-step algorithm and an example follow. The example is probaly the quickest and easiest way to see how to do the translation, but the algorithm contains some details that warrants at least one read through.

UML	C++
-name : String	private: String name;
- instances : int	private: Static int instances;

Mapping UML attributes to C++ member fields (aka member variables):

1. The visibility symbols appearing in a UML class diagram correspond to the C++ keywords:
   - "-" → "private:"
   - "+" → "public:"
   - "#" → "protected:"

2. Create one labeled visibility section in the C++ class for each visibility symbol appearing in the UML class attribute section. Each label consists of one the three keywords above followed by a ":"

3. Move the data type forward, and, if necessary, convert the type into a C++ data type or class name (UML diagrams are not language specific, so, for example, an attribute might be specified as "string" rather than "String"

or as "integer" rather than "int")

4. Write the name, discard the ":" and append a ";"

5. An underlined attribute is static; add the keyword "static" at the beginning of the line (static features are covered later).

Translating UML Behaviors to C++ code

Before reviewing the short process of converting UML operations into C++ member functions, recall that short functions my be defined in the class while long functions are only prototyped in the class. In either case the UML does not specify the contents of the function body.

Translating UML operations to C++ code is similar to translating attributes but does entail a few more steps. following the approach above, a step-by-step algorithm and an example both follow. Although the example is probably still the Quickest and easiest way to see how to do the translation, the algorithm does contains some details that you should know.

UML	C++
+ Person(a_name: string, a_height: double, a_weight: int)	Public: Person(string a_name, double a_height, int a_weight);
+ pay_taxes(): bool	public: bool pay_taxes();

+ catch_bus(direction: int): void	~~public~~ : private:
	static Address get_address();
	
	Public:
	void catch_bus(int direction);

- get_address(): Address	The prototype may be placed in a Separate "private" section (keeping variables and functions Separate) or placed in the private section with the member variables.

Mapping UML operations into C++ member functions:

1. The visibility symbols appearing in a UML class diagram correspond to the C++ keywords
   - "-" → "private:"
   - "+" → "public:"
   - "#" → "protected:"

2. Created a "public" section if one was not created previously. If needed, created a "protected" section. (It is often the case that only a "private" section is created while translating attributes and that a "public" section is created when translating operations)

3. As explained above, the UML is independent of any programming language (to the extent possible), which means that you can convert the UML diagram into any object-oriented language like C++, Java, or C#). If necessary, convert any types (return or arguments) into an appropriate C++ data type or class name

4. Move the return type from the right end of the UML operation to the beginning of the function prototype / definition and discard the trailing ":"

5. Copy the operation name to the function name

6. The function argument list is enclosed within parentheses and is a comma separated list of data type and argument name pairs. Convert each behaviour argument into a C++ function argument as described for attributes above

7. Either end the prototype with a ";" or define the body of a (short) C++ function between an opening "{" and a closing "}"

Abstraction and Encapsulation.

Abstraction

- Data abstraction means hiding of data.
- Abstraction is implemented automatically while writing the code in the form of class and object.

- It shows only important things to the user and hides the internal details.

Example: Program demonstrating Data Abstraction.

```
# include <iostream>
using namespace std;
class Addition
{ private: int a=10, b=10, c; // hidden data from outside world
  public:
    int add()
    {
      c=a+b;
      cout <<"Addition is: "<< c;
    }
};
int main()
{
  Addition a;
  a.add();
  return 0;
}
```

Output:

Addition is: 20

- In the above example, class Addition adds numbers together and returns the addition or sum. The public member add() function is the interface to the outside world and a user needs to know to use the class. The private member int a, b, c are something that the user does not need to know about, but is needed for the class to operate properly.

- Abstraction provides security for the data from the unauthorized methods and can be achieved by using class.

Encapsulation:

- Encapsulation is a process of bundling the data and functions in a single unit.

- It binds the data and functions together that manipulate the data and keeps them safe from outside interference and misuse.

