Unit 2: Classes and Objects (8 hrs)

Specific Objectives:

- Implement the use of class, object, and method, Data Abstraction, Encapsulation, message passing, data hiding in C++.
- Understand and implement the concept of constructor, destructor, memory allocation and advanced functions in C++.

1. Introduction to C++: Origin of C++, Basic C++ Program Structure, Console Input/ output Streams and Manipulators
2. Structure in C and C++
3. Classes and Objects
4. Array of Objects
5. Class Diagram and Object Diagram
6. Access Specifiers and Visibility Mode
7. State and Behavior, Methods and Responsibilities
8. Implementation of Data Abstraction, Encapsulation, Message Passing and Data Hiding
9. Memory Allocations for Objects
10. Constructor: Default Constructor, Parameterized Constructor, Copy Constructor
11. Constructor Overloading
12. Destructors
13. Dynamic Memory Allocation: new and delete
14. Dynamic Constructor
15. Functions: Inline Function, Default argument, Passing and Returning Value, Pointer and Reference, Static Data Member and Static Member Function
16. Friend Function and Friend Class

1. Introduction to C++: Origin of C++, Basic C++ Program Structure, Console Input/ output Streams and Manipulators

Introduction to C++:

C++ is a powerful and versatile programming language that was developed as an extension of the C programming language. It was created by Bjarne Stroustrup ***(B-yar-ne Strov-stroop)*** in the early 1980s and has since become one of the most widely used programming languages for various applications.

C++ is known for its efficiency, performance, and ability to handle low-level programming tasks while also providing high-level abstractions. It combines both procedural and object-oriented programming paradigms, allowing developers to write efficient and structured code.

Here are some key features and concepts of C++:

1. **Object-Oriented Programming (OOP**): C++ supports the principles of OOP, including encapsulation, inheritance, and polymorphism. This allows you to organize your code into reusable objects and classes, making it easier to manage and maintain.
2. **Standard Template Library (STL):** The STL is a collection of template classes and functions that provide common data structures (such as vectors, lists, and maps) and algorithms (such as sorting and searching) in a generic and reusable way. It simplifies the development process by offering pre-implemented components.
3. **Templates:** C++ supports templates, which enable you to write generic code that can work with different data types. Templates allow you to create reusable functions and classes that can be customized for specific data types at compile time.
4. **Exception Handling:** C++ provides a robust exception handling mechanism that allows you to handle and recover from runtime errors gracefully. With exception handling, you can catch and handle exceptions, preventing your program from crashing.
5. **Performance:** C++ is known for its efficiency and performance. It provides low-level control over system resources and allows you to write code that executes quickly and uses system resources efficiently. This makes C++ a popular choice for system-level programming, game development, and other performance-critical applications.
6. **Portability:** C++ is a standardized language, and compilers are available on various platforms. This makes it highly portable, allowing you to write code that can run on different operating systems and hardware architectures with minimal modifications.

Origin of C++:

C++ is a general-purpose programming language that was developed as an extension of the C programming language. Its origins can be traced back to the early 1980s when a Danish computer scientist named Bjarne Stroustrup started working on creating a language that would enhance the capabilities of C for large-scale software development.

Stroustrup's motivation behind creating C++ was to add object-oriented programming (OOP) features to C, while retaining the efficiency and low-level control of C. He wanted to develop a language that could support both high-level abstractions and low-level operations, making it suitable for a wide range of applications.

The name "C++" was chosen by Stroustrup as an incremental notation representing the addition of OOP concepts to the C programming language. In C++, the "++" operator is used to increment a value by 1, so the name symbolizes the extension of C.

The development of C++ started in 1979, and the language underwent several iterations and refinements over the years. The first commercial implementation of C++ was released in 1985. In 1998, the International Organization for Standardization (ISO) standardized the language, resulting in the ISO/IEC 14882:1998 standard, often referred to as "C++98."

ISO/IEC: ***International Organization for Standardization/International Electrotechnical Commission***

Since then, the C++ language has continued to evolve, with new standards being released periodically. Notable updates include C++11 (2011), C++14 (2014), C++17 (2017), and C++20 (2020). These updates introduced various new features, improvements, and libraries to enhance the language and make it more expressive, efficient, and safer.

C++23 is the next ISO/IEC 14882 standard for the C++ programming language. Slated for release in December 2023, it will replace the previous C++20 standard (C++ standards are on a fixed three-year release cycle).

C++ has become one of the most widely used programming languages and finds applications in various domains such as systems programming, game development, embedded systems, scientific computing, and more. Its flexibility, performance, and extensive standard libraries have contributed to its popularity among developers.

Basic C++ Program Structure:

A basic C++ program consists of several key elements that define its structure. Here's a breakdown of the essential components:

1. **Preprocessor Directives:**
   - Preprocessor directives, starting with a hash symbol (#), are used to include header files or perform other pre-compilation tasks. Common directives include **#include** for including header files and **#define** for defining constants.
2. **Main Function:**
   - Every C++ program must have a **main()** function, which serves as the entry point of the program. Execution starts from this function.
3. **Function Declarations and Definitions:**
   - You can define additional functions before or after the **main()** function. Function declarations specify the function's name, return type, and parameters. Definitions provide the implementation of the functions.
4. **Statements and Expressions:**
   - Inside functions, you write statements and expressions to perform specific actions. Statements are individual instructions or commands, while expressions produce a value when evaluated.
5. **Variables and Data Types:**
   - Variables hold data during program execution. You declare variables by specifying their type and name. C++ supports various built-in data types such as **int**, **float**, **char**, **bool**, etc.
6. **Control Flow:**
   - Control flow statements determine the order of execution in your program. Common control flow statements include **if**, **else**, **switch**, **while**, **for**, etc. They allow you to make decisions and repeat blocks of code based on specific conditions.
7. **Input and Output:**
   - C++ provides the standard input/output streams, **cin** and **cout**, for reading input from the user and displaying output to the screen, respectively. You can use the **extraction (>>)** and **insertion (<<)** operators with these streams.

8. Return Statement:

- At the end of the **main()** function, you typically include a **return** statement to indicate the termination of the program. The value returned by **main()** is used as the exit status of the program.

Here's a simple example illustrating the basic structure of a C++ program:

```
#include <iostream>

// Function declaration
void greet();

// Main function
int main() {
    // Function call
    greet();

    // Print a message
    std::cout << "Hello, World!" << std::endl;

    // Return statement
    return 0;
}

// Function definition
void greet() {
    std::cout << "Greetings!" << std::endl;
}
```

Console Input/ output Streams:

In C++, the console input/output streams provide a way to interact with the user through the console or command line. The standard input stream (`cin`) is used for reading input from the user, while the standard output stream (`cout`) is used for displaying output to the console. Here's an overview of how to use these streams:

1. Output using `cout`:

- You can display text and values to the console using the `cout` object from the `iostream` library. Here's an example:

```
#include <iostream>

  int main() {
      int num = 42;
      std::cout << "The value of num is: " << num << std::endl;
      return 0;
  }
```

The output will be: `The value of num is: 42`. The `<<` operator is used to insert values into the `cout` stream.

2. Input using `cin`:

- You can read user input from the console using the `cin` object from the `iostream` library. Here's an example:

```
#include <iostream>

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    std::cout << "You entered: " << num << std::endl;
    return 0;
}
```

In this example, the program prompts the user to enter a number, and the input is stored in the variable `num` using the `>>` extraction operator.

3. String input using `getline`:

- If you want to read a whole line of text instead of a single word or number, you can use the `getline()` function from the `string` library. Here's an example:

```
#include <iostream>
#include <string>

int main() {
    std::string name;
    std::cout << "Enter your name: ";
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

The `getline()` function reads a line of input until it encounters a newline character (`'\n'`), and stores it in the `name` variable.

4. Formatting output:

- You can format the output using various **manipulators** and modifiers. For example, you can control the number of decimal places for floating-point numbers or specify the width of output fields. Here's an example:

```
#include <iostream>
#include <iomanip>

int main() {
    double pi = 3.14159;
    std::cout << std::setprecision(4) << pi << std::endl;
    std::cout << std::setw(10) << std::setfill('*') << pi << std::endl;
    return 0;
}
```

The `setprecision()` manipulator sets the number of decimal places, and `setw()` with `setfill()` sets the width and fill character for the output field.

These are some basic operations for console input and output in C++. The `cin` and `cout` objects are widely used for interactive programs, data input, and output formatting. Remember to include the `<iostream>` header to work with these streams.

Manipulators:

Manipulators are helping functions in C++ that are used to modify the input/output stream. What it means, it will not modify the value of a variable, it will only modify the streams or formatting streams using the insertion (<<) and extraction (>>) operators.

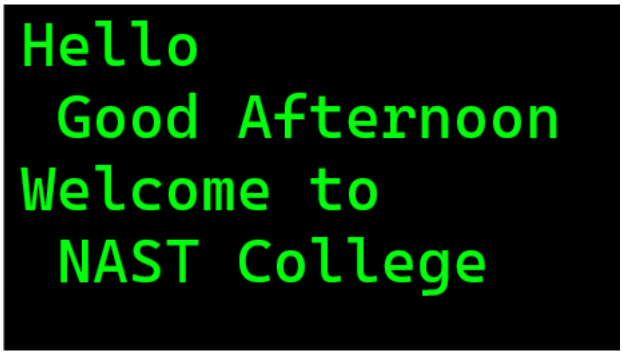
1. Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.
2. Manipulators are operators that are used to format the data display.
3. To access manipulators, the file **iomanip** should be included in the program. **#include<iomanip>**

Manipulators are used for enhancing streams or formatting streams. For writing the data, we can adopt some formats. For example, a common manipulator that we used is the **endl** that is used for the endline. Instead of endl, we can also say that **cout** << **"\n";** This will also print a new line. So, endl is a manipulator which is used for formatting stream. So, it is useful for formatting output streams.

Example to understand \n Manipulator in C++:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << "Hello \n Good Afternoon\n";
cout << "Welcome to \n NAST College\n\n";
return 0;
}
```

Output:



```
Hello
 Good Afternoon
Welcome to
 NAST College
```

Integer Manipulators in C++:

Now let us see what are other manipulators available. There are some manipulators available for data types like integer and float. For integer type data, we have manipulators,

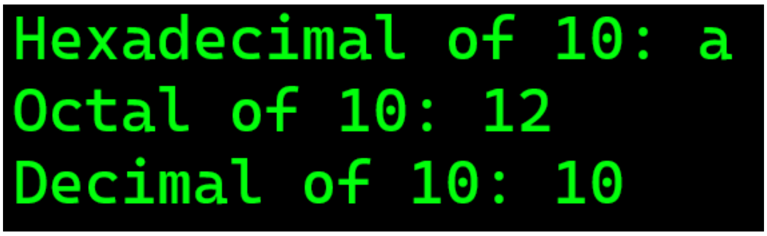
1. **hex** – it will display the data in hexadecimal.
2. **oct** – it will display data in the octal form.
3. **dec** – to display data in decimal form.

For example, if we say **cout << hex << 163;**

The output of the above statement will be **A3**. The hexadecimal form of 163 is A3. So, we can mention the manipulator then the output will be in that form. So, all the integers will be in the hexadecimal form that is written after the manipulator. And if you want to change the number system then you have to mention decimal, octal anything that you want. For a better understanding, please have a look at the below example.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << "Hexadecimal of 10: " << hex << 10 <<"\n";
cout << "Octal of 10: " << oct << 10 <<"\n";
cout << "Decimal of 10: " << dec << 10 <<endl<<endl;return 0;
}
```

Output:



```
Hexadecimal of 10: a
Octal of 10: 12
Decimal of 10: 10
```

Float Manipulators in C++:

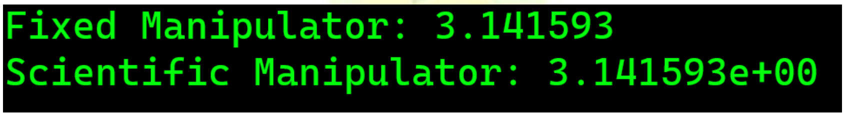
Now, similarly, for floating points, we have manipulators,

1. **Fixed**: It will show in the fixed floating-point number. For example, **cout << fixed << 3.14159265358979;** Then it prints 6 digits after decimal number i.e. 3.141593 will be displayed.
2. **scientific:** It will display the number in scientific form or exponent form. For example, **cout << fixed << 3.14159265358979;** Then the scientific form or exponent form of this number i.e. 3.141593e+00 will be displayed.

For a better understanding, please have a look at the below example.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << "Fixed Manipulator: " << fixed << 3.14159265358979<<endl;
cout << "Scientific Manipulator: " <<scientific << 3.14159265358979<<"\n";
return 0;
}
```

Output:



```
Fixed Manipulator: 3.141593
Scientific Manipulator: 3.141593e+00
```

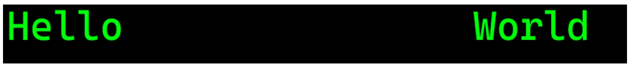
There are other manipulators also available. They are as follows:

setw – It will set some amount of space for displaying the data. For example,

cout << "Hello" << setw(10) << "World";

This "World" will be displayed in 10 spaces. Though the number of the alphabet is only 5, it will be shown in 10 places. For a better understanding, please have a look at the below example.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << "Hello" << setw(10) << "World";
return 0;
}
```



```
Hello          World
```

2. Structure in C and C++

In C and C++, the structure is a user-defined composite data type that allows you to group together different variables of different types into a single unit. It is often used to represent a collection of related data elements.

Here's the syntax for defining a structure in C and C++:

