

# **CS-241L Object Oriented Programming**



**Lab Instructor: Mr. Usman Ahmed Raza**

**University of Engineering & Technology Lahore.  
(New Campus)**

## Rubrics for Lab

	<b>Advanced</b> <b>4</b>	<b>Proficient</b> <b>3</b>	<b>Approaching Proficiency</b> <b>2</b>	<b>Beginning</b> <b>1</b>
<b>Syntax</b> Ability to understand and follow the rules of the programming language.	Program compiles and contains no evidence of misunderstanding or misinterpreting the syntax of the language.	Program compiles and is free from major syntactic misunderstandings but may contain non-standard usage or superfluous elements.	Program compiles but contains errors that signal misunderstanding of syntax.	Program does not compile or contains typographical errors leading to undefined names.
<b>Logic</b> Ability to specify conditions, control flow, and data structures that are appropriate for the problem domain.	Program logic is correct, with no known boundary errors, and no redundant or contradictory conditions.	Program logic is mostly correct but may contain an occasional boundary error or redundant or contradictory condition.	Program logic is on the right track with no infinite loops, but shows no recognition of boundary conditions (such as <b>&lt;vs. &lt;=</b> )	Program contains some conditions that specify the opposite of what is required (less than vs. greater than), confuse Boolean AND/OR operators, or lead to infinite loops.
<b>Correctness</b> Ability to code formulae and algorithms that reliably produce correct answers or appropriate results.	Program produces correct answers or appropriate results for all inputs tested.	Program produces correct answers or appropriate results for most inputs.	Program approaches correct answers or appropriate results for most inputs but can contain miscalculations in some cases.	Program does not produce correct answers or appropriate results for most inputs.

	<b>Advanced</b>	<b>Proficient</b>	<b>Approaching Proficiency</b>	<b>Beginning</b>
	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>
<b>Completeness</b> Ability to apply rigorous case analysis to the problem domain.	Program shows evidence of excellent case analysis, and all possible cases are handled appropriately.	Program shows evidence of case analysis that is mostly complete but may have missed minor or unusual cases.	Program shows some evidence of case analysis but may be missing significant cases or mistaken in how to handle some cases.	Program shows little recognition of how different cases must be handled differently.
<b>Clarity</b> Ability to format and document code for human consumption.	Program contains appropriate documentation for all major functions, variables. Formatting, indentation, and other white space aids readability.	Program contains some documentation on major functions, variables. Indentation and other formatting are appropriate.	Program contains some documentation but has occasionally misleading indentation.	Program contains no documentation, or grossly misleading indentation.
<b>Modularity</b> Ability to decompose a problem into coherent and reusable functions, files, classes, or objects.	Program is decomposed into coherent and reusable units, and unnecessary repetition has been eliminated.	Program is decomposed into coherent units but may still contain some unnecessary repetition.	Program is decomposed into units of appropriate size, but they lack coherence or reusability. Program contains unnecessary repetition.	Program is one big function or is decomposed in ways that make little sense.

## Lab No. 1: Visual Studio guidelines and Array

CLO 4

### Visual Studio 2019 Installation

Welcome to Visual Studio 2019! In this version, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

#### Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

Check the [system requirements](#). These requirements help you know whether your computer supports Visual Studio 2019.

Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.

Reboot. The reboot ensures that any pending installs or updates don't hinder the Visual Studio install.

Free up space. Remove unneeded files and applications from your %SystemDrive% by, for example, running the Disk Cleanup app.

For questions about running previous versions of Visual Studio side by side with Visual Studio 2019, see the [Visual Studio 2019 Platform Targeting and Compatibility](#) page.

#### Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file. To do so, choose the following button to go to the Visual Studio download page. Select the edition of Visual Studio that you want and choose the Free trial or Free download button.

[Download Visual Studio](#)

#### Step 3 - Install the Visual Studio installer

Run the bootstrapper file you downloaded to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

From your Downloads folder, double-click the bootstrapper that matches or is similar to one of the following files:

vs\_community.exe for Visual Studio Community

vs\_professional.exe for Visual Studio Professional

vs\_enterprise.exe for Visual Studio Enterprise

If you receive a User Account Control notice, choose Yes to allow the bootstrapper to run.

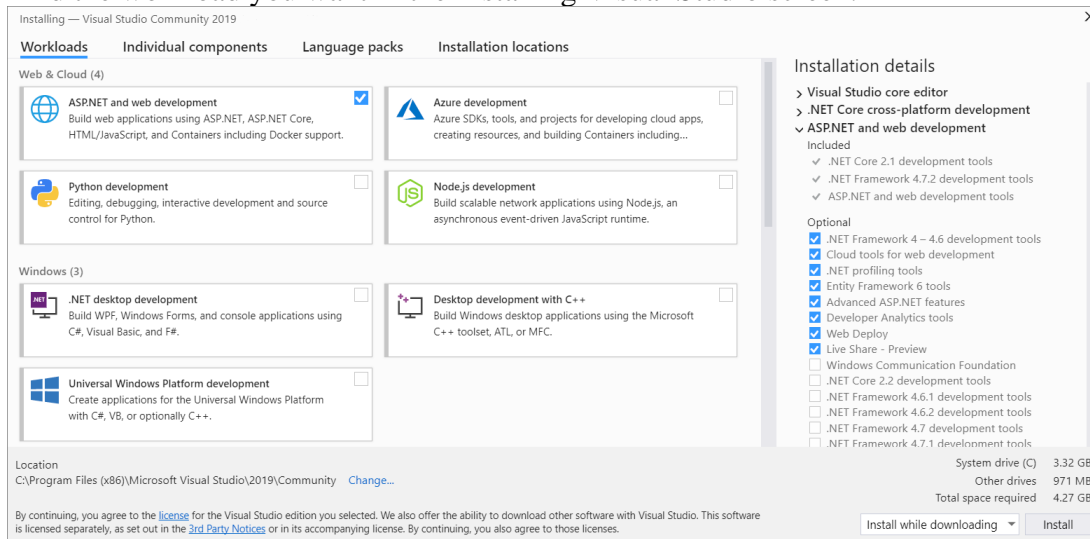
We'll ask you to acknowledge the Microsoft [License Terms](#) and the Microsoft [Privacy Statement](#).

Choose Continue.

#### Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting

the *workloads*, or feature sets, that you want. Here's how.  
Find the workload you want in the Installing Visual Studio screen.



For core C and C++ support, choose the "Desktop development with C++" workload. It comes with the default core editor, which includes basic code editing support for over 20 languages, the ability to open and edit code from any folder without requiring a project, and integrated source code control.

Additional workloads support other kinds of development. For example, choose the "Universal Windows Platform development" workload to create apps that use the Windows Runtime for the Microsoft Store. Choose "Game development with C++" to create games that use DirectX, Unreal, and Cocos2d. Choose "Linux development with C++" to target Linux platforms, including IoT development.

The Installation details pane lists the included and optional components installed by each workload. You can select or deselect optional components in this list. For example, to support development by using the Visual Studio 2017 or 2015 compiler toolsets, choose the MSVC v141 or MSVC v140 optional components. You can add support for MFC, the experimental Modules language extension, IncrediBuild, and more.

After you choose the workload(s) and optional components you want, choose Install.

Next, status screens appear that show the progress of your Visual Studio installation.