- It is used to secure the data from other methods, when making a data private, these data are used within the class only and not

accessible outside the class.

```
// C++ program to explain Encapsulation
#include <iostream>
using namespace std;
class Encapsulation
{
  private:
        // data hidden from outside world
        int x;
  public:
        // function to set value of
        // variable x
        void set(int a)
        {
            x=a;
        }
        // function to return value of
        // variable x
        int get()
        {
            return x;
        }

};
// main function
int main()
```

```
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

Output:

5.

In the above program the variable x is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are binded together which is nothing but encapsulation.

Advantages of Encapsulation:

- Encapsulation provides abstraction between an object and its users.
- It protects an object from unwanted access by unauthorized users.

Difference between abstraction and encapsulation:

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it.	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. Abstraction: outer layout, used in terms of design. For Eg: outer look of a mobile phone, like it has a display screen & key pad buttons to dial a number.	4. Encapsulation: Inner layout, used in terms of implementation. For eg: Inner Implementation detail of a mobile phone, how keypad button and Display screen are connect with each other using circuits.

Applications of abstraction and Encapsulation

Abstraction: Abstraction is the act of representing essential information without including background details and Explainations.

Encapsulations: Encapsulations is the act of wrapping up of attributes (represented by data members) and operations (represented by functions) under one single unit (represented by class).

Taking Real world example.

Suppose you go to an automatic cola vending machine and request for a cola. The machine processes your request and gives the cola.

- Here automatic cola vending machine is a class. It contains both data i.e. cola can and operations. i.e. service mechanism and they are wrapped/Integrated under a single unit cola Vending Machine. This is called Encapsulation

- you need not know how the machine is working. is called Abstraction.
- you can interact with cola can only through service mechanism. You cannot access the details about internal data like how much cans it contains, mechanism etc. This is Data hiding.
- you cannot pick the can directly. You request for cola through proper instructions and request mechanism (i.e. by paying amount & and filling request) and get that cola only through specified channel. This is message passing.

The working and data is hidden from you. This is possible because that vending machine is made (or Encapsulated or Integrated) so. Thus, we can say Encapsulation is a way to implement Abstraction.

friend function

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows-

```
Class Box {
  double width;
 public;
    double length;
    friend void printWidth (Box Box);
    void setWidth (double wid);
};
```

To declare all members functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne –

friend class ClassTwo;

Consider the following program –

```
#include<iostream>
using namespace std;
class Box {
   double width;
 public;
   friend void printWidth (Box box);
   void setWidth (double wid);
};
// Member function definition
void Box::setWidth (double wid) {
    width = wid;
}
// Note: printWidth() is not a member function
void printWidth (Box box) {          of any class.
   /* Because printWidth() is a friend of Box, it can
   directly access any member of this class */
   cout <<"width of box:" << box.width << endl;
}
```

```
// main function for the program
int main() {
   Box box;

   // Set box width without member function
   box.setWidth (10.0);

   // use friend function to print the width.
   printWidth (box);

   return 0;
}
```

When the above code is complied and executed, it produces the following result-
width of box: 10

Inline function

C++ inline function is powerful concept that is commonly used with classes. If a function is inline, the complier places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline funetion could require all clients of the funetion to be

recomplied because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline Qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without use of the inline Specifier.

Following is an Example, which makes use of inline function to return max of two numbers.

```
# include <iostream>
using namespace std;
inline int Max(int x, int y)
{
  return (x>y)? x: y:
}
// main function for the program
int main()
{
```

```
    cout << "Max (20,10): " << Max(20,10) << endl;
    cout << "Max (0,200): " << Max(0,200) << endl;
    cout << "Max (100,1010): " << Max(100,1010) << endl;
    return 0;
}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

UML use case diagram

The purpose of a use case diagram in UML is to demonstrate the different ways that a user might interact with a system. Create a professional diagram for nearly any use case using our UML diagram tool.

In the Unified Modelling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To built one, you'll use a

set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system.

When to apply use case diagrams

A use case diagram doesn't go into a lot of detail— for example, don't expect it to model the order in which steps are performed. Instead, a proper use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems. Experts recommend that use case diagrams be used to supplement a more descriptive textual use case.