```
struct MyStruct {
    // Member variables or fields
    dataType1 member1;
    dataType2 member2;
    // ...
    dataTypeN memberN;
};
struct MyStruct Variable_Name;
```

In this syntax:

- **struct** is the keyword used to define a structure.
- **MyStruct** is the name given to the structure type. You can choose any valid identifier as the structure name.
- **dataType1**, **dataType2**, ..., **dataTypeN** represent the data types of the member variables. These can be any valid C/C++ data types.
- **member1**, **member2**, ..., **memberN** are the names of the member variables. You can choose any valid identifier as the variable name.
- **Variable_Name** represent the MyStruct structure variable, that will now be able to store the member1, member2, ……. of a variable_name.

Once you have created a variable of structure, you can access its members using the dot operator (.).

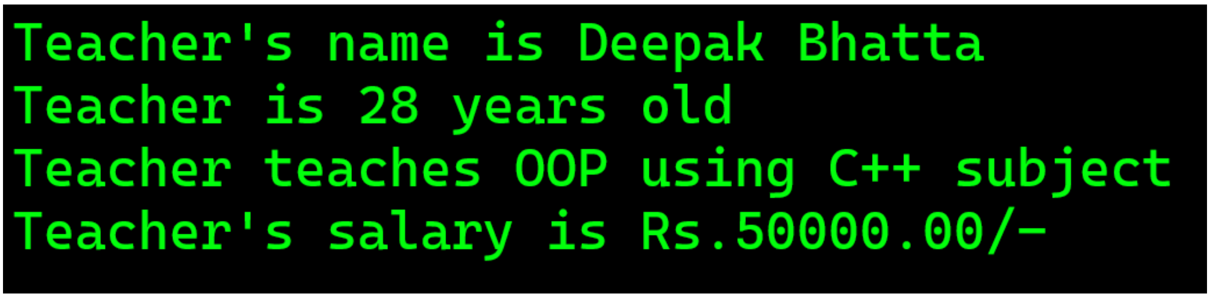
Here are some of the differences between C structures and C++ structures:

- C structures cannot have member functions, while C++ structures can.
- C structures cannot have default values for their members, while C++ structures can.
- C structures cannot have static members, while C++ structures can.
- C structures cannot have constructors or destructors, while C++ structures can.
- C structures cannot have access modifiers, while C++ structures can.

Example: Structure using C

```c
#include <stdio.h>

struct Teacher {
  char *name;
  int age;
  char *subject;
  float salary;
};

int main() {
  struct Teacher t;

  t.name = "Deepak Bhatta";
  t.age = 28;
  t.subject = "OOP using C++";
  t.salary = 50000.00;

  printf("Teacher's name is %s\n", t.name);
  printf("Teacher is %d years old\n", t.age);
  printf("Teacher teaches %s subject\n", t.subject);
  printf("Teacher's salary is Rs.%.2f/-\n", t.salary);

  return 0;
}
```

Output:

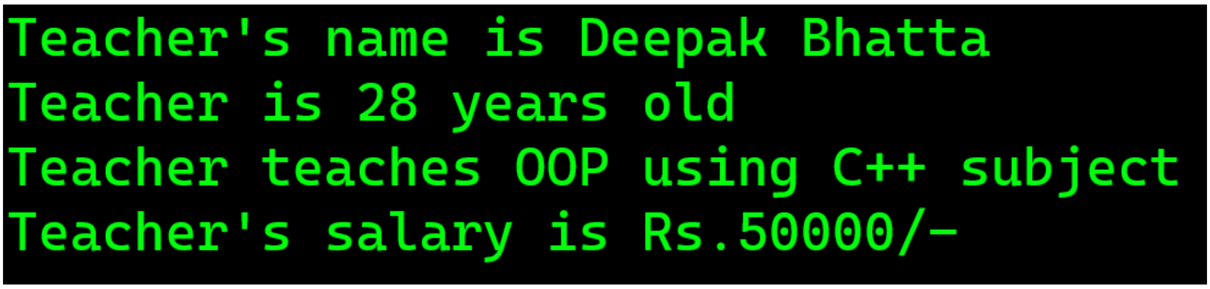


```
Teacher's name is Deepak Bhatta
Teacher is 28 years old
Teacher teaches OOP using C++ subject
Teacher's salary is Rs.50000.00/-
```

Example: Structure using C++

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

struct Teacher {
  string name;
  int age;
  string subject;
  float salary;
};

int main() {
  Teacher t;

  t.name = "Deepak Bhatta";
  t.age = 28;
  t.subject = "OOP using C++";
  t.salary = 50000.00;

  cout << "Teacher's name is " << t.name << endl;
  cout << "Teacher is "<<t.age << " years old" <<endl;
  cout << "Teacher teaches " << t.subject << " subject" <<endl;
  cout << "Teacher's salary is Rs." << t.salary << "/-" <<endl;

  return 0;
}
```

Output:



```
Teacher's name is Deepak Bhatta
Teacher is 28 years old
Teacher teaches OOP using C++ subject
Teacher's salary is Rs.50000/-
```

Extensions to Structures:

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. Inheritance, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

student A; // C++ declaration

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as class. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

Note: ***The only difference between a structure and a class in C++ is that, by default, the members of a class are private while, by default, the members of a structure are public.***

3. Classes and Objects

Specifying a Class:

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type the can be treated like any other built-in data type. Generally, a class specification has two parts:

1. **Class declaration**
2. **Class function definitions**

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.

The general form of a class declaration is:

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

The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called class *members*.

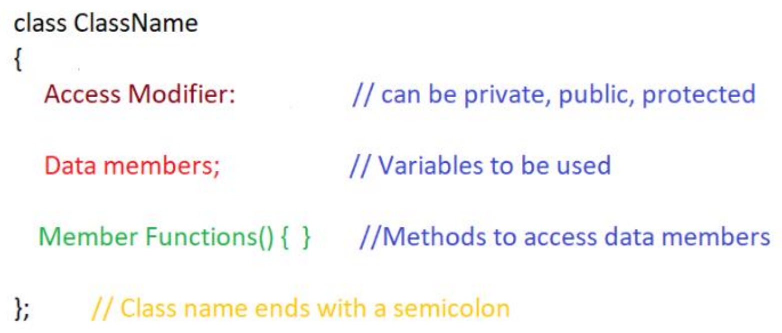
They are usually grouped under two sections, namely, ***private** and **public*** to denote which of the members are private and which of them are public.

The keywords private and public are known as visibility labels. ***Note that these keywords are followed by a colon (:).***

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

The variables declared inside the class are known as data members and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class.

The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.



```
class ClassName
{
   Access Modifier:          // can be private, public, protected

   Data members;             // Variables to be used

   Member Functions() { }    //Methods to access data members

};      // Class name ends with a semicolon
```

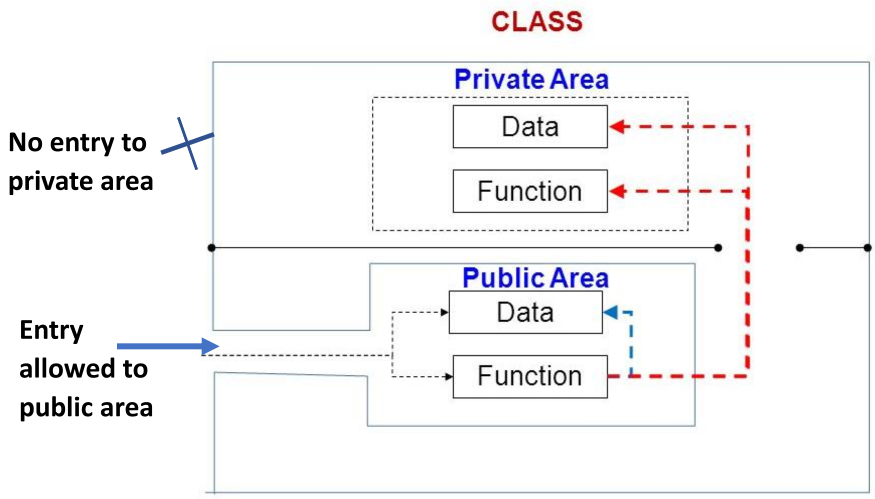


Fig: Data Hiding in Classes

A simple Class Example:

A typical class declaration would look like:

```
class Employee
{
      string empName;
      int empId;
      double empSalary;
  public:
      void getData(string name, int id, double salary);
      void putData(void)

};
```

Variables declaration
private by default

Function declaration
using prototype ends
with semicolon ;

Creating Objects:

The declaration of Employee class does not define any objects of Employee but only specifies what they will contain. Once a class has been declared, we can create variables of that type by using the class name.

Example:

Employee emp;

Create a variable **emp** of type **Employee.** In C++, the ***class variables are known as objects.*** Therefore, emp is called an object of type Employee. We may also declare more than one object in one statement.

Example:

Employee emp1, emp2, emp3;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a template and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures.

That is to say, the definition

```
class Employee
{
        ………
        ………
        ………
}emp1, emp2, emp3;
```

would create the objects emp1, emp2 and emp3 of type Employee.

Accessing Class Members:

Note that, The private data of class can be accessed only through the member functions of that class. The main() cannot contain statements that access empName, empId, empSalary directly.

Syntax for calling a member function:

```
object_name.function_name(actual_arguments);
```

For example, the function call statement

```
emp.getData("Deepak Bhatta", 11, 50000.60);
```

This assigns the value Deepak Bhatta to empName, 11 to empId and 50000.00 to empSalary of the object emp by implementing the getData() function.

Similarly, the statement

```
emp.putData();
```

it displays the values of data members. Remember, a member function can be invoked only by using an object (of the same class). for example **getData("Deepak Bhatta", 11, 50000.00);** has no any meaning.

emp.putData(); sends a message to object emp requesting it to display its contents.

Defining a Member functions:

Member functions can be defined in two places:

1. Outside the class definition
2. Inside the class definition

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body Since C++ does not support the old version of function definition, the ANSI prototype form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header This 'label' tells the compiler which class the function belongs to.

The general form of a member function definition is:

```
Return_type class_name function_name (argument declaration)
{
      Function body
}
```

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

For instance, consider the member functions **getData()** and **putData()** as discussed above.

```
void Employee :: getData(string name, int id, double salary)
{
      empName = name;
      empId = id;
      empSalary = salary;
}
void Employee :: putData(void)
{
      cout << “Employee ID: “ << empId << endl;
      cout << “Employee Name: “ << empName << endl;
      cout << “Employee Salary: “ << empSalary << endl;
}
```

Since these functions do not return any value, their **return_type** is void. Function arguments are declared using the ANSI prototype.

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

These characteristics are

- Several different classes can use the same function name. The **'membership label'** will resolve their scope
- Member functions can access the private data of the class. A nonmember function cannot do so *(However, an exception to this rule is a **friend function**)*
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition:

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

```
class Employee
{
      string empName;
      int empId;
      double empSalary;
  public:
      void getData(string name, int id, double salary); // declaration
      //inline function
      void putData(void)      // definition inside the class
            {
            cout << “Employee ID: “ << empId << endl;
            cout << “Employee Name: “ << empName << endl;
            cout << “Employee Salary: “ << empSalary << endl;
            }
};
```

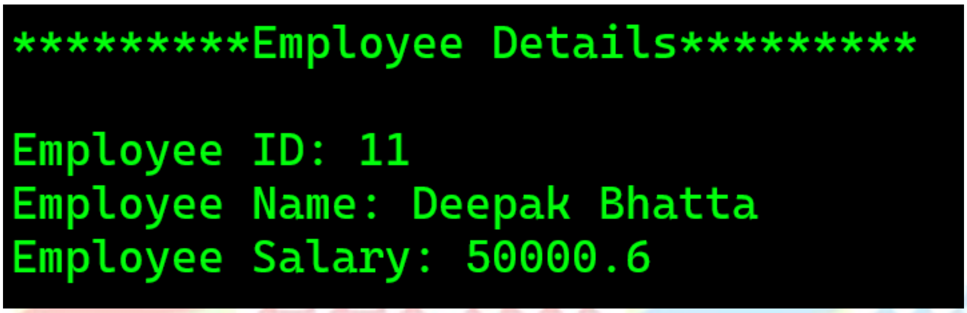
When a function is defined inside a class, it is treated as an **inline function**. Therefore, all the restrictions and limitations that apply to an inline function are applicable here. ***Normally, only small functions are defined inside the class definition.***

Example: Class Implementation:

```
#include<iostream>
using namespace std;
class Employee
{
    string empName; //private by default
    int empId; //private by default
    double empSalary; //private by default

public:
    void getData(string name, int id, double salary); // prototype
declaration

    // function defined inside class
    void putData(void)
    {
        cout << "Employee ID: " << empId << endl;
        cout << "Employee Name: " << empName << endl;
        cout << "Employee Salary: " << empSalary << endl;
    }

};
//*************** Member Function Definition ****************

void Employee :: getData(string name, int id, double salary) // Use
Membership Label
{
    empId = id; // Private variables directly used
    empName = name; // Private variables directly used
    empSalary = salary; // Private variables directly used
}
```

```
//************** Main Program *******************
int main()
{
    Employee emp1;  // Create object emp1
    cout << "\n*********Employee Details*********" << endl << endl;
    emp1.getData("Deepak Bhatta",11, 50000.60); // Call Member function
    emp1.putData(); // Call Member function

    return 0;
}
```

Output:



```
*********Employee Details*********

Employee ID: 11
Employee Name: Deepak Bhatta
Employee Salary: 50000.6
```

This program features the class Employee. This class contains three private variables and two public functions. The member functions **getData()** which has been defined outside the class supplied value to both the variables.

```
    empId = id; // Private variables directly used
    empName = name; // Private variables directly used
    empSalary = salary; // Private variables directly used
```

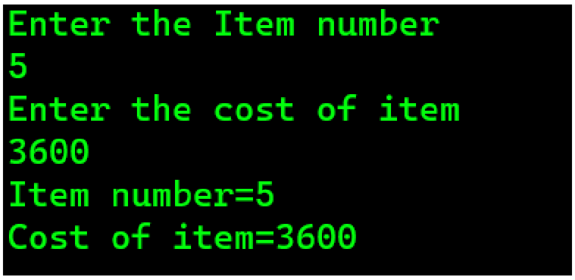
written in the function definition of **getData().** This shows that the member functions can have direct access to private data items.