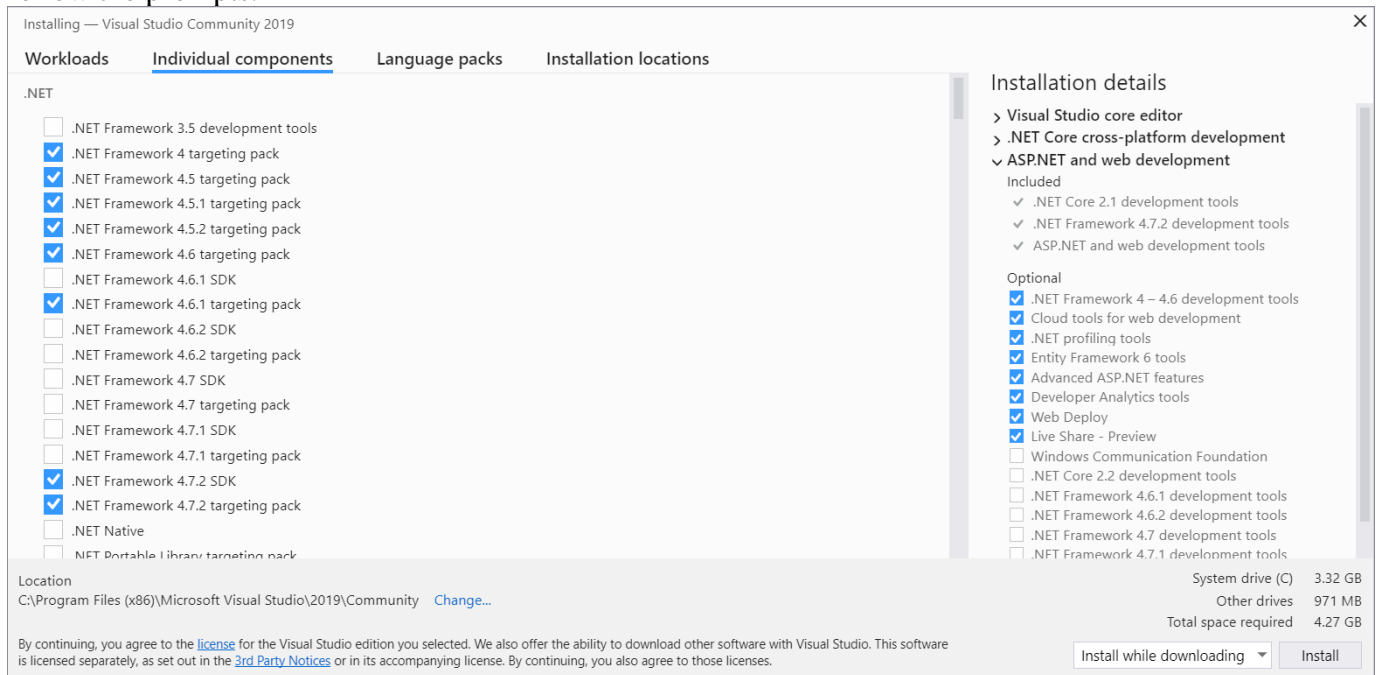
#### Tip

At any time after installation, you can install workloads or components that you didn't install initially. If you have Visual Studio open, go to Tools > Get Tools and Features... which opens the Visual Studio Installer. Or, open Visual Studio Installer from the Start menu. From there, you can choose the workloads or components that you wish to install. Then, choose Modify.

#### Step 5 - Choose individual components (Optional)

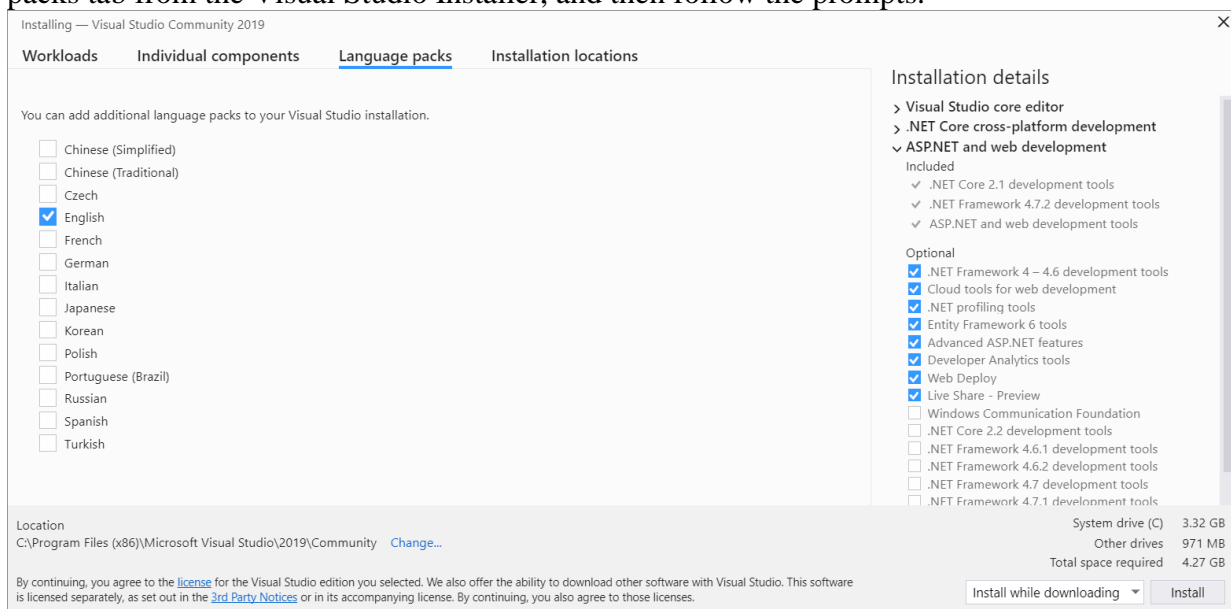
If you don't want to use the Workloads feature to customize your Visual Studio installation, or you want to add more components than a workload installs, you can do so by installing or adding

individual components from the Individual components tab. Choose what you want, and then follow the prompts.



## Step 6 - Install language packs (Optional)

By default, the installer program tries to match the language of the operating system when it runs for the first time. To install Visual Studio in a language of your choosing, choose the Language packs tab from the Visual Studio Installer, and then follow the prompts.

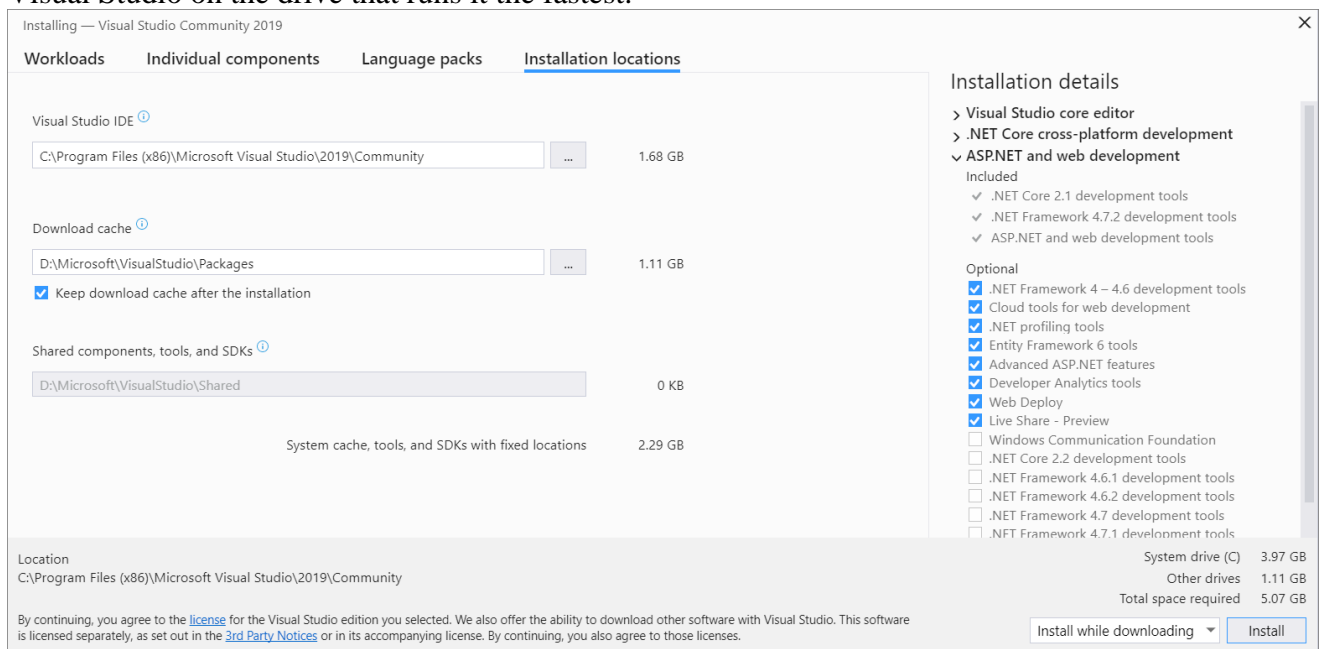


Change the installer language from the command line

Another way that you can change the default language is by running the installer from the command line. For example, you can force the installer to run in English by using the following command: `vs_installer.exe --locale en-US`. The installer will remember this setting when it's run the next time. The installer supports the following language tokens: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru, and tr-tr.