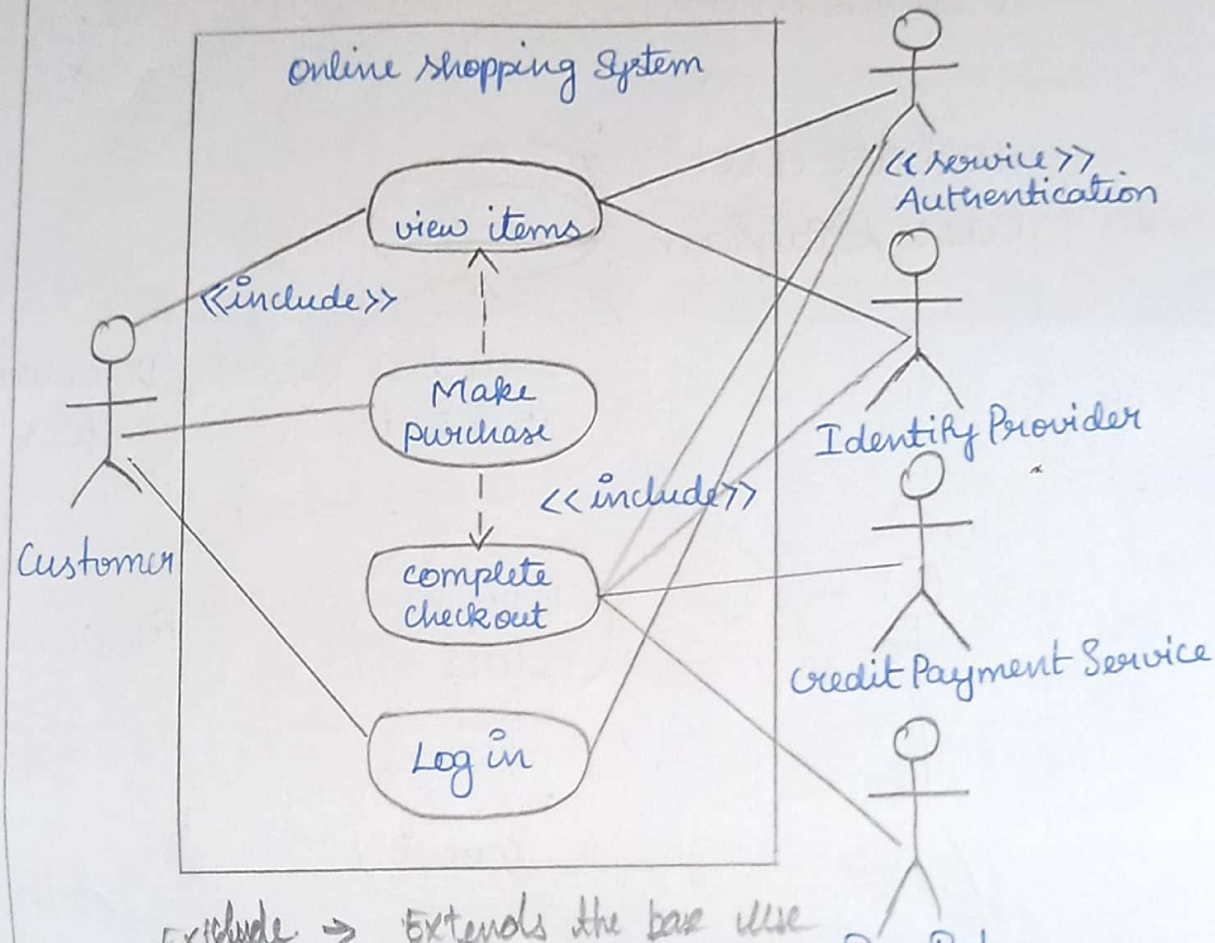
UML is the modelling toolkit that you can use to built your diagrams. Use cases are represented with a labeled oval shape. Stick figures represent

actors in the process. and the actor's participation in the system is modeled with a line between the actor and use case. To depict the system boundary, draw a box around the use case itself.

UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirement in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

include → used to extend the base use case and it must a condition.



Extend → extends the base use case, base use case can run each without calling the use case.

Use case diagram Components:

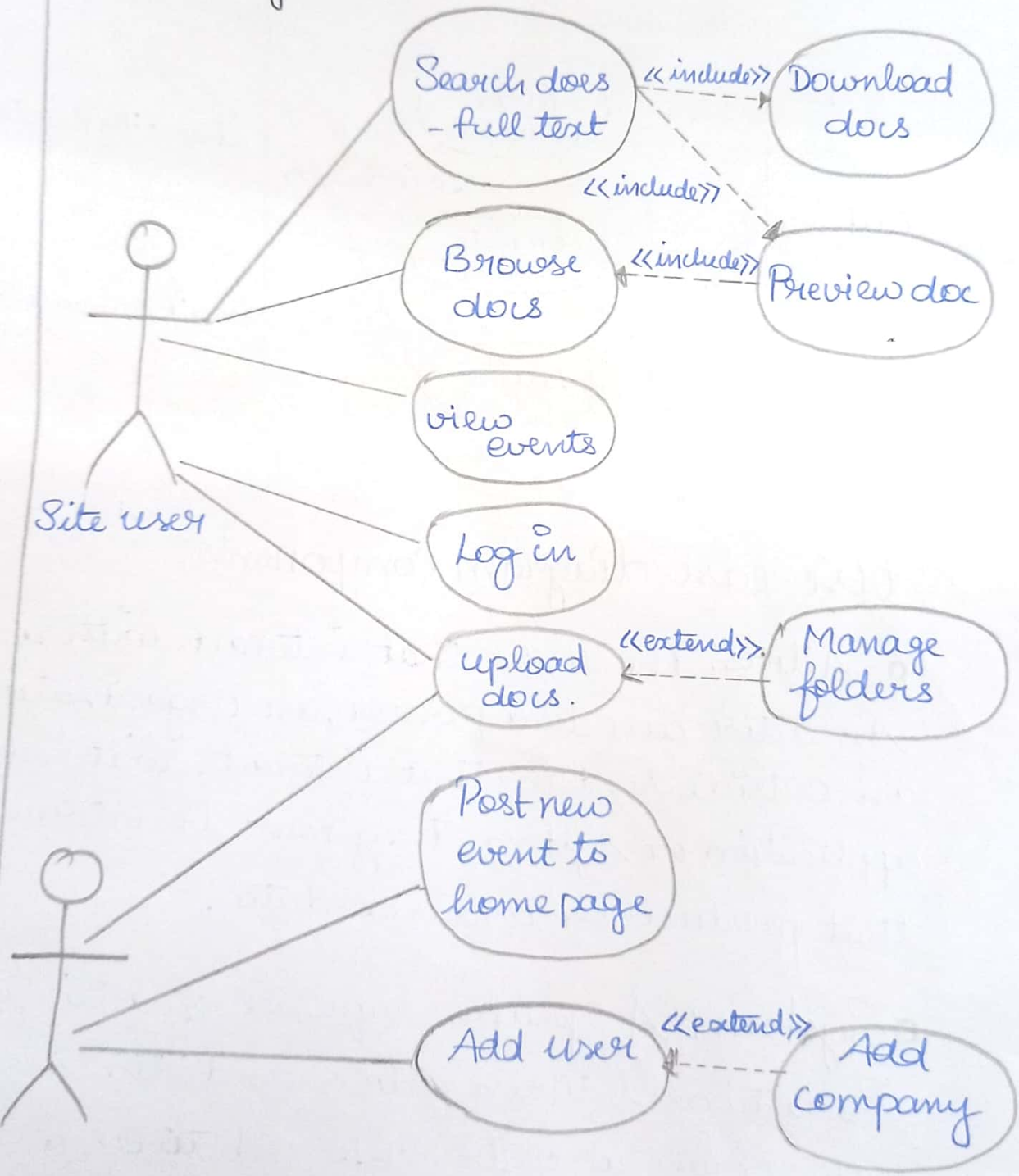
- **Actors:** The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.

- **System:** A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.

- **Goals:** The end result of most rule use cases. A successful diagram should describe the activities and

Variants and used to reach the goal.

website use case Diagram.



Use case diagram symbols and notation

The notation for a use case diagram is pretty straightforward and doesn't involve as many types of symbols as other UML diagrams.