The member function **putData()** has been defined inside the class and therefore behaves like an inline function. This function displays the value of private variables **empId, empName, empSalary**. The program creates one object emp1. **Employee emp1**

Simple Example 1: Inside the class definition:

```
#include<iostream>

using namespace std;
class Item {
private:

    int number;
    float cost;
public:
    void getdata() {
    cout << "Enter the Item number" << endl;
    cin >> number;
    cout << "Enter the cost of item" << endl;
    cin >> cost;
  }
  void display() {
    cout << "Item number=" << number << endl;
    cout << "Cost of item=" << cost << endl;
  }
};

int main() {
  Item x;
  x.getdata();
  x.display();
  return 0;
}
```

Output:

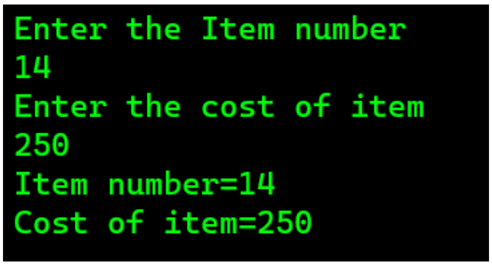


```
Enter the Item number
5
Enter the cost of item
3600
Item number=5
Cost of item=3600
```

Simple Example 2: Outside the class definition:

```
#include<iostream>

using namespace std;
class Item {
  private: int number;
  float cost;
  public: void getdata();
  void display();
};
void Item::getdata() {
  cout << "Enter the Item number" << endl;
  cin >> number;
  cout << "Enter the cost of item" << endl;
  cin >> cost;
}
void Item::display() {
  cout << "Item number=" << number << endl;
  cout << "Cost of item=" << cost << endl;
}
int main() {
  Item x;
  x.getdata();
  x.display();
  return 0;
}
```

Output:



```
Enter the Item number
14
Enter the cost of item
250
Item number=14
Cost of item=250
```

Inside Class Definition	Outside Class Definition	Inside Class Definition
Access Specifiers	Members can be declared as public, private, or protected within the class.	Access specifiers are not used within the class definition.
Scope	Members have access to all other members of the class, including private members.	Members have access to each other based on their access specifiers (e.g., public members can be accessed by all, private members only within the class).
Definition Syntax	Members are defined directly inside the class using their regular syntax.	Members are declared inside the class using their regular syntax and defined outside the class using the scope resolution operator (::).
Usage	Useful for small and simple classes, as it keeps the class definition concise.	Useful for complex classes or when the class definition needs to be separated into multiple files.
Separation of Code	Implementation code (member functions) is mixed with the class definition.	Implementation code is separated from the class definition, making it easier to manage and understand.
Inline Functions	Member functions declared and defined inside the class are automatically considered as inline functions.	Member functions can be defined as inline functions using the inline keyword, regardless of the class definition location.
Friend Declarations	Friend declarations can be made within the class to grant access to private members to other classes or functions.	Friend declarations can be made outside the class, allowing access to private members of a class from specific functions or classes.
Encapsulation	Encapsulation can be achieved by declaring members as private and providing public member functions to access and modify those private members.	Encapsulation is achieved using the access specifiers and appropriate member function declarations.

Inside Class Definition:

Advantages:

1. **Conciseness:** Keeps the class definition concise and self-contained.
2. **Accessibility:** Provides easy access to all members, including private ones.
3. **Simplicity:** Suitable for small and simple classes, making the code easier to read and understand.

Disadvantages:

1. **Readability and Maintainability:** Can become harder to read and maintain as the class grows in size.
2. **Compiling Efficiency:** May result in increased compilation time if the class has inline member functions.
3. **Separation of Concerns:** Mixes the declaration and implementation code, potentially hindering separation of concerns.

Outside Class Definition:

Advantages:

1. **Separation of Concerns:** Allows for clear separation between the class declaration and implementation code, improving code organization and maintainability.
2. **Encapsulation:** Encourages encapsulation by explicitly specifying access specifiers and providing public member functions for accessing and modifying private members.
3. **Reusability:** Enables easy reuse of the class declaration in multiple files or projects, promoting code modularity.

Disadvantages:

1. **Code Scattering:** The class declaration and member function definitions are spread across multiple files, making it more challenging to navigate and understand the implementation.
2. **Access Control:** Relies on access specifiers, which can lead to accidental or unintended access if not used consistently or properly.
3. **Increased Complexity:** Requires managing and organizing multiple files, which can introduce additional complexity.

4. Array of Objects

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of object. As we know, an array is a collection of similar date type, we can also have arrays of variables of type class. Such variables are called array of object. First, we define a class and then array of objects are declared. An array of objects is stored inside the memory in the same way as multidimensional array.

Array of object is used to create an array of similar type of object, we can also create different categories of employee.

```
Employee hod[3]; // array of hod
Employee accountant[2]; // array of accountant
Employee faculty[25]; // array of faculty
```

The array **hod** contains three objects, namely, **hod[0], hod[1], hod[2]** of type **Employee** class. Similarly, the **accountant** array contains 2 objects and the **faculty** array contains 25 objects. An array of objects stored inside the memory in the same way as a multi-dimensional array. The array **hod** is represented in the below figure. *Note that only space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.*

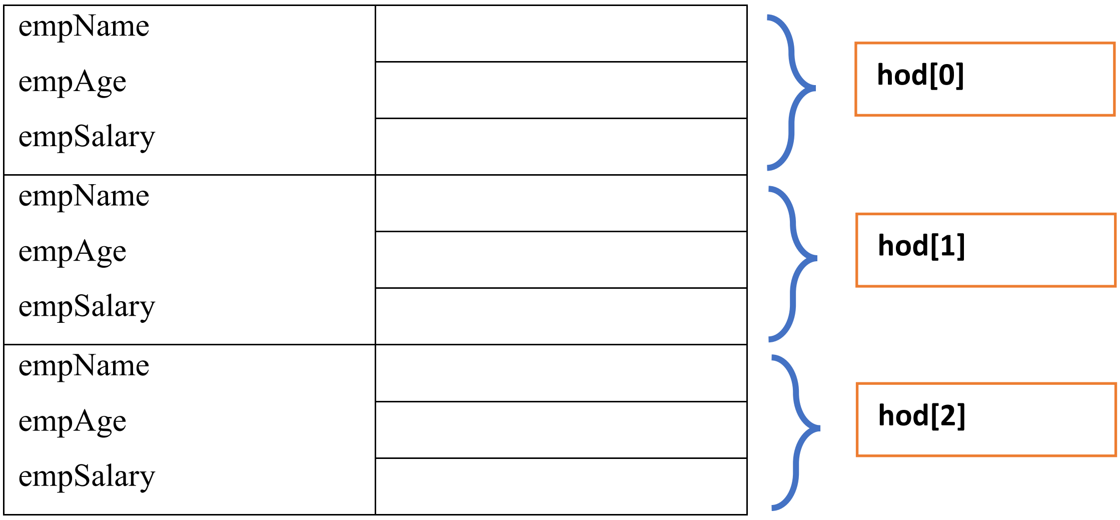


Fig: Storage of data items of an object array

Example: Array of objects

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

class Employee
{
    string name;
    int age;
    float salary;
public:
    void getData();
    void display();
};
void Employee::getData()
{
    fflush(stdin);
    cout<<"Enter Employee Name:";
    getline(cin,name);

    fflush(stdin);
    cout<<"Enter Employee Age :";
    cin>>age;

    fflush(stdin);
    cout<<"Enter Employee Salary:";
    cin>>salary;
    cout<<endl;
}
void Employee::display()
{
    cout.precision(2);
    cout<<"Name:"<<name<<endl;
    cout<<"Age:"<<age<<endl;
    cout<<"Salary:"<<fixed<<salary<<endl;
}
```

The fflush() function in C++ flushes any buffered data to the respective device.

Buffered data is the temporary or application specific data stored in the physical memory of the computer until a certain time.

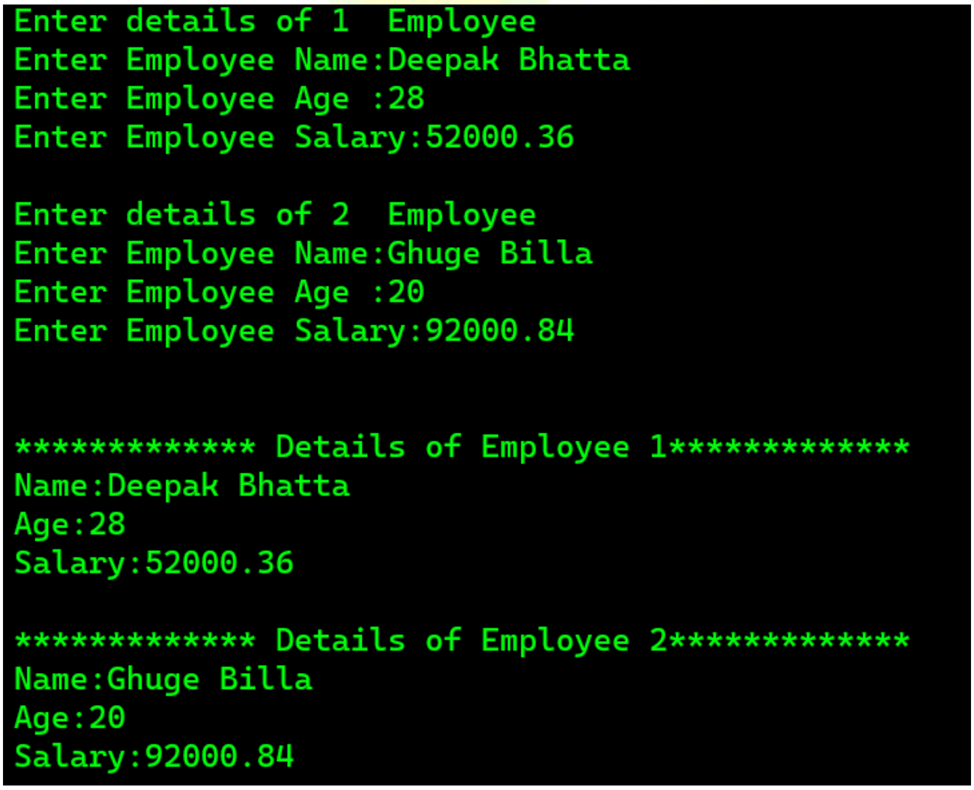
The fflush() function is defined in <cstdio> header file.

setprecision(n)

for individual data

```
int main()
{
    int i;
    Employee emp[2]; //Creating Array of 2 employees. Here emp is an object which has array so two objects will be created emp[0] and emp[1]
    for(i=0; i<2; i++)
    {
        cout << "Enter details of " << i+1 <<"  Employee"<<endl;
        emp[i].getData();
    }
    for(i=0; i<2; i++)
    {
        cout <<endl <<"************* Details of Employee " << i+1 << "*************" <<endl;
        emp[i].display();
    }
    return 0;
}
```

Output:

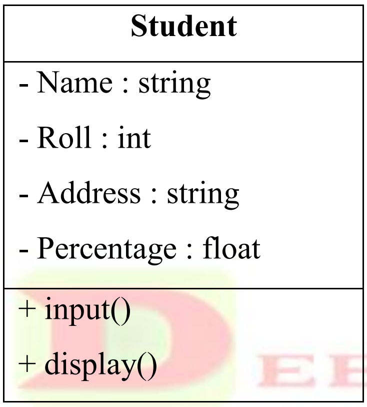


```
Enter details of 1  Employee
Enter Employee Name:Deepak Bhatta
Enter Employee Age :28
Enter Employee Salary:52000.36

Enter details of 2  Employee
Enter Employee Name:Ghuge Billa
Enter Employee Age :20
Enter Employee Salary:92000.84