### Step 7 - Change the installation location (Optional)

You can reduce the installation footprint of Visual Studio on your system drive. You can choose to move the download cache, shared components, SDKs, and tools to different drives, and keep Visual Studio on the drive that runs it the fastest.



### Important

You can select a different drive only when you first install Visual Studio. If you've already installed it and want to change drives, you must uninstall Visual Studio and then reinstall it.

### Step 8 - Start developing

After Visual Studio installation is complete, choose the Launch button to get started developing with Visual Studio.

On the start window, choose Create a new project.

In the search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workload(s) that you chose during installation. To see different templates, choose different workloads.

You can also filter your search for a specific programming language by using the Language drop-down list. You can filter by using the Platform list and the Project type list, too.

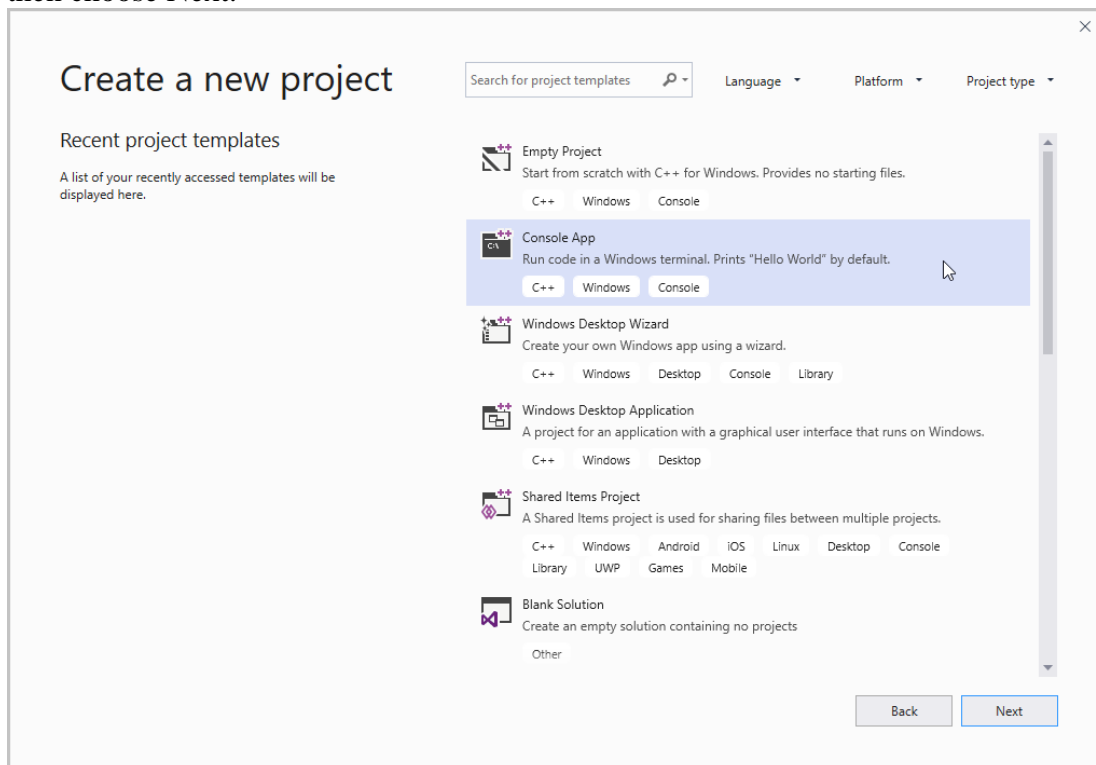
Visual Studio opens your new project, and you're ready to code!

When Visual Studio is running, you're ready to continue to the next step.

### Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It manages the relationship between all the project's files and any external files. To create your app, first, you'll create a new project and solution.

In Visual Studio, open the File menu and choose New > Project to open the Create a new Project dialog. Select the Console App template that has C++, Windows, and Console tags, and then choose Next.



In the Configure your new project dialog, enter *HelloWorld* in the Project name edit box. Choose Create to create the project.



## Configure your new project

Console App C++ Windows Console

Project name

HelloWorld

Location

C:\Users\username\source\repos

Solution name ⓘ

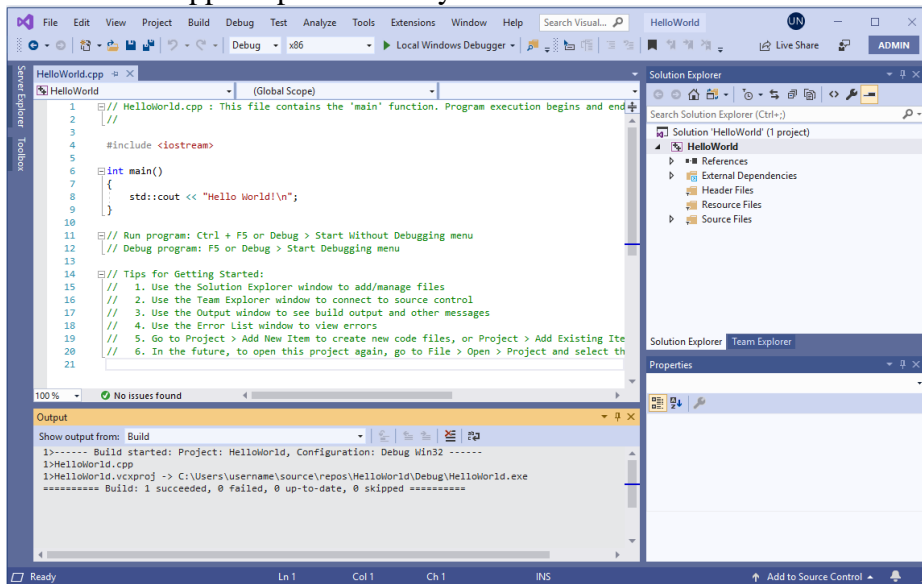
HelloWorld

☐ Place solution and project in the same directory

Back

Create

Visual Studio creates a new project. It's ready for you to add and edit your source code. By default, the Console App template fills in your source code with a "Hello World" app:



When the code looks like this in the editor, you're ready to go on to the next step and build your app.

Array

**Lab Task 1**

Write a program to input numbers in the array and calculate the average of all the numbers entered by the user.

**Lab Task 2**

Write a program to input 15 integers in the array. Now display amount of even and number of odd numbers in the array

**Lab Task 3**

Write a program to allow the user to store the temperature of the months of the year. Calculate the average temperature for the entire year and compare it to the average temperature of the previous year which is entered by the user. If the average temperature is greater than the previous year, then display “Global warming is present” otherwise display “no global warming”.

Expected Output

Enter average temperature of previous year: 37.5 Enter temperature for this month:23

Enter temperature for this month:28 Enter temperature for this month:32

...

The average temperature for this year is :38 Global Warming is present.

**Lab Task 4**

Write a program to enter the marks of midterm of a class of 15 students. Find out the average marks obtained by the entire class. Display the number of students whose midterm marks were equal or greater than the average of the class and the amount of students who obtained less than average of class.