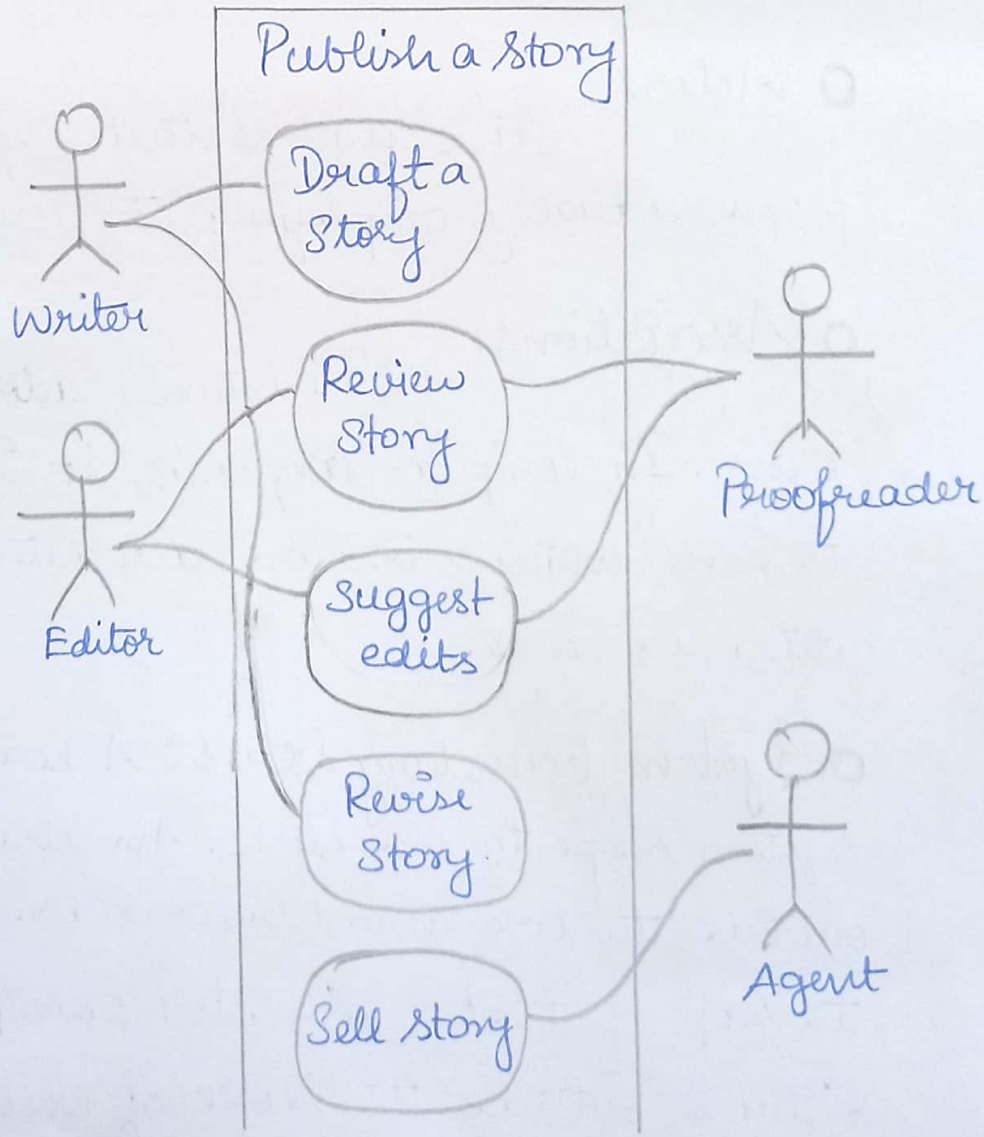
- **Use cases:** Horizontally shaped ovals that represent the different users that a user might have.
- **Actors:** Stick figures that represent the people actually employing the use cases.
- **Associations:** A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For example, Psycho killer is outside the scope of occupations in

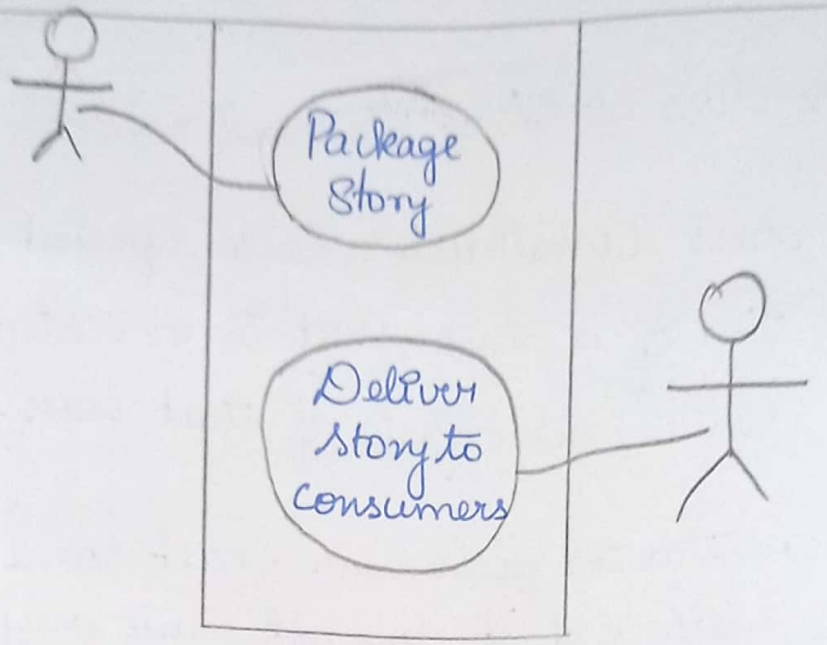
the chainsaw example found below.