************* Details of Employee 1*************
Name:Deepak Bhatta
Age:28
Salary:52000.36

************* Details of Employee 2*************
Name:Ghuge Billa
Age:20
Salary:92000.84
```

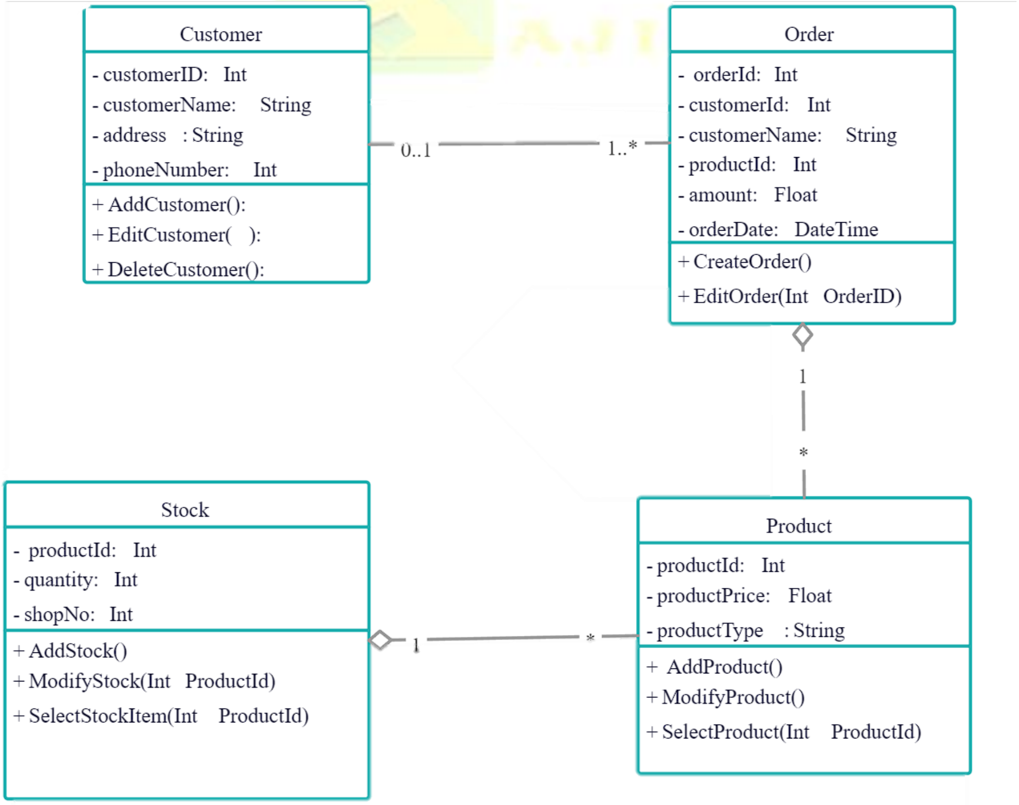
5. Class Diagram and Object Diagram

A class diagram is a UML diagram that shows the structure of a system by representing the system's classes, their attributes, and their relationships. An object diagram is a UML diagram that shows the instances of a system's classes at a particular point in time.

In C++, class diagrams can be used to design classes and their relationships. Object diagrams can be used to visualize the state of a system at a particular point in time.



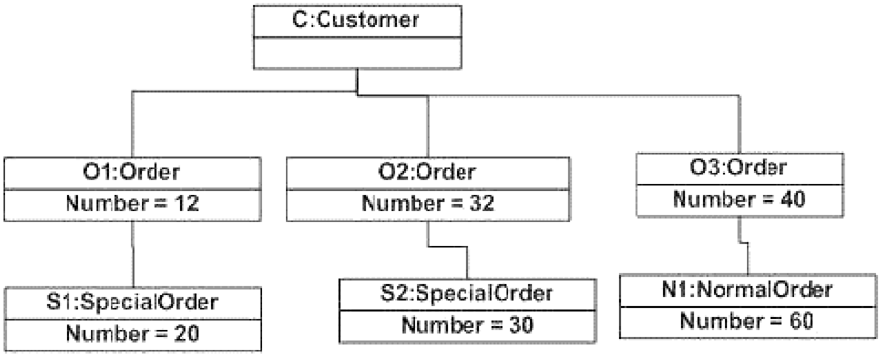
In this example, "**Student**" is the name of the class. The "+" signs denote public and "-" denote private access modifiers.



Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.

Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams. Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.

Object diagram of an order management system



An object diagram represents instances of classes and the relationships between these instances at a particular point in time. It shows objects and their attributes and associations.

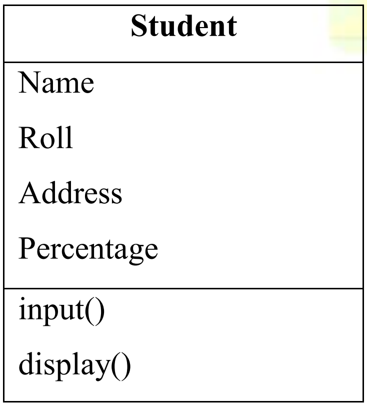
An object diagram is a structural diagram that represents a snapshot of a system at a specific point in time. It shows the instances of classes and the relationships between these instances. Object diagrams provide a visual representation of objects and their attributes, as well as the associations or links between them.

Feature	Class Diagram	Object Diagram
Purpose	Shows the static structure of a system, including classes, attributes, and methods.	Shows a snapshot of the dynamic state of a system at a particular point in time, including objects, their attributes, and their relationships.
Notation	Uses symbols to represent classes, attributes, methods, and relationships.	Uses symbols to represent objects, their attributes, and their relationships.

Example	A class diagram for a banking system might show classes for customers, accounts, and transactions.	An object diagram for a banking system might show a particular customer with a particular account and a particular transaction.
Example	A class diagram for a banking system might show the classes `Account`, `Transaction`, and `Customer`. The `Account` class would have attributes for the account number, balance, and owner. The `Transaction` class would have attributes for the transaction type, amount, and date. The `Customer` class would have attributes for the customer name, address, and phone number.	An object diagram for a banking system might show an object of type `Account` with the account number `123456789`, the balance `\$1000`, and the owner `John Smith`. The object diagram might also show objects of type `Transaction` for a deposit of `\$500` and a withdrawal of `\$200`.

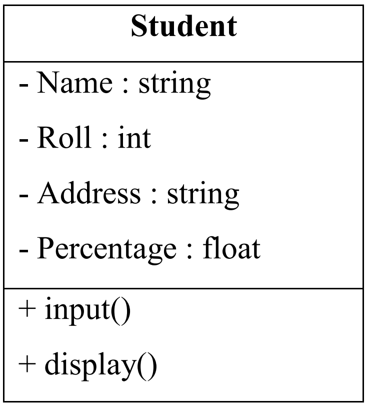
Example:

WAP to define the class in C++ as shown in class diagram.



OR

(Question may ask)



input() : to input initial values

display() : to display the record of students who passed

Note:- 45% is pass percentage

```
#include<iostream>
using namespace std;
class student
{
```

```
private:
    string name,address;
    int roll;
    float per;
public:
    void input()
    {
        fflush(stdin);
        cout<<"Enter Student Name : ";
        getline(cin,name);

        fflush(stdin);
        cout<<"Enter Student address : ";
        getline(cin,address);

        fflush(stdin);
        cout<<"Enter Student roll : ";
        cin>>roll;

        fflush(stdin);
        cout<<"Enter Student percentage : ";
        cin>>per;
        cout<<endl;
    }
    void display()
    {
        if(per>=45)
        {
            cout<<endl<<"********************************"<<endl;
            cout<<"Name = "<<name<<endl;
            cout<<"Roll = "<<roll<<endl;
            cout<<"Address = "<<address<<endl;
            cout<<"Percentage = "<<per<<endl;
        }
    }
};
```

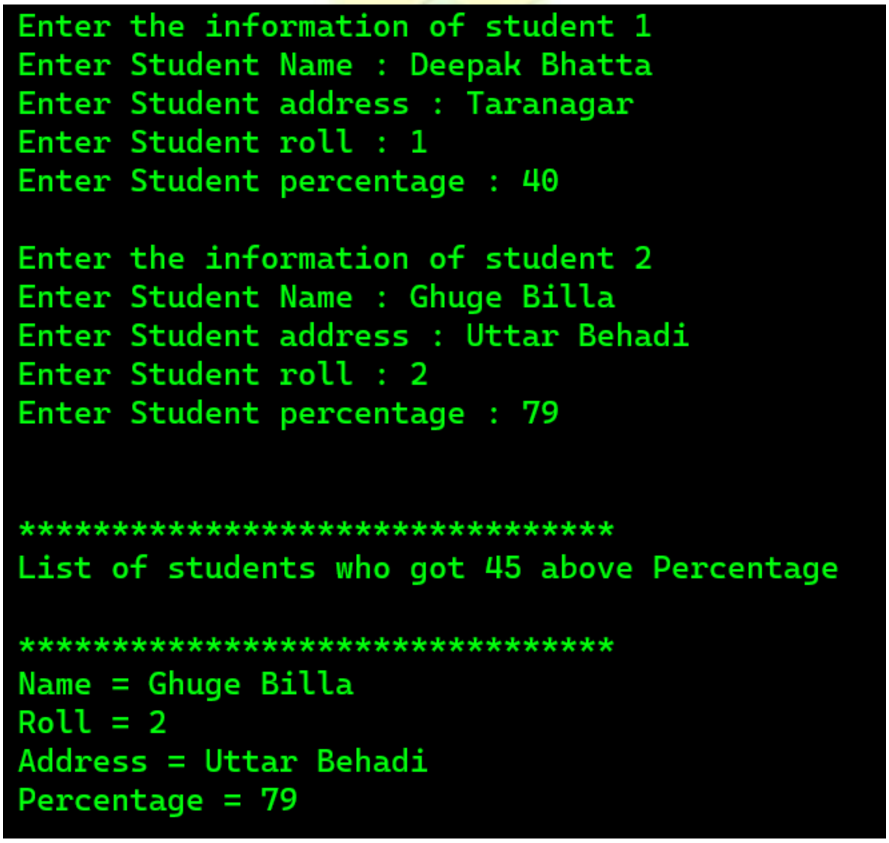
You can check for pass students who got 45 above inside the class definition as well otherwise, check inside the **main() function** using **if(s[i].per>=45)**

```
int main()
{
    int i;
    student s[2];
    for(i=0; i<2; i++)
    {
        cout<<"Enter the information of student "<<i+1<<endl;
        s[i].input();
    }
    cout<<endl<<"********************************"<<endl;
    cout<<"List of students who got 45 above Percentage"<<endl;
    for(i=0; i<2; i++)
    {
        s[i].display();
    }
    return 0;
}
```

OR

```
for(i=0; i<2; i++)
{
  if(s[i].per>=45)
  {
    s[i].display();
  }
}
```

Output:



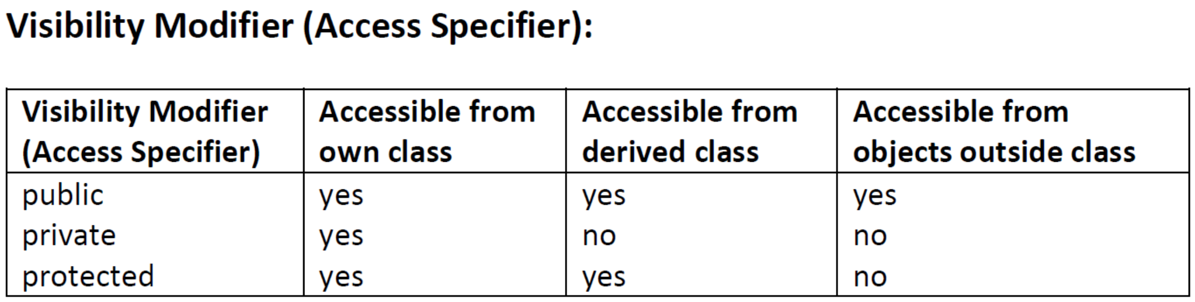
```
Enter the information of student 1
Enter Student Name : Deepak Bhatta
Enter Student address : Taranagar
Enter Student roll : 1
Enter Student percentage : 40

Enter the information of student 2
Enter Student Name : Ghuge Billa
Enter Student address : Uttar Behadi
Enter Student roll : 2
Enter Student percentage : 79


********************************
List of students who got 45 above Percentage

********************************
Name = Ghuge Billa
Roll = 2
Address = Uttar Behadi
Percentage = 79
```

6. Access Specifiers and Visibility Mode

In C++, access specifiers are used to define the visibility and accessibility of class members (variables and functions) within a class or from outside the class. There are three access specifiers in C++: **public**, **private**, and **protected**.

1. **Public Access Specifier:**
    - When a class member is declared as **public**, it can be accessed by any code that has access to the object of that class.
    - Public members are accessible from anywhere in the program, including outside the class.
    - They can be accessed using the object of the class or through the class's pointers.
2. **Private Access Specifier:**
    - When a class member is declared as **private**, it can only be accessed by other members of the same class.
    - Private members are not accessible from outside the class.
    - They are typically used to encapsulate and hide implementation details of a class, providing data hiding.
3. **Protected Access Specifier:**
    - Protected access is similar to private access, but with an additional feature.
    - Protected members are accessible within the class and its derived classes.
    - Protected members are not accessible from outside the class or any non-derived class.

By default, in C++, class members are private if not explicitly specified.



Visibility Modifier (Access Specifier):

Visibility Modifier (Access Specifier)	Accessible from own class	Accessible from derived class	Accessible from objects outside class
public	yes	yes	yes
private	yes	no	no
protected	yes	yes	no

In OOPs, the mechanism of deriving new class from old one is called inheritance.

- *The old class is referred to as **(base class /parent class/super class)** and new class is called as **(derived class/child class /sub class)**.*

Defining Derived Class (Specifying Derived Class):

General syntax:

```
class derived_class_name : visibility_mode base_class_name
{
//members of derived class
} ;
```

- The colon (:) indicates that the **derived_class_name** is derived from the **base_class_name**.
- The **visibility_mode** is optional, if present, may be either private or public.
- *The default visibility mode is **private**.*
- Visibility mode specifies whether the features of the base class are privately derived or publicly derived or derived on protected.

Example:

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

class MyClass {
public:
    int publicVar;    // Public member
    void publicMethod() {
       // Code accessible by any code that has access to an object of MyClass
    }
private:
    int privateVar;   // Private member
    void privateMethod() {
        // Code accessible only within the MyClass
    }
protected:
    int protectedVar; // Protected member
    void protectedMethod() {
        // Code accessible within MyClass and its derived classes
    }
};
```

Public Access Specifier:

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

class MyClass {
public:
    int publicVar;
    void publicMethod() {
        cout << "This is a public method." << endl;
    }
};

int main() {
    MyClass obj;
    obj.publicVar = 10;
    cout << obj.publicVar << endl; // Print the value of publicVar
    obj.publicMethod();
    return 0;
}
```

In this example, the **publicVar** member and the **publicMethod()** function are accessible from outside the class. We create an object of the **MyClass** and access its public members.

In the above example, if you make **publicVar** member as private then you cannot access it, it occurs an error *[error: 'int MyClass::publicVar' is private within this context]*

*For this you need to use the concept of **friend** function. We have next topic Function in our syllabus there we will discuss more.*

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

class MyClass {
//public:
    int publicVar;
public:
    void publicMethod() {
        cout << "This is a public method." << endl;
```