Expected Output

Enter midterm marks of student1: 26 Enter midterm marks of student2: 34 Enter midterm marks of student3: 28 Enter midterm marks of student4: 31 Enter midterm marks of student5: 44 Enter midterm marks of student6: 41 Enter midterm marks of student7: 37 Enter midterm marks of student8: 41 Enter midterm marks of student9: 39 Enter midterm marks of student10: 32 Enter midterm marks of student11: 42 Enter midterm marks of student12: 12 Enter midterm marks of student13: 27

Enter midterm marks of student14: 6 Enter midterm marks of student15: 19

The average marks of the entire class is: 30.6

The number of students obtaining average or more marks are 9 The number of students obtaining below average marks are 6 Lab Task 5

(Linear Search)

15 numbers are entered into an array. The number to be searched is entered through the by another user. Write a program to find if the number to be searched is present in the array and if it is present, display the indexes at which the number is present and the number of times it appears in the array.

Expected Output

Array = 12, 14, 8,6, 21, 27, 12, 3, 9, 12, 8, 17, 12, 14, 11

Number to be searched:14 14 is present at index: 1

14 is present at index: 13

It is present 2 times in array Number to be searched:7 Not present in array Number to be searched:12 12 is present at index: 0

12 is present at index: 6

12 is present at index: 9

12 is present at index: 12

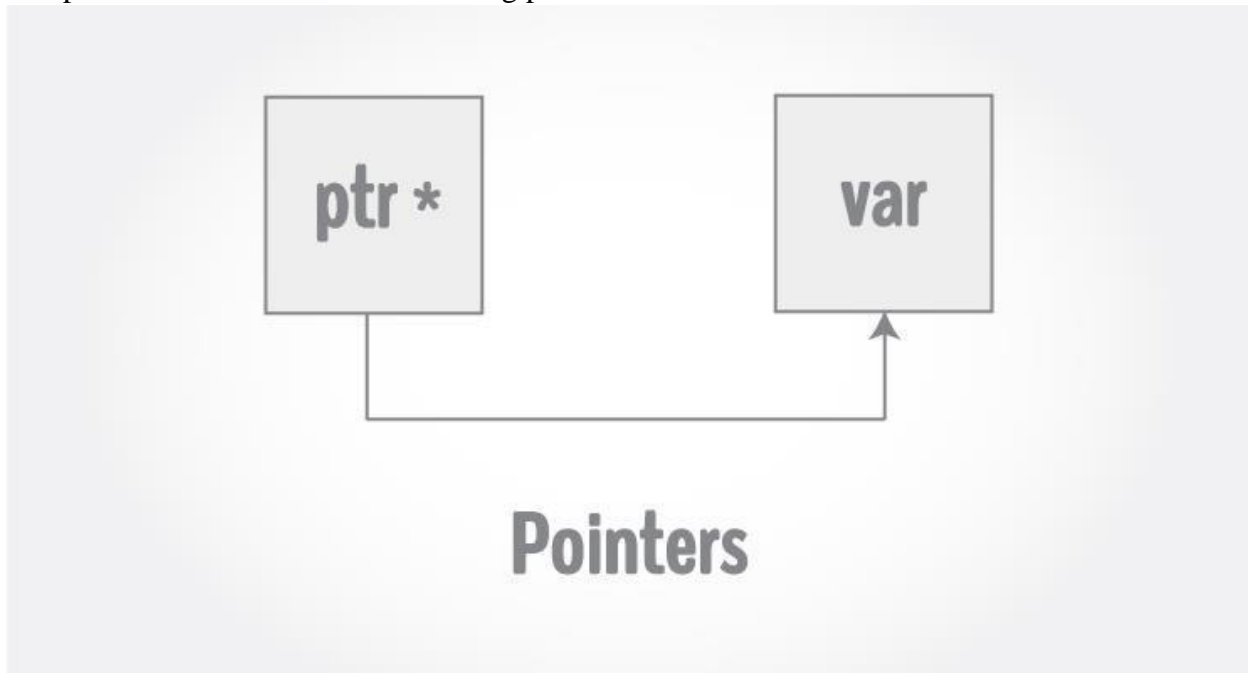
It is present 4 times in array

## Lab No. 2: Pointers and command line arguments

CLO4

### C++ Pointers

In this lab, you'll learn everything about pointers. You'll learn how values are stored in the computer and how to access them using pointers.



Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python.

Pointers are used in C++ program to access the memory and manipulate the address.

#### Address in C++

To understand pointers, you should first know how data is stored on the computer.

Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.

To know where the data is stored, C++ has an & operator. The & (reference) operator gives you the address occupied by a variable.

If `var` is a variable then, `&var` gives the address of that variable.

#### Example 1: Address in C++

```
1. int main()
2. {
3.     int var1 = 3;
4.     cout << &var1 << endl;
5. }
```

## Output

```
0x7fff5fbff8ac
```

**Note:** You may not get the same result on your system.

---

## Pointers Variables

C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.

Pointers variables are variables that points to a specific address in the memory pointed by another variable.

---

## How to declare a pointer?

```
int *p;
```

OR,

```
int* p;
```

The statement above defines a pointer variable *p*. It holds the memory address

The asterisk is a dereference operator which means **pointer to**.

Here, pointer *p* is a **pointer to int**, i.e., it is pointing to an integer value in the memory address.

---

## Reference operator (&) and Deference operator (\*)

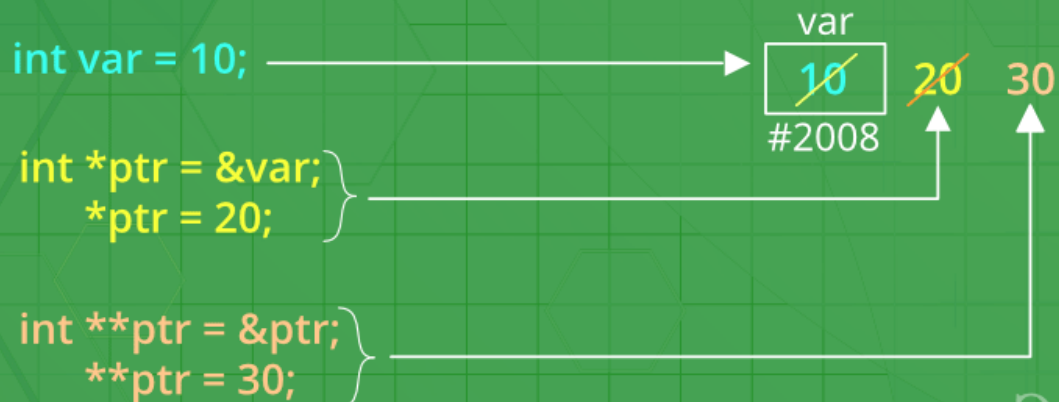
Reference operator (&) as discussed above gives the address of a variable.

To get the value stored in the memory address, we use the dereference operator (\*). **For example:** If a *number* variable is stored in the memory address **0x123**, and it contains a value **5**.

The **reference (&)** operator gives the value **0x123**, while the **dereference (\*)** operator gives the value **5**.

**Note:** The (\*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

# How pointer works in C



## Example: C++ Pointers

C++ Program to demonstrate the working of pointer.

```
1. int main()
2. {
3.     int *pc, c;
4.
5.     c = 5;
6.     cout << "Address of c: " << &c << endl;
7.     cout << "Value of c: " << c << endl;
8.
9.     pc = &c;    // Pointer pc holds the memory address of variable c
10.    cout << "Address that pointer pc holds "<< pc << endl;
11.    cout << "Content of the address pointer pc holds " << *pc << endl;
12.
13.    c = 11;    // The content inside memory address &c is changed from 5 to
14.    11.
15.    cout << "Address pointer pc holds " << pc << endl;
16.    cout << "Content of the address pointer pc holds: " << *pc << endl;
17.
18.    *pc = 2;
19.    cout << "Address of c: " << &c << endl;
20.    cout << "Value of c: " << c << endl;
21.
22.    return 0;
23. }
```