- Packages: A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.

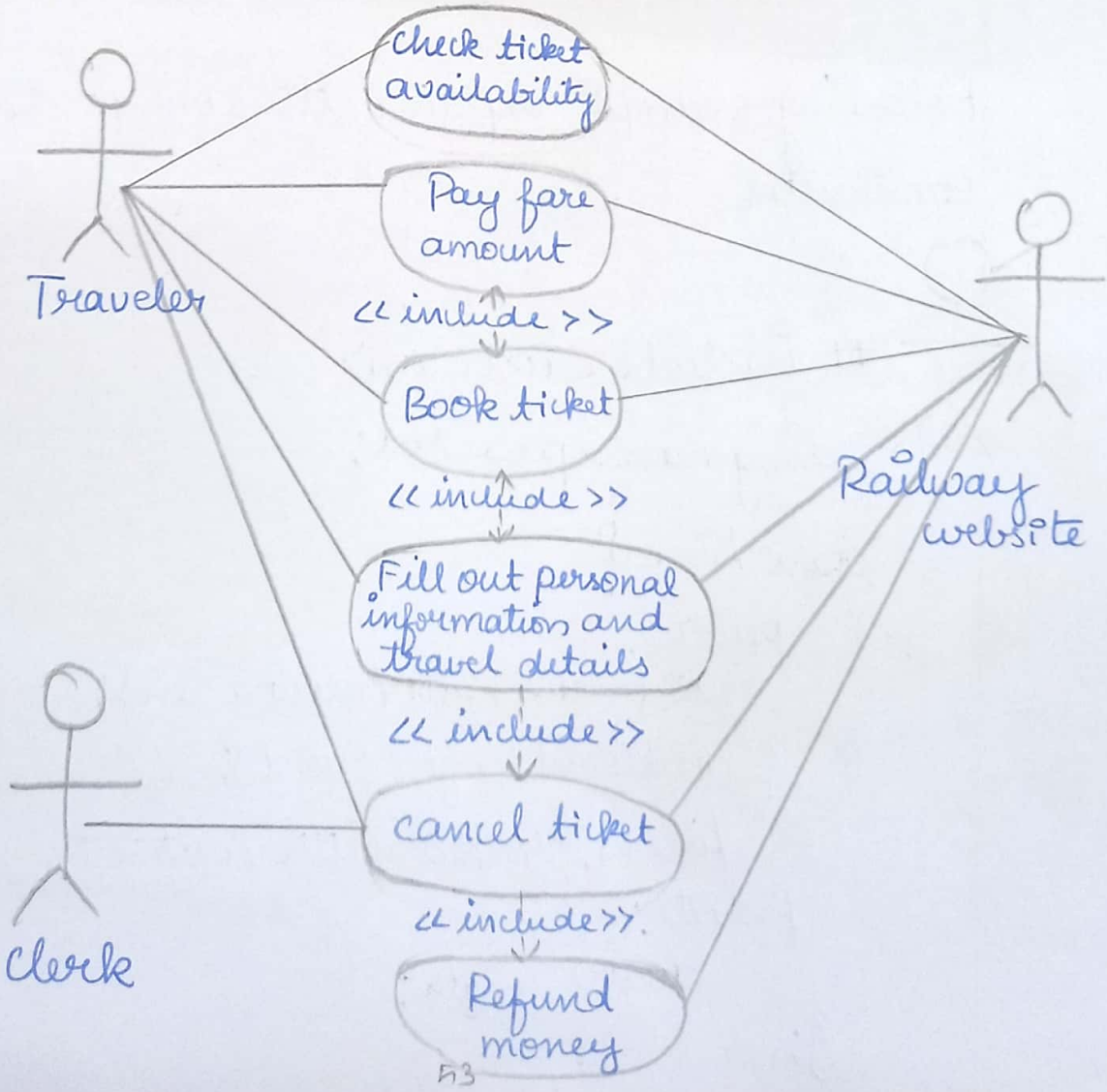
Use case diagram examples

Book publishing use case diagram example





Railway reservation use case diagram examples



The class constructor

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Following example explains the concept of constructor -

```
#include <iostream>
using namespace std;
class Line {
   public:
      void setLength (double len);
      double getLength (void);
      Line(); // This is the constructor
   Private:
      double length;
};
```

```
// member function definition including constructor.
Line::Line(void) {
    cout << "object is being created" << endl;
}
void Line::setLength(double len) {
    length = len;
}
double Line::getLength(void) {
    return length;
}
// Main function for the program
int main() {
    Line line;
        // set line length
        line.setLength(6.0);
        cout << "Length of line: " << line.getLength()
                                    << endl;
        return 0;
}
```

when the above code is compiled and executed, it produces the following result-

Object is being created

Length of line : 6

Parameterized Constructor

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This help you to assign initial value to an object at the time of its creation as shown in the following example—

```
#include <iostream>
using namespace std;
class Line{
   public:
     void setLength (double len);
     double getLength (void);
     Line (double len); // This is the constructor
   Private:
     double length;
};
// Member functions definition including
                              constructor
Line::. Line (double len){
    cout << "object is being created, length =
    length = len;                  " << len << endl;
}
 void Line:: setLength (double len) {
} length = len;
```

```
double Line::getLength(void){
    return length;
}

// Main function for the program
int main(){
   Line line(10.0);

   // get initially set length.
   cout << "Length of line:" << line.getLength()<<endl

   // set line length again
   line.setLength(6.0);
   cout << "Length of line:" << line.getLength()<<
                                        endl;
   return 0;
}
```

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

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization lists to Initialize Fields

In case of parameterized constructor, you can use following syntax to initialize the fields -

```
Line::Line(double len): length(len){
  cout << "Object is being created, length =
                                  " << leng << endl;
}
```

Above syntax is equal to the following syntax-