Remove **publicVar** from public and make it private see what the code looks like you cannot access the private member from outside the class. *Use the concept of **friend** function.*

```
    }
   // friend int main(); [incase you have private variable]
};

int main() {
    MyClass obj;
    obj.publicVar = 10; // Error! Cannot access private members from here. [To
resolve this issue use the concept of friend function inside the MyClass]
    cout << obj.publicVar << endl; // Print the value of publicVar
    obj.publicMethod();
    return 0;
}
```

Private Access Specifier:

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

class MyClass {
private:
    int privateVar;
    void privateMethod() {
        cout << "This is a private method." << endl;
    }

public:
    void publicMethod() {
        cout << "This is a public method." << endl;
        privateVar = 20;  // Private member accessed within the class
        cout << privateVar << endl; // Print the value of privateVar
        privateMethod(); // Private method called within the class
    }
};
int main() {
    MyClass obj;
    obj.publicMethod();
    return 0;
}
```

In this example, the **privateVar** member and the **privateMethod()** function are only accessible within the class. However, the **publicMethod()** function can access the private members. When we call **publicMethod()**, it internally accesses the private members of the class.

Protected Access Specifier:

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

class BaseClass {
protected:
    int protectedVar;
    void protectedMethod() {
        cout << "This is a protected method." << endl;
    }
public:
    int getProtectedVar() {
        return protectedVar;
    }
};

class DerivedClass : public BaseClass {
public:
    void derivedMethod() {
        protectedVar = 30;   // Accessing protected member from the derived class
        protectedMethod();    // Calling protected method of the base class
    }
};

int main() {
    DerivedClass obj;
    obj.derivedMethod(); // Accessing the protected member through the derived class
    cout << obj.getProtectedVar() << endl; // Print the value of protectedVar
    return 0;
}
```

Explanation of above code:

1. The code defines two classes: **BaseClass** and **DerivedClass**. **DerivedClass** inherits publicly from **BaseClass**, which means that all public and protected members of **BaseClass** are accessible within **DerivedClass**.
2. In the **BaseClass**, there is a protected member variable called **protectedVar**, and a protected member function called **protectedMethod()**. Both the variable and the method are accessible within the class and any derived classes.
3. **BaseClass** also provides a public member function **getProtectedVar()** that returns the value of **protectedVar**. Since this member function is defined in the base class, it can access the protected member.
4. In **DerivedClass**, there is a member function called **derivedMethod()**, which demonstrates accessing the protected member and calling the protected method inherited from **BaseClass**. Within this member function, **protectedVar** is assigned the value 30, and **protectedMethod()** is called.
5. In the **main()** function, an object **obj** of type **DerivedClass** is created.
6. **obj.derivedMethod()** is called to modify the value of the protected member and call the protected method.
7. Finally, **std::cout** is used to print the value of the protected member by calling **obj.getProtectedVar()**.

7. State and Behavior, Methods and Responsibilities

In C++, state refers to the data or variables that an object holds, while behavior refers to the actions or operations that an object can perform. Methods are the functions associated with an object that define its behavior, and responsibilities are the tasks or functionalities that an object is responsible for.

Here's a breakdown of each concept in C++:

1. **State:** State represents the data or variables that an object holds. These variables define the current condition or attributes of an object. For example, a "**Car**" class may have state variables such as "**color**," "**make**," "**model**," and "**speed**."
2. **Behavior:** Behavior refers to the actions or operations that an object can perform. These actions can manipulate the state of an object or interact with other objects. For instance, a "**Car**" class may have behaviors such as "**accelerate**," "**brake**," and "**changeGear**."
3. **Methods:** Methods are functions associated with a class or object that define its behavior. They encapsulate the actions that an object can perform. In C++, methods are declared within the class definition and can access the object's state variables. For example, a "**Car**" class may have methods like "**accelerate()**" and "**brake()**" to perform the respective actions.
4. **Responsibilities:** Responsibilities represent the tasks or functionalities that an object is responsible for. These responsibilities define the purpose of the object in the system. For instance, in a banking system, a "**BankAccount**" class may have responsibilities such as "**deposit**," "**withdraw**," and "**getBalance**."

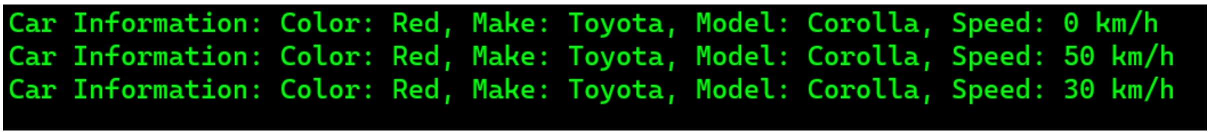
In C++, you typically define the state variables as private or protected members of a class to encapsulate them. The methods, which define the behavior and responsibilities, are usually declared as public members of the class. This ensures that the state remains encapsulated and can only be accessed or modified through the defined methods, promoting encapsulation and information hiding.

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

class Car {
private:
    string color;
    string make;
    string model;
    int speed = 0;

public:
    void setColor(const string newColor) {
        color = newColor;
    }

    void setMake(const string newMake) {
        make = newMake;
    }

    void setModel(const string newModel) {
        model = newModel;
    }

    void accelerate(int amount) {
        speed = speed + amount;
    }

    void brake(int amount) {
        speed = speed - amount;
        if (speed < 0)
            speed = 0;
    }

    void printInfo() {
        cout << "Car Information: "
                  << "Color: " << color << ", "
                  << "Make: " << make << ", "
                  << "Model: " << model << ", "
                  << "Speed: " << speed << " km/h" << endl;
    }
};

int main() {
    Car myCar;

    myCar.setColor("Red");
    myCar.setMake("Toyota");
    myCar.setModel("Corolla");

    myCar.printInfo();

    myCar.accelerate(50);
    myCar.printInfo();

    myCar.brake(20);
    myCar.printInfo();

    return 0;
}
```

In this code, the main function demonstrates the usage of the **Car** class. An instance of the **Car** class named **myCar** is created. The state of the car, such as its **color**, **make**, and **model**, is set using the **setColor**, **setMake**, and **setModel** methods, respectively.

After each modification of the car's state, the **printInfo** method is called to display the car's information. Additionally, the **accelerate** and **brake** methods are used to change the car's **speed**, and the information is printed again to reflect the changes.

Output:



```
Car Information: Color: Red, Make: Toyota, Model: Corolla, Speed: 0 km/h
Car Information: Color: Red, Make: Toyota, Model: Corolla, Speed: 50 km/h
Car Information: Color: Red, Make: Toyota, Model: Corolla, Speed: 30 km/h
```

8. Implementation of Data Abstraction, Encapsulation, Message Passing and Data Hiding

Implementation of Data Abstraction:

Data abstraction is a fundamental concept in object-oriented programming that allows us to hide the internal details of a class and provide only essential information to the outside world. It helps in encapsulating data and behavior within a class and providing a clean interface for interacting with the class.

In C++, data abstraction is typically achieved using classes and access specifiers **(public, private, and protected)**. The private members of a class are hidden from the outside world and can only be accessed through public member functions, which act as the interface to the class.

Let's see a simple example of data abstraction in C++:

```
#include <iostream>
using namespace std;
class Rectangle {
private:
    int length;
    int width;
public:
    void setDimensions(int L, int W) {
        length = L;
        width = W;
    }
    int getArea() {
        return length * width;
    }
};
int main() {
    Rectangle rect;
    rect.setDimensions(5, 3);
    int area = rect.getArea();
    cout << "Area: " << area << endl;
    return 0;
}
```

Explanation of above data abstraction code:

In this example, we have a **Rectangle** class representing a rectangle shape. The class has two private data members: **length** and **width**, which are hidden from the outside world. The public member functions **setDimensions** and **getArea** provide the interface to interact with the class. The **setDimensions** function allows us to set the length and width of the rectangle, while the **getArea** function calculates and returns the area of the rectangle based on the stored dimensions.

In the **main** function, we create an instance of the **Rectangle** class named **rect**. We then use the **setDimensions** function to set the dimensions of the rectangle to 5 and 3. Finally, we call the **getArea** function to retrieve the area of the rectangle and print it to the console.

By encapsulating the data members (**length** and **width**) as private and providing public member functions to interact with them, we achieve data abstraction. The internal details of the class are hidden, and the user only needs to know how to use the provided public interface to work with the class.

Implementation of Encapsulation:

Encapsulation is one of the fundamental principles of object-oriented programming (OOP) that allows you to bundle data and the methods that operate on that data into a single unit called a class. It helps in hiding the internal details of an object and provides a way to access and modify the data through controlled methods.

Let's see a simple example of Encapsulation in C++:

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

class Car {
private:
    string brand;
    string model;
    int year;
```

```
public:
    // Setter methods
    void setBrand(string b) {
        brand = b;
    }

    void setModel(string m) {
        model = m;
    }

    void setYear(int y) {
        year = y;
    }

    // Getter methods
    string getBrand() {
        return brand;
    }

    string getModel() {
        return model;
    }

    int getYear() {
        return year;
    }
};

int main() {
    Car myCar;

    // Set the car details using setter methods
    myCar.setBrand("Toyota");
    myCar.setModel("Camry");
    myCar.setYear(2020);
    // Get and display the car details using getter methods
    cout << "Brand: " << myCar.getBrand() << endl;
    cout << "Model: " << myCar.getModel() << endl;
    cout << "Year: " << myCar.getYear() << endl;

    return 0;
}
```

Explanation of above Encapsulation code:

In this example, we have a class called **Car**, which has private member variables **brand**, **model**, and **year**. These variables are encapsulated within the class and cannot be accessed directly from outside the class.

To access and modify these variables, we have provided public setter and getter methods. The setter methods (**setBrand()**, **setModel()**, and **setYear()**) are used to set the values of the private variables, while the getter methods (**getBrand()**, **getModel()**, and **getYear()**) are used to retrieve the values.

In the **main()** function, we create an instance of the **Car** class called **myCar**. We then use the setter methods to set the brand, model, and year of the car. Finally, we use the getter methods to retrieve and display the car details.

Implementation of Message Passing:

Message passing is a way for objects to communicate with each other in C++. It is a fundamental concept in object-oriented programming (OOP), and it is used to implement many of the features of OOP, such as polymorphism and inheritance.

In message passing, an object sends a message to another object. The message contains a request for the receiving object to perform some action. The receiving object then executes the requested action and returns a response to the sending object.

Message passing is a powerful way to decouple objects from each other. This means that objects can be changed or replaced without affecting the other objects that they communicate with. This makes it easier to maintain and extend large software systems.

There are two main types of message passing in C++: synchronous and asynchronous.

- **Synchronous message** passing is the simplest type of message passing. When an object sends a synchronous message to another object, the sending object blocks until the receiving object has responded. This means that the sending object cannot continue until the receiving object has finished executing the requested action.
- **Asynchronous message** passing is more complex than synchronous message passing. When an object sends an asynchronous message to another object, the sending object does not block. This means that the sending object can continue executing its code even though the receiving object has not yet responded. The receiving object will eventually respond to the message, but the sending object will not be notified of the response until it is ready to receive it.

Example:

```
#include<iostream>
using namespace std;
class Car {
public:
  void Drive() {
    cout << "The car is driving." << endl;
  }
};

class Person {
public:
  void DriveCar(Car &car) {
    car.Drive();
  }
};

int main() {
  Car car;
  Person person;
  person.DriveCar(car);
  return 0;
}
```

Explanation of above Message passing code:

In this example, the **Person** object sends a message to the **Car** object. The message is a request for the **Car** object to execute the **Drive()** method. The **Car** object then executes the **Drive()** method and returns a response to the **Person** object.

In this example, the message passing is synchronous. This means that the **Person** object blocks until the **Car** object has finished executing the **Drive()** method.

Message passing can also be asynchronous. In asynchronous message passing, the **Person** object would not block. This means that the **Person** object could continue executing its code even though the **Car** object has not yet responded. The **Car** object would eventually respond to the message, but the **Person** object would not be notified of the response until it is ready to receive it.

Implementation of Data Hiding:

Data hiding in C++ is typically achieved using the concept of access specifiers. Access specifiers define the visibility and accessibility of class members (variables and functions) from outside the class. There are three access specifiers in C++:

1. **Public:** Public members are accessible from anywhere, both inside and outside the class.
2. **Private:** Private members are only accessible from within the class. They are hidden from the outside world.
3. **Protected:** Protected members are similar to private members, but they are also accessible by derived classes.

To implement data hiding in C++, you can follow these steps:

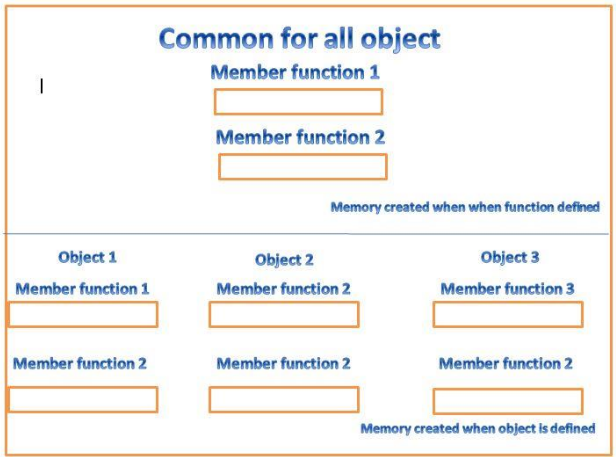
Step 1: Define a class and specify the access specifiers for the members.

Step 2: Access the members from outside the class using objects.