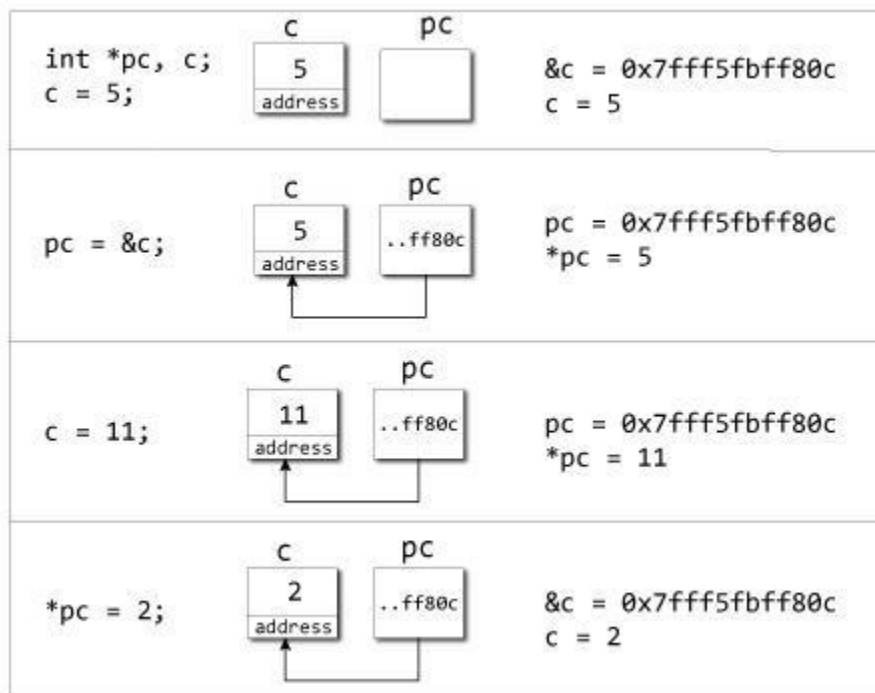
**Output**

Address of c: 0x7fff5fbff80c  
Value of c: 5

Address that pointer pc holds: 0x7fff5fbff80c  
Content of the address pointer pc holds: 5

Address pointer pc holds: 0x7fff5fbff80c  
Content of the address pointer pc holds: 11

Address of c: 0x7fff5fbff80c  
Value of c: 2



### Common mistakes when working with pointers

Suppose you want pointer pc to point to the address of c. Then,

```
int c, *pc;
```

```
pc=c; /* Wrong! pc is address whereas, c is not an address. */
```

```
*pc=&c; /* Wrong! *pc is the value pointed by address whereas, &c is an address.  
*/
```

```
pc=&c; /* Correct! pc is an address and, &pc is also an address. */
```

```
*pc=c; /* Correct! *pc is the value pointed by address and, c is also a value. */
```

In both cases, pointer pc is not pointing to the address of c.

### **C++ Call by Reference: Using pointers**

This method used is called passing by value because the actual value is passed. There is another way of passing an argument to a function where the actual value of the argument is not passed. Instead, only the reference to that value is passed.

Example: Passing by reference using pointers

```
int main()
{
    int a = 1, b = 2;
    swap(&a, &b);
    cout << a << endl;
    cout << b << endl;
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

### **Output**

```
a = 1
b = 2
```

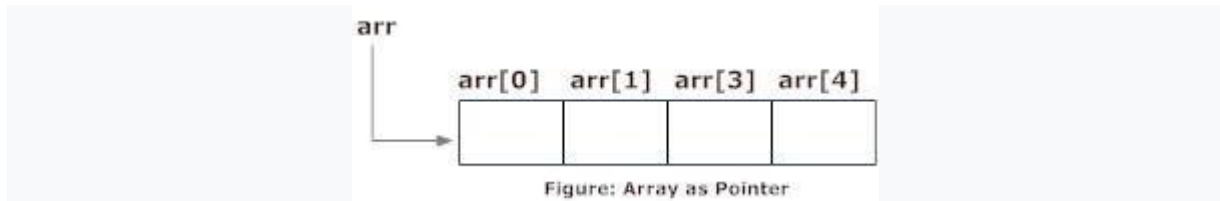
### **C++ Pointers and Arrays**

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:



```
int* ptr;  
  
int a[5];  
  
ptr = &a[2]; // &a[2] is the address of third element of a[5].
```



Since ptr points to the third element.

### C++ Program to insert and display data entered by using pointer notation.

```
int main() {  
    float arr[5];  
  
    // Inserting data using pointer notation  
    cout << "Enter 5 numbers: ";  
    for (int i = 0; i < 5; ++i) {  
        cin >> *(arr + i) ;  
    }  
  
    // Displaying data using pointer notation  
    cout << "Displaying data: " << endl;  
    for (int i = 0; i < 5; ++i) {  
        cout << *(arr + i) << endl ;  
    }  
  
    return 0;  
}
```

### Output

```
Enter 5 numbers: 2.5  
3.5  
4.5
```

```
5
2
Displaying data:
2.5
3.5
4.5
5
2
```

### Command line arguments:

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems. To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

For better understanding run this code on your linux machine.

```
// Name of program mainreturn.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    cout << "You have entered " << argc
        << " arguments:" << "\n";

    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";

    return 0;
}
```

## Lab Task:

**Question 1:** Declare three integers x, y and sum and three pointers xPtr, yPtr, sumPtr. Point three pointers to their respective variable.

Take input in x and y using xPtr and yPtr. Do not use direct references i.e. x and y integers Add x and y and save the result in sum. Do not use direct references i.e. x, y and sum integers

Print addition's result. For example, if user entered x = 5 and y = 9 your program should print: 5

+ 9 = 14. Do not use direct references i.e. x, y and sum integers

**Note:** You have to do all the processing using pointers i.e. indirect references to variables.

**Question 2:** Write a program to find the factorial of a number using pointers

**Question 3:** Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int i = 0; i < 10; i++) cout << array[i] << endl;
```

**Question 4:** Write a function which will take pointer and display the number on screen. Take number from user and print it on screen using that function.

**Question 5:** Write a C++ program to accept five integer values and print the elements of the array in reverse order using a pointer.

**Question 6:** Make an array of 6 elements, find the largest number from the array and show the value and address of that element using pointer

## Lab No. 3: Getters, Setters and constructors

---

**Write Getters, Setters and different type of constructors, Constructor Overloading, and Implement Access Modifiers**

**CLO 1, 4**

### Encapsulation

The meaning of Encapsulation is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get and set methods**.

### Access Private Members

To access a private attribute, use public "get" and "set" methods:

### Example

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

```
class Employee { private:  
    // Private attribute int salary;
```

```
public:
```

```
// Setter
```

```
void setSalary(int s) { salary = s;  
}
```

```
// Getter
```

```
int getSalary() { return salary;  
}
```

```
};
```

```
int main() { Employee myObj;  
myObj.setSalary(50000); cout << myObj.getSalary(); return 0;  
}
```

**Constructors:** Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:

Default Constructor

Parameterized Constructor

**Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.

**Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

Example

```
class default_construct
{
public:
int id;
//Default Constructor default_construct()
{
cout << "Default Constructor called" << endl; id=-1;
}
};
```

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.

Problem Set:

With the help of comments in the code, explain it and modify it for 2 parameters in the constructor. Write down the output of the code.

```
#include <stdio.h> class param_construct
{
public:
int id;
//Parameterized Constructor Param_construct(int x)
{
cout << "Parameterized Constructor called" << endl; id=x;
}
};
int main() {

// obj1 will call Parameterized Constructor Param_construct obj1(20);
cout << "Param id is: " <<obj1.id << endl; return 0;
}
```

**Constructor Overloading:**

Making constructor with same name but different parameters is called Constructor overloading. Or making the default and parameterized constructor with same name is also called Constructor overloading.

Example:

```
#include<iostream> #include<string> Using namespace std;
class person{ private:
string name; int age; float CGPA; public:
person()    //default constructor
{
name= " ";
age =0;
CGPA= 0.0;
}
person(inta) //parameterized constructor with same name i.e constructor overloading
{
age=a;
}
void setAge(int a)
{
if (a>0) age=a;
}
int getAge()
{
return age;
}
```

Lab Task:

Rewrite the above given codes and see the results.

Write a class naming Book and display the following properties of 2 books using Getters and Setters:

Book Name

Book Price

Total pages in the book

Write a program having class Time, by using Getters and Setters you should be able to set the hours, minutes and seconds and display the final time in the main function.