```
Line::Line(double len){
  cout << "Object is being created, length =" << len <<
                                                  endl;
  length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows -

```
C::C(double a, double b, double c): X(a), Y(b),
                                    Z(c){
  ....
}
```

The class Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a

pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following examples explains the concept of destructor –

```
#include <iostream>
using namespace std;
class Line {
    public:
        void setLength (double len);
        double getLength (void);
        Line(): // This is the constructor
                                declaration
~Line(); // This is the destructor declaration
private:
    double length;
};
```

```
// Member functions definition including
                                    constructor
Line::Line(void){
    cout << "object is being created" << endl;
}
Line::~Line(void){
    cout << "object is being deleted" << endl;
}
void Line::setLength(double len){
    length = len;
}
double Line::getLength(void){
   return length;
}
// Main function for the program
  int main(){
    Line line;
    // set line length
    line.setLength(6.0);
    cout << "Length of line:" << line.getLength
                                    ()<<endl;
    return 0;
}
```

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

Object is being created
Length of line: 6
Object is being deleted

Previous Page Print Page

Difference between Constructor and Destructor

S.NO.	Constructor	Destructor
1.	Constructor helps to initialize the object of a class.	whereas destructor is used to destroy the instances.
2.	It is declared as classname(arguments if any){constructor's Body}.	whereas it is Declared as ~classname(no arguments){};
3.	Constructor can either accept an arguments or not.	while it can't have any arguments,

4.	A constructor is called when an instance or object of a class is created.	It is called while object of the class is freed or deleted.
5.	Constructor is used to allocate the memory to an instance or object.	while it is used to deallocated the memory of an object of a class.
6.	Constructor can be overloaded.	while it can't be overloaded.
7.	The constructor's name is same as the class name.	Here, its name is also same as the class name proceded by tiled (~) operator.
8.	In a class, there can be multiple constructors.	while in a class, there is always a single destructor.
9.	There is a concept of copy constructor which is used to initialize a object from another object.	while here, there is no copy constructor concept.