Example:

```
class MyClass {
private:
    int privateData;   // private member variable
public:
    int publicData;   // public member variable

private:
    void privateMethod() {
cout<<"I am Hiding my Data"<<endl;
        // private member function
    }

public:
    void publicMethod() {
cout<<"I am showing my Data"<<endl;
        // public member function
    }
};

int main() {
    MyClass obj;
    obj.publicData = 10;   // accessing publicData
    cout<<obj.publicData<<endl;
    obj.publicMethod();    // calling publicMethod

    // The following statements will result in compilation errors:
    // obj.privateData = 20;     // accessing privateData (error)
    // obj.privateMethod();      // calling privateMethod (error)

    return 0;
}
```

9. Memory Allocations for Objects



Memory allocation for objects refers to the process of reserving memory in a computer's memory space to store the data and state of an object. When an object is created in a programming language like C++, memory must be allocated to hold its member variables, methods, and any other necessary data.

In C++, memory allocation for objects can be classified into two main categories: stack allocation and heap allocation.

1. **Stack Allocation:** Stack allocation involves reserving memory on the stack, which is a region of memory managed by the compiler. When an object is allocated on the stack, the memory is automatically allocated and deallocated as the object goes in and out of scope. This means that stack-allocated objects are automatically deallocated when they are no longer needed. Stack allocation is generally faster and has less overhead compared to heap allocation. It is the default and preferred way for most objects in C++. The memory for stack-allocated objects is managed by the compiler.
   Here's an example of stack allocation:
   ```
   void function() {
       int x = 5;  // stack-allocated integer
       // ...
   }  // x is deallocated automatically when function() returns
   ```
2. **Heap Allocation:** Heap allocation involves reserving memory on the heap, which is a larger region of memory used for dynamic memory allocation. When an object is allocated on the heap, memory is allocated using the **new** operator, and it remains allocated until it is explicitly deallocated using the **delete** operator. Heap-allocated objects have a longer lifespan and need to be manually managed. This allows for more control over the lifetime and scope of the object. Heap allocation is useful when the object's lifetime extends beyond the scope of a function or when dynamic memory allocation is required.

It's important to note that with heap allocation, it is the responsibility of the programmer to explicitly deallocate the memory to avoid memory leaks.

Failure to deallocate heap-allocated memory can result in memory leaks, where memory is not freed up even when it is no longer in use, leading to inefficient memory usage.

Here's an example of heap allocation:

```
void function() {
    int* ptr = new int;  // heap-allocated integer
    // ...
    delete ptr;  // deallocate the memory
}
```

Additionally, if you allocate an array of objects on the heap, you should use **delete[]** to deallocate it properly:

```
void function() {
    int* arr = new int[10];  // heap-allocated array of integers
    // ...
    delete[] arr;  // deallocate the array
}
```

In summary, memory allocation for objects in C++ involves reserving memory either on the stack or the heap, depending on the desired lifetime and control requirements of the object. Stack allocation is automatic and efficient, while heap allocation provides greater control but requires manual memory management.

Extra:

Alternatively, in modern C++, it is recommended to use smart pointers like **std::shared_ptr** or **std::unique_ptr** for managing heap-allocated objects. These smart pointers provide automatic memory management and help prevent memory leaks.

```
#include <memory>
void function() {
    std::shared_ptr<int> ptr = std::make_shared<int>();  // heap-allocated
integer using std::shared_ptr
    // ...
}  // ptr is deallocated automatically when it goes out of scope
```

Using smart pointers reduces the manual memory management burden and provides automatic deallocation based on the ownership semantics.

10. Constructor: Default Constructor, Parameterized Constructor, Copy Constructor

Constructor:

A constructor is a **special member function** in object-oriented programming languages, such as C++, that is responsible for initializing objects of a class. It is **called automatically when an object is created**, ensuring that the object is properly initialized and ready for use.

The primary purpose of a constructor is to set up the initial state of an object. This includes initializing the data members of the class, allocating resources if needed, and performing any necessary setup operations. The constructor is automatically executed whenever an object is created. Thus, a constructor helps to initialize the objects without making a separates call to a member function.

It is called constructor because it constructs values of data members of a class.

How constructor differ from normal member function:

Constructors differ from normal member functions in several ways:

1. **Name and Return Type**: Constructors have the same name as the class, while normal member functions have their own distinct names. Constructors do not have a return type, not even **void**, whereas normal member functions have a return type (unless they are **void** functions).
2. **Automatic Invocation**: Constructors are automatically invoked when an object is created, while normal member functions are called explicitly by the user. Constructors are called implicitly, without the need for the user to explicitly call them. Normal member functions, on the other hand, need to be invoked using the object's name and the function call syntax.
3. **Initialization**: Constructors are primarily used to initialize the data members of a class and set up the object's initial state. They are responsible for ensuring that the object is properly initialized when it is created. Normal member functions, on the other hand, are used to perform specific operations or provide functionality to manipulate the object's data or perform some other task.

4. **Parameters**: Constructors can have parameters or no parameters at all, depending on the specific initialization requirements of the class. Normal member functions, on the other hand, typically have parameters that are used to pass data or provide information for the specific operations they perform.
5. **Multiple Constructors**: A class can have multiple constructors with different parameter lists. This allows objects to be created with different initialization values or no values at all. ***This feature is known as constructor overloading***. Normal member functions can also be overloaded, but they need to have different names or different parameter lists.
6. **Destructor**: Constructors are complemented by destructors, which are special member functions called when an object is destroyed or goes out of scope. Destructors are responsible for releasing resources acquired by the object and performing necessary cleanup operations. Normal member functions do not have a specific role in object destruction or cleanup.

Overall, constructors are used for object initialization and ensuring a valid state upon creation, while normal member functions are used for performing specific operations or providing functionality to manipulate the object's data.

In summary,

- Constructor has same name as the class name.
- Constructor don’t have return type.
- A constructor is automatically called when an object is created.
- Constructor can be declared in the public section.
- If we do not specify a constructor, C++ compiler generate a default constructor for us *(expects no parameters and has an empty body)*.

Types of Constructors:

1. **Default Constructor**: A default constructor is a constructor that takes no parameters. If a class does not explicitly define any constructors, the compiler automatically generates a default constructor. It initializes the data members of the class with default values, typically zero or null. The default constructor is used when an object is created without any initialization values.

   Syntax:

   ```
   ClassName() {
       // Initialization code
   }
   ```

2. **Parameterized Constructor**: A parameterized constructor is a constructor that takes one or more parameters. It allows objects to be created with specific initialization values. The parameterized constructor allows you to initialize the data members of the class with values passed as arguments during object creation.

   This can be done in two ways.

   - by calling the constructor implicitly

     Eg.: Car car("Toyota"); ***//implicit call***

   - by calling the constructor explicitly

     Eg.: Car car =Car("Toyota"); ***//explicit call***

   Syntax:

   ```
   ClassName(parameters) {
         // Initialization code using parameters
   }
   ```

3. **Copy Constructor**: A copy constructor is a constructor that creates a new object by making a copy of an existing object of the same class. It takes a reference to an object of the same class as its parameter. The copy constructor is used when an object is created as a copy of an existing object, either explicitly or implicitly.

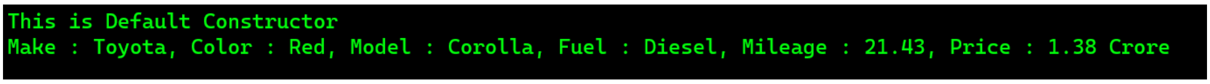
   Syntax:

   ```
   ClassName(const ClassName &obj) {
       // Copy the data members of 'obj' into the current object
   }
   ```

Example of Default Constructor:

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

//class declaration and definition
class Cars
{
private:
    //data members
    string make;
    string color;
    string model;
    string fuel;
    float mileage;
    string price;
public:
    //default Constructor
    Cars()
    {
        cout<<"This is Default Constructor"<<endl;
        make = "Toyota";
        color = "Red";
        model = "Corolla";
        fuel = "Diesel";
        mileage = 21.43;
        price = "1.38 Crore";
    }
    //member functions
    void displayData()
    {
        cout<<"Make : "<<make<<", "
        <<"Color : "<<color<<", "
        <<"Model : "<<model<<", "
        <<"Fuel : "<<fuel<<", "
        <<"Mileage : "<<mileage<<", "
        <<"Price : "<<price<<endl;
    }
};

int main()
{
    Cars car1;
    car1.displayData();
    return 0;
}
```

Output:

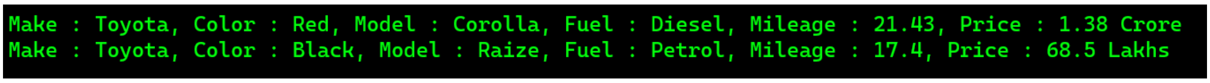


```
This is Default Constructor
Make : Toyota, Color : Red, Model : Corolla, Fuel : Diesel, Mileage : 21.43, Price : 1.38 Crore
```

Example of Parametrized Constructor:

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

//class declaration and definition
class Cars
{
private:
    //data members
    string make;
    string color;
    string model;
    string fuel;
    float mileage;
    string price;
public:
    //Parameterized Constructor
    Cars(string newMake, string newColor, string newModel, string newFuel,
float newMileage, string newPrice)
    {
        make = newMake;
        color = newColor;
        model = newModel;
        fuel = newFuel;
        mileage = newMileage;
        price = newPrice;
    }
    void displayData()
    {
        cout<<"Make : "<<make<<", "
        <<"Color : "<<color<<", "
        <<"Model : "<<model<<", "
        <<"Fuel : "<<fuel<<", "
        <<"Mileage : "<<mileage<<", "
        <<"Price : "<<price<<endl;
    }
};
int main()
{
    Cars car1("Toyota","Red","Corolla","Diesel",21.43,"1.38 Crore");
//implicit call
    Cars car2 = Cars("Toyota","Black","Raize","Petrol",17.4,"68.5 Lakhs");
//explicit call
    car1.displayData();
    car2.displayData();
    return 0;
}
```

Output:



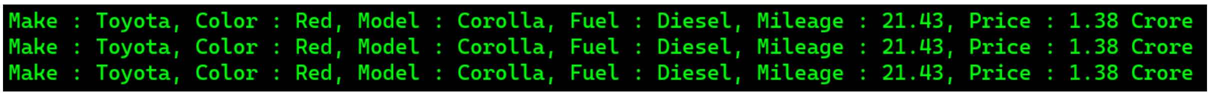
```
Make : Toyota, Color : Red, Model : Corolla, Fuel : Diesel, Mileage : 21.43, Price : 1.38 Crore
Make : Toyota, Color : Black, Model : Raize, Fuel : Petrol, Mileage : 17.4, Price : 68.5 Lakhs
```

Example of Copy Constructor:

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

//class declaration and definition
class Cars
{
private:
    //data members
    string make;
    string color;
    string model;
    string fuel;
    float mileage;
    string price;

public:
 //Parameterized Constructor
  Cars(string newMake, string newColor, string newModel, string newFuel,
float newMileage, string newPrice)
    {
        make = newMake;
        color = newColor;
        model = newModel;
        fuel = newFuel;
        mileage = newMileage;
        price = newPrice;
    }

    //Copy Constructor
    Cars(Cars &old_obj) // reference type parameter [old_obj is a referenced
variable (This is user-defined copy constructor)AKA Deep Copy
    {
        make = old_obj.make;
        color = old_obj.color;
        model = old_obj.model;
        fuel = old_obj.fuel;
        mileage = old_obj.mileage;
        price = old_obj.price;
    }

    void displayData()
    {
        cout<<"Make : "<<make<<", "
        <<"Color : "<<color<<", "
        <<"Model : "<<model<<", "
        <<"Fuel : "<<fuel<<", "
        <<"Mileage : "<<mileage<<", "
        <<"Price : "<<price<<endl;
    }
};

int main()
{
    Cars car1("Toyota","Red","Corolla","Diesel",21.43,"1.38 Crore");
//implicit call // Calling the parameterized constructor.
    Cars  car2(car1); //Copy Constructor
    Cars car3 = car2; //Copy Constructor