*Hint: One bigger setter function within which all 3 setters will be called and that bigger setter will be called within the constructor.*

Make a calculator having basic functions of addition, multiplication, subtraction and division and displays the result.

Make a class of Student that shows the name, registration number and CGPA of 5 students. 6- Write a program that takes value of height, length and breadth of a box and displays its volume.

## Lab No. 4: Array of Objects, Inline function, Constructor Overloading and Pointer to

### Objects

CLO 1, 4

#### Target

#### Pointers to Objects:

So far, we have studied about the pointer data members, but You can also define pointers to class objects. For example, the following statement defines a pointer variable named rectPtr:

```
Rectangle *rectPtr;
```

The rectPtr variable is not an object, but it can hold the address of a Rectangle object. The following code shows an example.

```
Rectangle myRectangle; // A Rectangle object
Rectangle *rectPtr; // A Rectangle pointer
rectPtr = &myRectangle; // rectPtr now points to myRectangle
```

#### **Static and Dynamic memory allocation:**

Static allocated memory is the one that is allocated before the program runs, while in dynamically allocated memory we can use exactly as much as we need.

#### **Example 1:**

```
Int i;
a[3]=97; //statically allocated memory
aptr[3]=87;
```

Class object pointers can be used to dynamically allocate objects. The following code shows an example.

#### **Example 2:**

```
// Define a Rectangle pointer.
Rectangle *rectPtr;
//Dynamically allocate a Rectangle object.
rectPtr = new Rectangle;
//Store values in the object's width and length.
rectPtr->setWidth(10.0);
rectPtr->setLength(15.0);
```

#### **Inline Member Functions:**

When the body of a member function is written inside a class declaration, it is declared inline.

#### **Example 3:**

```
#include<iostream.h>
using namespace std;
inline int max(int x, int y)
{
    return ((x>y) ? x:y);
}
Int main()
{
```



```

int a,b;
cout<<"Enter two Values";
cin>>a>>b;
cout<<"Greater value is"<<max(a,b);
return 0;
}

```

### **Array of Objects:**

As with any other data type in C++, you can define arrays of class objects. An array of InventoryItem objects could be created to represent a business's inventory records. Here is an example of such a definition:

```
InventoryItem inventory[40];
```

This statement defines an array of 40 InventoryItem objects. The name of the array is inventory, and the default constructor is called for each object in the array. If you wish to define an array of objects and call a constructor that requires arguments, you must specify the arguments for each object individually in an initializer list. Here is an example:

```
InventoryItem inventory[3] = {"Hammer", "Wrench", "Pliers"};
```

#### **Example 4**

### **InventoryItem.h**

```

// Specification file for the InventoryItem class.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <cstring> // Needed for strlen and strcpy
// InventoryItem class declaration.
class InventoryItem
{
private:
    char *description; // The item description
    double cost; // The item cost
    int units; // Number of units on hand
public:
    // Constructor
    InventoryItem(char *desc, double c, int u)
    { // Allocate just enough memory for the description.
        description = new char [strlen(desc) + 1];

        // Copy the description to the allocated memory.
        strcpy(description, desc);

        // Assign values to cost and units.
        cost = c;
        units = u;}
}

```

```

// Destructor
~InventoryItem()
{ delete [] description; }

const char *getDescription() const
{ return description; }

double getCost() const
{ return cost; }

int getUnits() const
{ return units; }
};
#endif

```

## Program

```

// This program demonstrates an array of class objects.
#include <iostream>
#include "InventoryItem.h"
using namespace std;

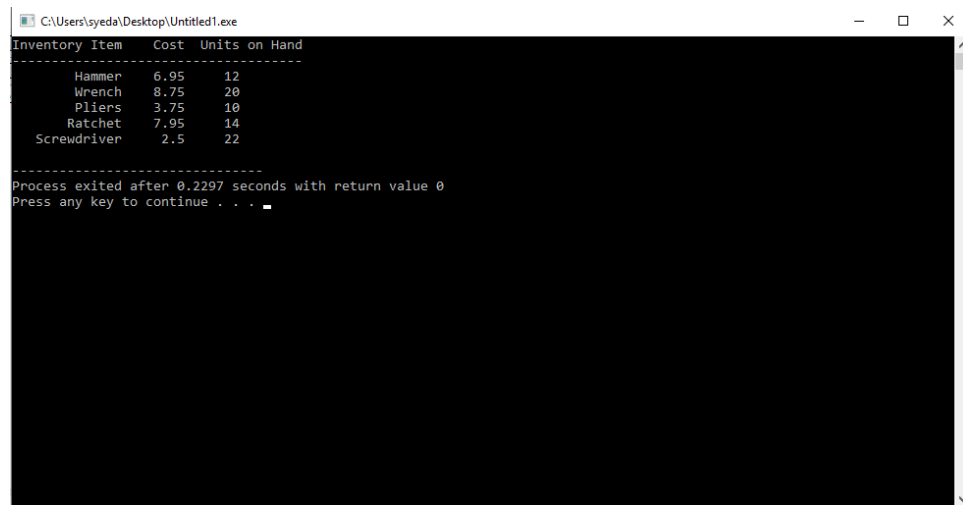
int main()
{
    const int NUM_ITEMS = 5;
    InventoryItem inventory[NUM_ITEMS] = {
        InventoryItem("Hammer", 6.95, 12),
        InventoryItem("Wrench", 8.75, 20),
        InventoryItem("Pliers", 3.75, 10),
        InventoryItem("Ratchet", 7.95, 14),
        InventoryItem("Screwdriver", 2.50, 22) };
    cout << setw(14) << "Inventory Item"
    << setw(8) << "Cost" << setw(8)
    << setw(16) << "Units on Hand\n";
    cout << ".....\n";

    for (int i = 0; i < NUM_ITEMS; i++)
    {
        cout << setw(14) << inventory[i].getDescription();
        cout << setw(8) << inventory[i].getCost();
        cout << setw(7) << inventory[i].getUnits() << endl;
    }

    return 0;
}

```

## Output



```
C:\Users\syeda\Desktop\Untitled1.exe
Inventory Item  Cost  Units on Hand
-----
Hammer        6.95   12
Wrench        8.75   20
Pliers        3.75   10
Ratchet       7.95   14
Screwdriver    2.5    22
-----
Process exited after 0.2297 seconds with return value 0
Press any key to continue . . .
```

## Lab Tasks

- 1- Two parties want to buy a house, one party is interested in knowing the total area and price of the house while the second party wants to know about the number of rooms as well. Design a code that takes only one class but fulfils the requirements of both parties.

*Hint: Use Constructor Overloading*

- 2- Using dynamic memory allocation, enter marks (*float*) of subjects of a student and calculate GPA of that student.
- 3- Get name and Employee ID of 3 employees of a company and by using pointer to object display them in the main function.
- 4- Design a class that has an array of floating-point numbers. The constructor should accept an integer argument and dynamically allocate memory to the array to hold that many numbers. In addition, there should be member functions to perform the following operations:

- Store a number in any element of the array
- Retrieve a number from any element of the array
- Return the highest value stored in the array
- Return the lowest value stored in the array
- Return the average of all the numbers stored in the array

*Hint: Use array of Objects*

## Lab No. 5 : Constructor, Destructor and Copy Constructor

---

**Target:**

**CLO 1,4**

**Constructors:** Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:

- **Default Constructor**
- **Parameterized Constructor**
- **Copy Constructor**

**Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.

**Copy Constructors:** A copy constructor is a member function which initializes an object using another object of the same class.