    car1.displayData();
    car2.displayData();
    car3.displayData();
    return 0;
}
```

This statement call the copy constructor

Output:



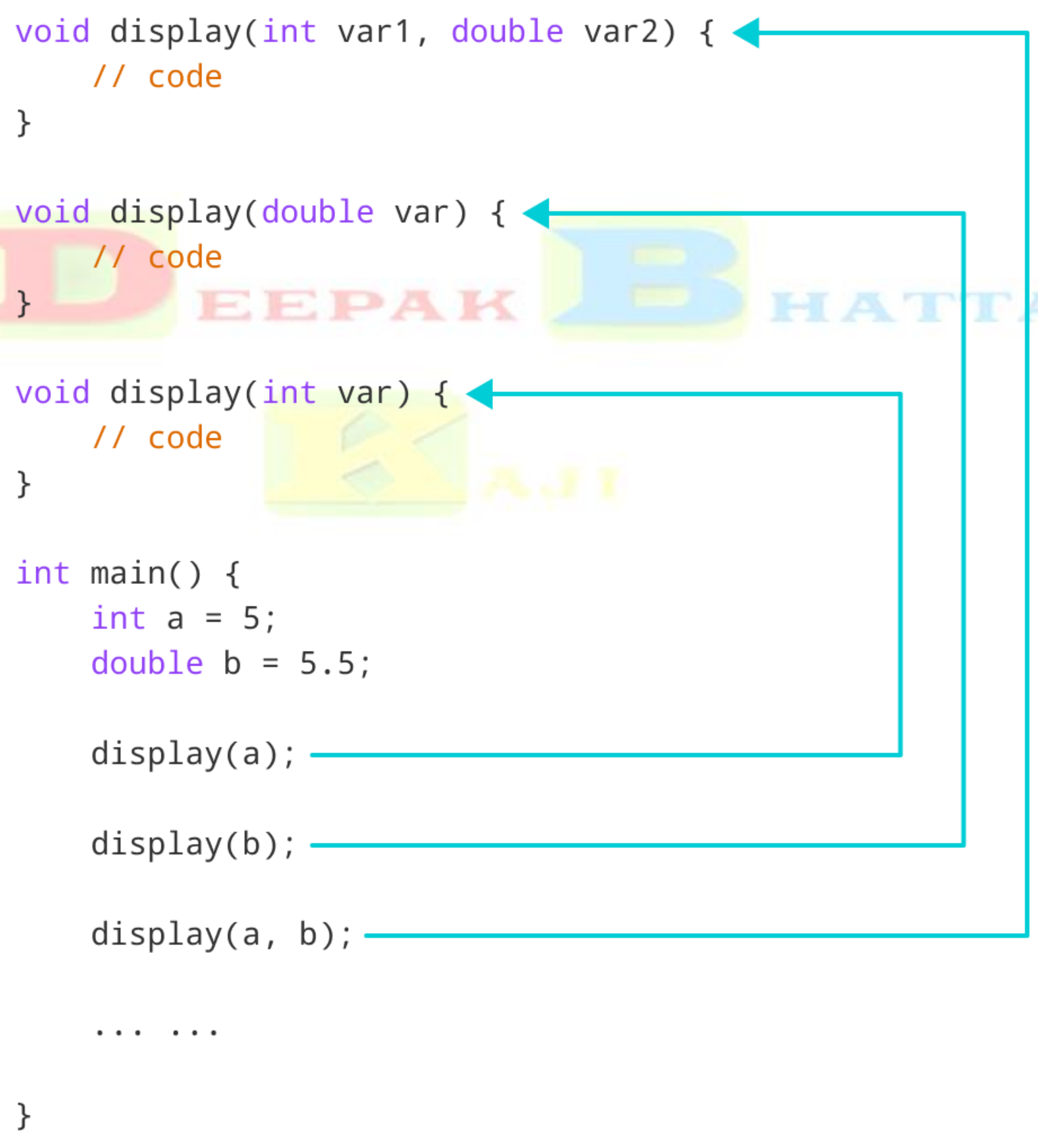
```
Make : Toyota, Color : Red, Model : Corolla, Fuel : Diesel, Mileage : 21.43, Price : 1.38 Crore
Make : Toyota, Color : Red, Model : Corolla, Fuel : Diesel, Mileage : 21.43, Price : 1.38 Crore
Make : Toyota, Color : Red, Model : Corolla, Fuel : Diesel, Mileage : 21.43, Price : 1.38 Crore
```

11. Constructor Overloading

Before starting constructor overloading we should know what exactly is overloading, for this we have to know about function overloading concept in C++.

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions.



```
void display(int var1, double var2) {
    // code
}

void display(double var) {
    // code
}

void display(int var) {
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);

    display(b);

    display(a, b);

    ... ...

}
```

For example:

```
// same name different arguments
int test()
{
//code
}
int test(int a)
{
//code
}
float test(double a)
{
 //code
}
int test(int a, double b)
{
//code
}
```

Here, all 4 functions are **overloaded functions**.

Notice that the return types of all these 4 functions are not the same. ***Overloaded functions may or may not have different return types but they must have different arguments***.

For example,

```
// Error code
int test(int a)
{
//code
}
double test(int b)
{
//code
}
```

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

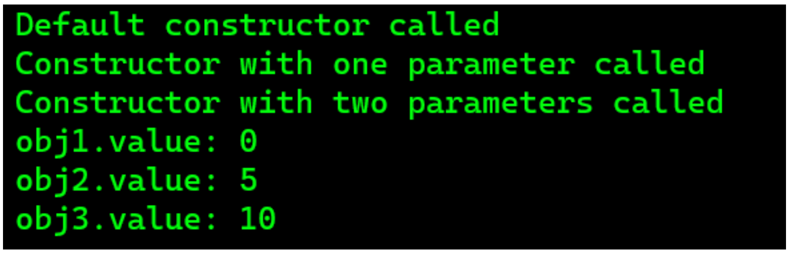
Same like function overloading we have constructor overloading.

Constructor overloading in C++ allows you to define **multiple constructors** for a class with **different parameter** lists. This enables you to create objects using different sets of arguments or initialize objects in different ways. Constructor overloading provides flexibility and allows you to create objects with different initial states based on the arguments passed to the constructor.

Example 1: Constructor Overloading Simple concept

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

class MyClass {
public:
    int value;

    // Default constructor
    MyClass() {
        value = 0;
        cout << "Default constructor called" << endl;
    }
    // Constructor with one parameter
    MyClass(int val) {
        value = val;
        cout << "Constructor with one parameter called" << endl;
    }
    // Constructor with two parameters
    MyClass(int val1, int val2) {
        value = val1 + val2;
        cout << "Constructor with two parameters called" << endl;
    }
};
int main() {
    // Creating objects using different constructors
    MyClass obj1; // Calls the default constructor
    MyClass obj2(5); // Calls the constructor with one parameter
    MyClass obj3(2, 3); // Calls the constructor with two parameters
    // Accessing object values
    cout << "obj1.value: " << obj1.value << endl;
    cout << "obj2.value: " << obj2.value << endl;
    cout << "obj3.value: " << obj3.value << endl;

    return 0;
}
```

Output:



```
Default constructor called
Constructor with one parameter called
Constructor with two parameters called
obj1.value: 0
obj2.value: 5
obj3.value: 10
```

Example 2: C++ program to demonstrate constructor overloading

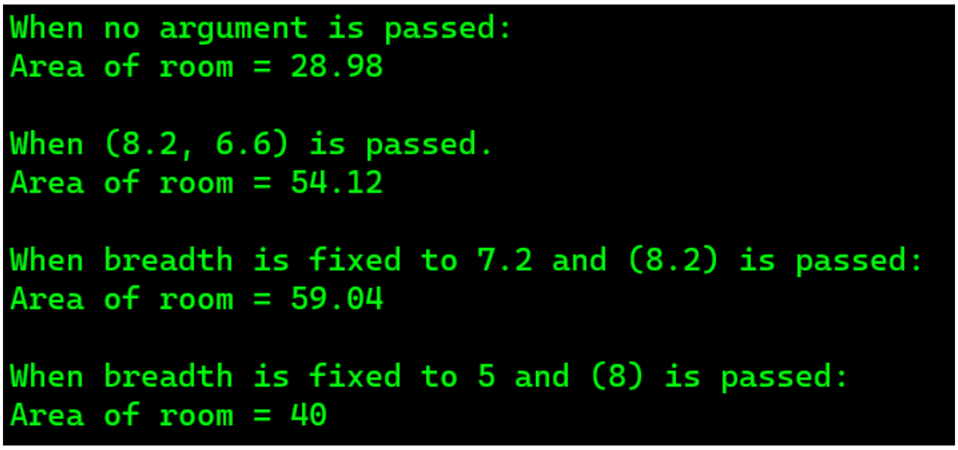
```
#include <iostream>
using namespace std;

class Room
{
private:
    double length;
    double breadth;
public:
    // 1. Constructor with no arguments
    Room()
    {
        length = 6.9;
        breadth = 4.2;
    }
    // 2. Constructor with two arguments both double datatype
    Room(double l, double b)
    {
        length = l;
        breadth = b;
    }
    // 3. Constructor with one argument double datatype
    Room(double len)
    {
        length = len;
        breadth = 7.2;
    }
    // 4. Constructor with one argument int datatype
    Room(int len)
    {
        length = len;
        breadth = 5;
    }
```

```
Room(double len)
 {
     length = len;
     breadth = 5;
 }
```

We cannot use same datatype arguments; it does not create overloading. It still shows an error even if you change the Argument variable name. *So must have different argument datatype.*

```
    double calculateArea()
    {
        return length * breadth;
    }
};
int main()
{
    Room room1, room2(8.2, 6.6), room3(8.2),room4(8);

    cout << "When no argument is passed: " << endl;
    cout << "Area of room = " << room1.calculateArea() << endl;

    cout << "\nWhen (8.2, 6.6) is passed." << endl;
    cout << "Area of room = " << room2.calculateArea() << endl;

    cout << "\nWhen breadth is fixed to 7.2 and (8.2) is passed:" << endl;
    cout << "Area of room = " << room3.calculateArea() << endl;

    cout << "\nWhen breadth is fixed to 5 and (8) is passed:" << endl;
    cout << "Area of room = " << room4.calculateArea() << endl;

    return 0;
}
```

Output:



```
When no argument is passed:
Area of room = 28.98

When (8.2, 6.6) is passed.
Area of room = 54.12

When breadth is fixed to 7.2 and (8.2) is passed:
Area of room = 59.04

When breadth is fixed to 5 and (8) is passed:
Area of room = 40
```

12. Destructors

In C++, destructors are special member functions of a class that are used to clean up resources and perform necessary actions before an object of that class is destroyed or goes out of scope. **They have the same name as the class, preceded by a tilde (~).**

Here's the general syntax of a destructor in C++:

```
class ClassName {
public:
    // Constructor(s) and other member functions

    // Destructor
    ~ClassName() {
        // Clean up resources and perform necessary actions
    }
};
```

Key points to note about destructors:

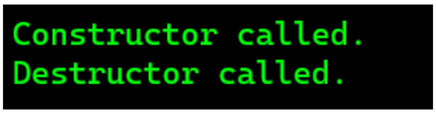
1. Destructors do not have return types, not even void.
2. A class can have only one destructor, and it cannot be overloaded with different parameters.
3. Destructors are automatically called when an object goes out of scope, such as when a local object is destroyed at the end of a function or when a dynamically allocated object is deleted using the **delete** keyword.

4. Destructors are called in the reverse order of object creation. If you have multiple objects of a class, the destructors will be called in the reverse order of their creation.
5. Destructors can be useful for releasing dynamically allocated memory, closing file handles, freeing resources, or performing any necessary cleanup actions before an object is destroyed.

Example 1: Simple Destructor Concept

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    // Constructor
    MyClass() {
        cout << "Constructor called." << endl;
    }
    // Destructor
    ~MyClass() {
        cout << "Destructor called." << endl;
    }
};

int main() {
    MyClass obj; // Object created
    // ... Do something with obj ...
    // Object goes out of scope, destructor called automatically
    return 0;
}
```

Output:



```
Constructor called.
Destructor called.
```

In the example, when the object **obj** goes out of scope at the end of the **main** function, the destructor is automatically called, resulting in the output: **"Destructor called."**

Remember, while destructors are not explicitly called by the programmer, they play an important role in managing resources and performing cleanup operations, ensuring that objects release their resources properly.

13. Dynamic Memory Allocation: new and delete

Dynamic memory allocation in C++ allows you to allocate memory at runtime and manage it manually using the **new** and **delete** operators.

1. The **new** operator: It is used to allocate memory for a single object dynamically. The general syntax is:

   ```
   pointer_variable = new data_type;
   ```

   For example, to allocate memory for an integer dynamically:

   ```
   int* ptr = new int;
   ```

   This statement allocates memory for an integer and returns a pointer to the allocated memory, which is then assigned to **ptr**.

2. The **delete** operator: It is used to deallocate memory that was allocated using **new**. The general syntax is:

   ```
   delete pointer_variable;
   ```

   For example, to deallocate the memory allocated for the integer pointer **ptr**:

   ```
   delete ptr;
   ```

   This statement frees the memory previously allocated using **new**.

   It's important to note that **delete** should only be used with pointers that were allocated using **new**.

3. Dynamic memory allocation for arrays: In addition to single objects, you can also allocate memory for arrays dynamically using **new[]** and deallocate it using **delete[]**.

   The syntax is as follows:

   ```
   pointer_variable = new data_type[size];
   ```

   For example, to allocate memory for an array of integers dynamically:

   ```
   int* arr = new int[5];
   ```

   To deallocate the memory for the array:

   ```
   delete[] arr;
   ```

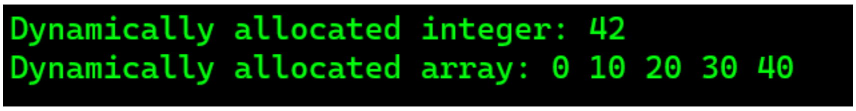
   This statement frees the memory previously allocated for the array using **new[]**.

   Remember that when using dynamic memory allocation, it is your responsibility to deallocate the memory using delete or delete[] to avoid memory leaks.

Example DMA in C++:

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

int main() {
    // Dynamic memory allocation for a single integer
    int* ptr = new int;  // Allocate memory for an integer

    *ptr = 42;  // Assign a value to the dynamically allocated integer

    cout << "Dynamically allocated integer: " << *ptr << endl;

    delete ptr;  // Deallocate the memory

    // Dynamic memory allocation for an array of integers
    int size = 5;
    int* arr = new int[size];  // Allocate memory for an array of integers

    for (int i = 0; i < size; i++) {
        arr[i] = i * 10;  // Assign values to the elements of the array
    }

    cout << "Dynamically allocated array: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";  // Print the elements of the array
    }
    cout << endl;

    delete[] arr;  // Deallocate the memory for the array

    return 0;
}
```

Output:



```
Dynamically allocated integer: 42
Dynamically allocated array: 0 10 20 30 40
```

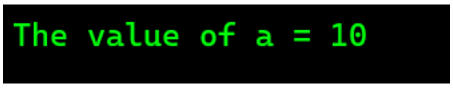
14. Dynamic Constructor

When allocation of memory is done dynamically using dynamic memory allocator **new** in a constructor, it is known as **dynamic constructor**. By using this, we can dynamically initialize the objects.