**Default Constructors:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

### **Example**

```
class student
{
    public:
    int id;
    //Default Constructor
    student()
    {
        cout << "Default Constructor called" << endl;
        id=-1;
    }
    Student(int a)
    {
        Id=a;
    }
};
Int main()
{
    Student obj;
    Student obj(10);
}
```

**Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created.

**Problem Set:**

With the help of comments in the code, explain it and modify it for 2 parameters in the constructor. Write down the output of the code.

```
#include <stdio.h>
class param_construct
{
    public:
    int id;
    //Parameterized Constructor
    Param_construct(int x)
    {
        cout << "Parameterized Constructor called" << endl;
        id=x;
    }
};
int main() {

    // obj1 will call Parameterized Constructor
    Param_construct obj1(20);
    cout << "Param id is: " <<obj1.id << endl;
    return 0;
}
```

**Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here –

```
classname (const classname &obj) {

    // body of constructor

}
```

Here, obj is a reference to an object that is being used to initialize another object.

### Problem Set:

Elaborate the use of copy constructor in the code given below, write down the usage and execution dry run on the paper. Write down the output of the code.

```
#include<iostream>
class Point
{
    private:
        int x, y;
```

```

public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()      { return x;}
    int getY()      { return y;}
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}

```

### When is copy constructor called?

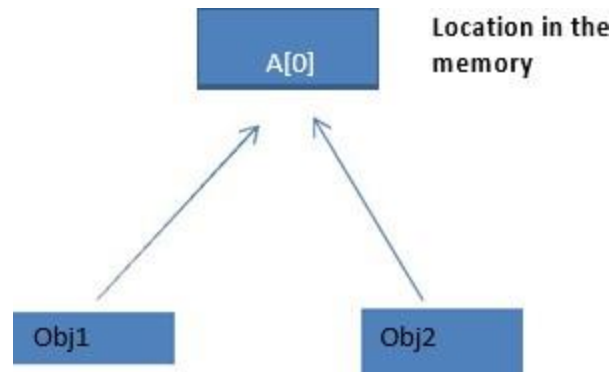
In C++, a Copy Constructor may be called in following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.

### Shallow Copy Constructor

The concept of shallow copy constructor is explained through an example. Two students are entering their details in excel sheet simultaneously from two different machines shared over a network. Changes made by both of them will be reflected in the excel sheet. Because same excel sheet is opened in both locations. This is what happens in shallow copy constructor. Both objects will point to same memory location.

Shallow copy copies reference to original objects. The compiler provides a default copy constructor. Default copy constructor provides a shallow copy as shown in below example. It is a bit-wise copy of an object. Shallow copy constructor is used when class is not dealing with any dynamically allocated memory.



In the below example you can see both objects, c1 and c2, points to same memory location. When `c1.concatenate()` function is called, it affects c2 also. So both `c1.display()` and `c2.display()` will give same output.

### Deep Copy Constructor

Let's consider an example for explaining deep copy constructor. You are supposed to submit an assignment tomorrow and you are running short of time, so you copied it from your friend. Now you and your friend have same assignment content, but separate copies. Therefore, any modifications made in your copy of assignment will not be reflected in your friend's copy. This is what happens in deep copy constructor.

Deep copy allocates separate memory for copied information. So, the source and copy are different. Any changes made in one memory location will not affect copy in the other location. When we allocate dynamic memory using pointers, we need user defined copy constructor. Both objects will point to different memory locations.



*Contents of FeetInches.h (Version 1)*

*// The FeetInches class holds distances or measurements  
// expressed in feet and inches.*

```

class FeetInches
{
private:
int feet; // To hold a number of feet
  
```

```

int inches; // To hold a number of inches
void simplify(); // Defined in FeetInches.cpp
public:
// Constructor
FeetInches(int f = 0, int i = 0)
{ feet = f;
  inches = i;
  simplify(); }

// Mutator functions
void setFeet(int f)
{ feet = f; }

void setInches(int i)
{ inches=i;
  simplify(); }

// Accessor functions
int getFeet() const
{ return feet; }

int getInches() const
{ return inches; }
};

#endif

```

Contents of FeetInches.cpp (Version 1)

```

// Implementation file for the FeetInches class
#include "FeetInches.h"
void FeetInches::simplify()
{
  if (inches >= 12)
  {
    feet += (inches / 12);
    inches = inches % 12;
  }
  else if (inches < 0)
  {
    feet -= ((abs(inches) / 12) + 1);
    inches = 12 - (abs(inches) % 12);
  }
}

```



### Lab Tasks:

#### Question 1- FeetInches Class Copy Constructor and multiply Function

Add a copy constructor to the FeetInches class. This constructor should accept a FeetInches object as an argument. The constructor should assign to the feet attribute the value in the argument's feet attribute and assign to the inches attribute the value in the argument's inches attribute. As a result, the new object will be a copy of the argument object.

**Question 2-** Create a BankAccount class having Balance as an attribute and add copy constructor to it. The constructor should accept a BankAccount object as an argument. It should assign to the balance field, the value in the argument's balance field. As a result, the new object will be copy of the argument object

**Question 3-** Write a program having class Date, by using Getters and Setters you should be able to set the day,month,year and display the final Date in the main function.

*Hint: One bigger setter function within which all 3 setters will be called and that bigger setter will be called within the constructor.*

*Use copy constructor for task1 and task2*

#### Question 4

Design and Implement a class clockType that manages time. The class clockType should store *hour, minutes, and seconds* and display it in the following format **(12:30:5)**. For any object of clockType, your program should be able to perform the following operations on that object.

1. Set the time
2. Get the time
3. Display the time
4. Increment the hour by 1
5. Increment the minute by 1
6. Increment the second by 1
7. Create another object of clockType and copy it with the object you created earlier;.And see the values in both objects.  
See they are copy or not.

#### Question 5

A class that represents a Rectangle, it will have the following member variables and methods:

- height
- width
- area
- getArea(int,int) - returns the area of the rectangle[Area = Width \* Height]

Write a program to create object of Rectangle. Then ask the user to enter height and width of object.

Then create another object which will be a copy of the first object. Carefully write the main function.

## Lab No. 6

---

Target:

CLO4

**Static members, “this” pointer, Arrow operator, Dynamic memory allocation with “new” operator de-allocation of object mem with delete operator.**

### **Static Members of a C++ Class**

We can define class members static using static keyword. 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. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created if no other initialization is present.

The declaration of a static data member in the member list of a class is not a definition. You must define the static member outside of the class declaration, in namespace as by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
```

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class X exist even though the static data member X::i has been defined.

**‘this’ Pointer:** It is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

To understand ‘this’ pointer, it is important to know that how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All access the same function definition as present in the code segment. Meaning each object gets its own copy of data members and all objects share single copy of member functions.
3. Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated? Compiler supplies an implicit pointer along with the functions names as ‘this’. The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.

For a class X, the type of this pointer is ‘X\* const’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X \*const’

### **Example 1:**

1) When local variable's name is same as member's name

```
#include<iostream.h>
/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};
int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output:

x = 20

### Example 2:

Here you can see that we have two data members num and ch. In member function setMyValues() we have two local variables having same name as data members name. In such case if you want to assign the local variable value to the data members then you won't be able to do until unless you use this pointer, because the compiler won't know that you are referring to object's data members unless you use this pointer. This is one of the example where you must use **this** pointer.

```
#include <iostream>
using namespace std;
class Demo {
private:
    int num;
    char ch;
public:
    void setMyValues(int num, char ch){
```

```

    this->num = num;
    this->ch = ch;
}
void displayMyValues(){
    cout<<num<<endl;
    cout<<ch;
}
};
int main(){
    Demo obj;
    obj.setMyValues(100, 'A');
    obj.displayMyValues();
    return 0;
}

```

## **Arrow Operator**

When using **dynamic allocation of objects**, we use pointers, both to single object and to arrays of objects. Here is a good rule of thumb: *For pointers to single objects, arrow operator is easiest.*

As if you have a pointer to an object, the pointer name would have to be dereferenced first, to use the dot-operator [example: (\*fp1).Show();]. Arrow operator is a nice shortcut, avoiding the use or parentheses to force order of operations:

### **Example 1:**

```

fraction *fp1, *fp2;    // Declaration of pointer objects
fp1 = new Fraction;     // uses
                        // default constructor

```

```

fp2 = new Fraction(3,5); // uses constructor with two parameters

```

```

fp1->Show();             // equivalent to (*fp1).Show();
fp2->GetNumerator();
fp2->Input();