Example: Basic Dynamic Constructor

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

class Basic
{
    int *a;
public:
    Basic()
    {
        a=new int;
        *a=10;
    }
    void display()
    {
        cout<<"The value of a = "<< *a<<endl;
    }
};
int main()
{
    Basic obj;
    obj.display();
}
```

Output:



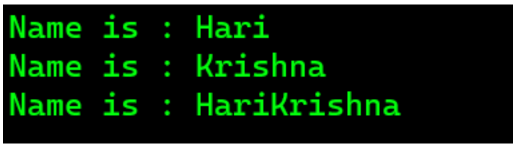
```
The value of a = 10
```

Example 2:

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

class Twins
{
    char *name;
    int len;

public:
    Twins()
    {
        len=0;
        name= new char[len+1]; //The extra 1 is for the null character at
the end of the string.   // len+1 for extra memory for null character
    }

    Twins(char *s) // It takes a character array (char *s) as an argument.
    {
        len =strlen(s);
        name = new char[len+1];
        strcpy(name,s);
    }

    void display()
    {
        cout<<"Name is : "<<name<<endl;

    }

    void join(Twins &a, Twins &b)
    {
        len = a.len + b.len; //a.len b.len represents the length of the name
array of the Twins object a and object b.
```

```
        delete name;
//The line delete name; is used to deallocate the memory previously allocated
for the name character array.
        name = new char[len+1];

        strcpy(name,a.name);
        strcat(name, b.name);
    }
};

int main()
{
    Twins name1("Hari");
    Twins name2("Krishna");
    Twins name3;

    name3.join(name1,name2);

    name1.display();
    name2.display();
    name3.display();
}
```

Output:



```
Name is : Hari
Name is : Krishna
Name is : HariKrishna
```

15. Functions: Inline Function, Default argument, Passing and Returning Value, Pointer and Reference, Static Data Member and Static Member Function

Inline Function: An inline function is a function that is expanded in-line at the point of its call, rather than invoking a function call. It is typically used for small, frequently executed functions, where the function code is inserted directly at the call site, eliminating the overhead of a function call. To declare a function as inline, you can use the **inline** keyword before the function declaration.

Default Argument: A default argument is a value provided in the function declaration that is automatically used if no corresponding argument is passed during the function call. It allows you to provide a default value for one or more function parameters, making those parameters optional. Default arguments are specified in the function declaration, typically in the function prototype.

Passing and Returning Value: Passing values to a function involves providing input values (arguments) to the function during its invocation. The function receives these values and can perform computations or operations using them. The function can also return a value back to the caller using a return statement, which can be of any data type.

Pointer and Reference: Pointers and references are used to manipulate and work with memory locations in C++.

- **Pointers:** A pointer is a variable that stores the memory address of another variable. By using pointers, you can indirectly access and modify the value of the variable it points to. ***Pointers are declared using the * operator***.

- **References:** A reference is an alias or an alternative name for an existing variable. Unlike pointers, references cannot be reassigned to point to a different variable. **References are declared using the & operator**.

Static Data Member and Static Member Function:

- **Static Data Member:** In C++, a static data member is a member of a class that is shared by all instances (objects) of the class. It means that a single copy of the static member is shared among all the objects of that class. Static data members are declared using the **static** keyword.

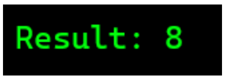
- **Static Member Function:** A static member function is a function that belongs to the class rather than the objects of the class. It can be invoked using the class name without creating an object of the class. Since it doesn't have access to the object-specific data members, it can only access static data members and other static member functions.

Examples:

Inline Function:

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

// Inline function declaration
inline int add(int a, int b) {
    return a + b;
}
int main() {
    int result = add(5, 3); // Function call
    cout << "Result: " << result << endl;
    return 0;
}
```

Output:

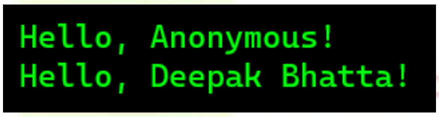


```
Result: 8
```

Default Argument:

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

// Function declaration with default argument
void greet(string name = "Anonymous") {
   cout << "Hello, " << name << "!" << endl;
}
int main() {
   greet(); // Using the default argument
   greet("Deepak Bhatta"); // Providing a custom argument
   return 0;
}
```

Output:



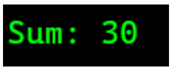
```
Hello, Anonymous!
Hello, Deepak Bhatta!
```

Passing and Returning Value:

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

// Function that calculates the sum of two numbers and returns the result
int add(int a, int b) {
    return a + b;
}

// Function that prints the result of adding two numbers
void printSum(int a, int b) {
    int sum = add(a, b); // Function call and assigning the return value to
a variable
    cout << "Sum: " << sum << endl;
}
```

```
int main() {
    printSum(10, 20); // Function call
    return 0;
}
```

Output:

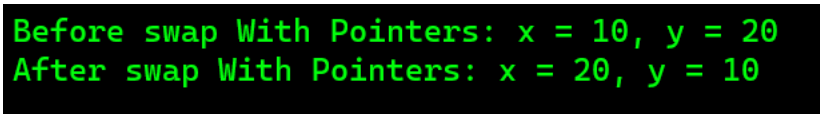


```
Sum: 30
```

Pointer Swapping:

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

// Function that swaps the values of two integers using pointers
void swapWithPointers(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;

    cout << "Before swap With Pointers: x = " << x << ", y = " << y << endl;

    // Swap using pointers
    swapWithPointers(&x, &y);
    cout << "After swap With Pointers: x = " << x << ", y = " << y << endl;

    return 0;
}
```

Output:

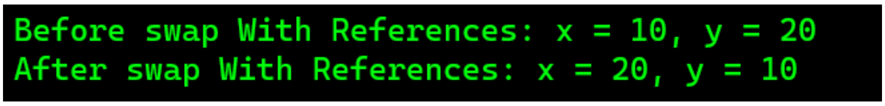


```
Before swap With Pointers: x = 10, y = 20
After swap With Pointers: x = 20, y = 10
```

Reference Swapping:

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

// Function that swaps the values of two integers using references
void swapWithReferences(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    cout << "Before swap With References: x = " << x << ", y = " << y <<
endl;
    // Swap using references
    swapWithReferences(x, y);
    cout << "After swap With References: x = " << x << ", y = " << y <<
endl;

    return 0;
}
```

Output:



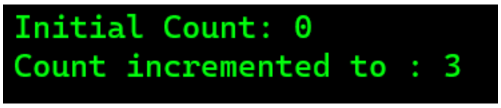
```
Before swap With References: x = 10, y = 20
After swap With References: x = 20, y = 10
```

Static Data Member:

A static member is shared by all objects of the class. The properties of a static member variables are similar to that of a C static variable. All static data is **initialized to zero** *(because its default value is zero)* when first object is created, if no other initialization is present. We **can't initialized it in the class definition** but it **can be initialized outside the class** using the scope resolution operator **::** to identify which class it belongs to. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. *It is shared memory for all objects of the class.* It retains its value.

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

class StaticData
{
public:
    int a;
    static int count;

    StaticData()
    {
        count++;
    }
};
int StaticData::count=0;

int main()
{
    cout<<"Initial Count: "<<StaticData::count<<endl;
    StaticData obj1,obj2,obj3; //default constructor is called
    cout<<"Count incremented to : "<<StaticData::count<<endl;
    return 0;
}
```

Output:

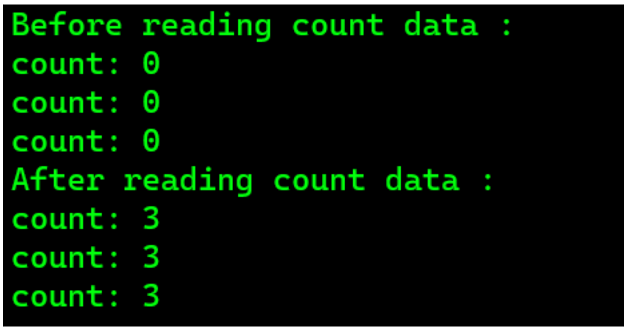


```
Initial Count: 0
Count incremented to : 3
```

Example 2:

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

class Item
{
    static int count;
    int num;
public:
    void getData(int a)
    {
        num = a;
        count++;
    }
    void getCount()
    {
        cout<<"count: "<<count<<endl;
    }
};

int Item :: count;

int main()
{
    Item obj1,obj2,obj3; //count is initialized to zero
   cout<<"Before reading count data : "<<endl;
//Display Count value
    obj1.getCount();
    obj2.getCount();
    obj3.getCount();
    obj1.getData(10); //Getting data into object obj1
    obj2.getData(20); //Getting data into object obj2
    obj3.getData(30); //Getting data into object obj3

    cout<<"After reading count data : "<<endl;
//Display Count value
    obj1.getCount();
    obj2.getCount();
    obj3.getCount();
    return 0;
}
```

Output:



```
Before reading count data :
count: 0
count: 0
count: 0
After reading count data :
count: 3
count: 3
count: 3
```

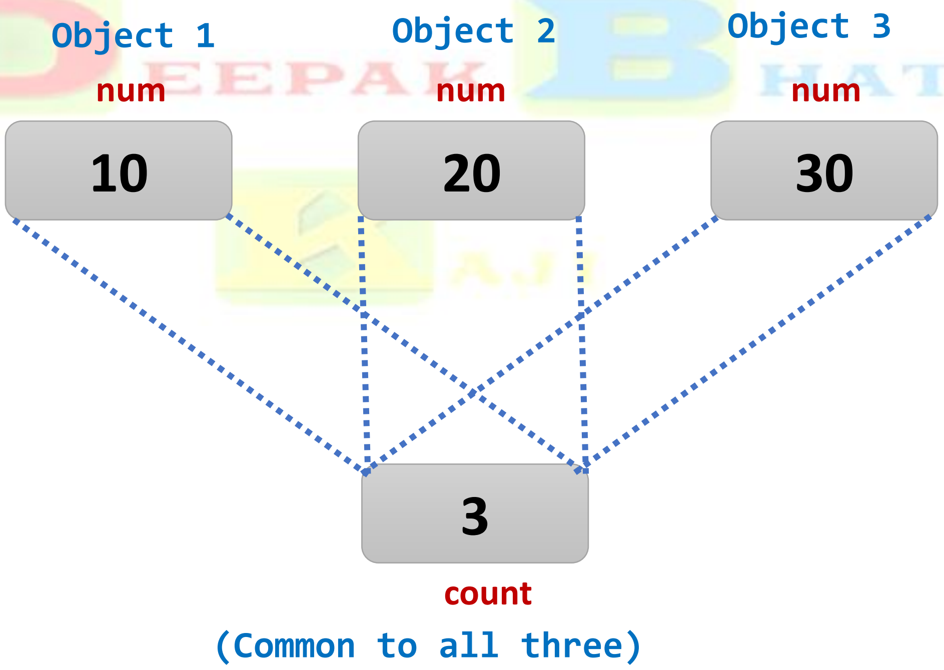


Figure: Sharing of a static data

Static Member Function:

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

A static function can have access to only other static members (functions or variables) declared in the same class.

A static member function can be called using the class name (instead of its objects) as follows:

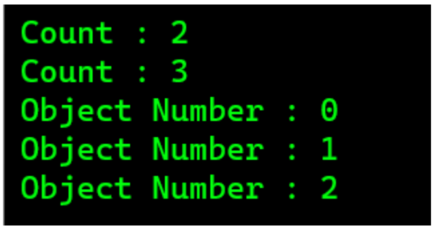
Class_name :: function_name;

If we create a member function of a class as a static called static member function. It is access only static data members. It is also accessible if we don't have any object of a class.

Example:

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

class Test
{
    int code;
    static int count;

public:
    void setcode()
    {
        code =count++;
    }
    void showCode()
    {
        cout<<"Object Number : " <<code<<endl;
    }
    static void showCount() //static Member function
    {
        cout<<"Count : "<<count<<endl;
    }

};

int Test :: count;

int main()
{
    Test obj1, obj2;

    obj1.setcode();
    obj2.setcode();

    Test :: showCount(); //accessing static function [you can access with
object as well]

    Test obj3;
    obj3.setcode();

    Test :: showCount();

    obj1.showCode();
    obj2.showCode();
    obj3.showCode();

    return 0;
}
```

Output:



```
Count : 2
Count : 3
Object Number : 0
Object Number : 1
Object Number : 2
```

16. Friend Function and Friend Class

In C++, a friend function or a friend class is a construct that allows external functions or classes to access private and protected members of another class. By declaring a function or class as a friend, it gains special access privileges to the private and protected members of the class it is declared as a friend of.

1. **Friend Function:** A friend function is a non-member function that is granted access to the private and protected members of a class. It is declared inside the class body using the **friend** keyword, but defined outside the class body like a regular function. The friend function can be called with an object of the class as an argument or can access the members using the scope resolution operator **::**.

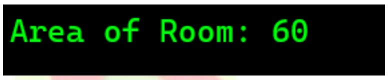
2. **Friend Class:** A friend class is a class that is granted access to the private and protected members of another class. To declare a class as a friend, the **friend** keyword is used within the class that wants to provide access. This allows the friend class to access private and protected members as if they were its own members.

Example Friend Function:

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

class Area
{
    int length;
    int width;
public:
    void setValues()
    {
        length=6;
        width=10;
    }
    friend int calculate(Area a);
};
```

```
int calculate(Area a)
{
    return a.length*a.width;
}
int main()
{
    Area obj;
    obj.setValues();
    cout<<"Area of Room : "<<calculate(obj)<<endl;

    return 0;
}
```

Output:



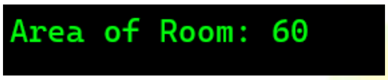
```
Area of Room: 60
```

Example Friend Class:

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

class Area {
private:
    int length;
    int width;

public:
    void setValues() {
        length = 6;
        width = 10;
    }

    friend class AreaCalculator;
};
```

```
class AreaCalculator {
public:
    int calculate(Area a) {
        return a.length * a.width;
    }
};

int main() {
    Area obj;
    obj.setValues();

    AreaCalculator calculator;
    cout << "Area of Room: " << calculator.calculate(obj) << endl;

    return 0;
}
```

Output:



```
Area of Room: 60
```

THE END