```

An object which can be created at run-time is referred to as dynamic object. A dynamic object can be created using 'new' keyword, as follow:

```

className *ptr;
ptr = new className;

```

The new operator returns the address of the object created, and it is stored in the pointer ptr. The variable ptr is a pointer object of the same class. The member variable of the object can be accessed using the pointer and -> (arrow) operator. A dynamic object can be destroyed using the 'delete' operator as follow:

```

delete ptr;

```

## **Memory Allocation**

The process of allocating and deallocating memory space in a better way is called memory management. There are two ways that memory gets allocated for data storage:

### **1. Compile Time (or static) Allocation**

- Memory for named variables is allocated by the compiler
- Exact size and type of storage must be known at compile time
- For standard array declarations, this is why the size has to be constant

### **2. Dynamic Memory Allocation**

- Memory allocated "on the fly" during run time.
- dynamically allocated space usually placed in a program segment known as the heap or the free store
- Exact amount of space or number of items does not have to be known by the compiler in advance.
- For dynamic memory allocation, pointers are crucial

#### **Important note:**

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

#### **How is it different from memory allocated to normal variables?**

For normal variables like "int a", "char str[10]", etc. memory is automatically allocated and deallocated. For dynamically allocated memory like "int \*p = new int[10]", it is programmers' responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

#### **How is memory allocated/deallocated in C++?**

C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

To allocate memory of any data type, the syntax is:

*pointer-variable = new data-type;*

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

*// Pointer initialized with NULL*

*// Then request memory for the variable*

*int \*p = NULL;*

*p = new int;*

*OR*

*// Combine declaration of pointer*

*// and their assignment*

*int \*p = new int;*

We can also initialize the memory using new operator:

*pointer-variable = new data-type(value);*

*Example:*

```
int *p = new int(25);
float *q = new float(75.25);
```

Allocate block of memory:

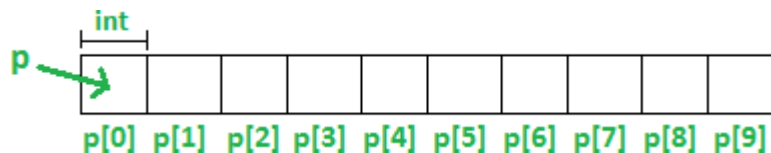
new operator is also used to allocate a block(an array) of memory of type data-type.

*pointer-variable = new data-type[size];*

where size(a variable) specifies the number of elements in an array. **Example:**

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



**delete operator:**

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

**Syntax:**

*// Release memory pointed by pointer-variable*

*delete pointer-variable;*

Here, pointer-variable is the pointer that points to the data object created by new.

**Examples:**

*delete p;*

*delete q;*

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

*// Release block of memory*

*// pointed by pointer-variable*

*delete[] pointer-variable;*

**Example:**

*// It will free the entire array pointed by p.*

*delete[] p;*

## Problem Set:

1. Write a C++ that creates four objects with roll\_num, name and marks as attributes of class student. Each object of class must be assigned a unique roll\_num. (using roll\_number as static data member).

**Print the output of roll\_num of each object.**

<b>Student</b>
<b>Data Member:</b> name (character array), marks (float), roll_num (int)
<b>Methods:</b> <b>Student( ):</b> (Constructor used to null and zero values of attributes). <b>Setdata()</b> (a type of void function used to take value of attributes)  <b>display() :</b> (a type of void function used to display value of attributes )

**2. Predict the output of following programs. If there are compilation errors, then fix them.**

```
#include<iostream.h>
class Test
{
private:
    int x;
public:
    Test(int x = 0) { this->x = x; }
    void change(Test *t) { this = t; }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj(5);
    Test *ptr = new Test (10);
    obj.change(ptr);
    obj.print();
    return 0;
}
```

**3. With the help of your understanding about arrow pointer and dynamic objects, explain the code on paper and predict the output of the following code.**

```
#include<iostream.h>
class data
```



```

{
    int x,y;
public:
    data()           // default constructor
    {
        x=10;
        y=50;
    }
    void display()
    {
        cout<<"\n x="<<x;
        cout<<"\n y="<<y;
    }
};
void main()
{
    data *d;         // declaration of object pointer
    d=new data;       // dynamic object
    d->display();
}

```

4. Write a program in which student array dynamically allocated and another marks array in which marks of students are stored. Find the average of students and highest marks of a student.
5. Write a C++ program in which user declare two-dimensional array dynamically. Find row sum and columns sum of all rows and columns of array.

# Lab No. 7

---

**Target:**

**CLO2**

**Static member variable and static function, friend function, operator overloading**

**Static Function Members:**

By declaring a function member as static, you make it independent of any object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

## **Friend Function**

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

Friend function declaration can appear anywhere in the class. But a good practice would be where the class ends. An ordinary function that is not the member function of a class has no privilege to access the private data members, but the friend function does have the capability to access any private data members. The declaration of the friend function is very simple. The keyword friend in the class prototype inside the class definition precedes it. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

```
class className

{ ... ..

    friend return_type functionName(class name);

    ... .. }

return_type functionName(argument/s)

{ ... .. }
```

```

    // Private and protected data of className can be accessed from
    // this function because it is a friend function of className.
    .....}

```

No friend keyword is used in the definition.

The advantage of encapsulation and data hiding is that a non-member function of the class cannot access a member data of that class. Care must be taken using friend function because it breaks the natural encapsulation, which is one of the advantages of object-oriented programming. It is best used in the operator overloading.

### **Addition of members of two different classes using friend Function**

```

#include <iostream>

using namespace std;

// forward declaration

class B;

class A {
    private:
        int numA;
    public:
        A(){
            numA=10;}

    // friend function declaration
    friend int add(A, B);
};

class B {
    private:
        int numB;
    public:
        B() {

```

```

        numb=20 }

// friend function declaration

friend int add(A , B); ////////////////it means that it is friend function of class A
and B

};

// Function add() is the friend function of classes A and B

// that accesses the member variables numA and numB and objects are passing as
arguments of friend function

int add(A objectA, B objectB)

{

int sum;

sum=objectA.numA+ objectB.numB

return sum;

}

int main()

{

A objectA;

B objectB;

cout<<"Sum: "<< add(objectA, objectB);

return 0;

}

```

## **Operator overloading**

**Operator overloading:** is an important concept in C++. The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used. However, for user-defined types (like: objects), you can redefine the way operator works. For example: If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings. This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

To overload an operator, a special operator function is defined inside the class as:

```
ClassName operator - (ClassName c2) ←
{
    ... ..
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... ..
    result = c1-c2;
    ... ..
}
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

### **Important points related to Operator Overloading:**

- Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
- Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
- Operator overloading cannot change the precedence and associativity of operators. However, if you want to change the order of evaluation, parenthesis should be used.
- There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .\* (member selection through pointer to function) and ?: (ternary operator).
- 

### **Problem Set:**

#### **Friend function.**

1. Write a C++ program to swap the values two integer members of different classes using **friend function**.

Pseudo code:

classA
Data Member: value1 (integer)
Methods: setdata( ): (a type of void function used to take value of value1 (integer)) display() : (a type of void function used to display value of value1 before swapping.) exchange (class1 ,class2 ) : Friend Function (swap and display values after swapping)

ClassB
Data Member: value2 (integer)
Methods: setdata( ): (a type of void function used to take value of value2(integer)) display() : (a type of void function used to display value of value2 before swapping.) exchange (class1 ,class2 ) :Friend Function (swap and display values after swapping)

b) Write a C++ program to swap the values two integer members of different classes using friend function (Use call by reference method).

2. Write a C++ program for addition of two complex numbers using **friend function** (use constructor function to initialize data members of complex class).

Complex
Int x Int img
Complex(): (default constructor with zero values of data members) complex( x, img) : (Parametrized constructor with given inputs at the time of object creation) complex sum( complex , complex) :(friend function ) show(complex ): display complex number of objects of class Complex.

**Hint:** The -> is called the **arrow operator**. It is formed by using the minus sign followed by a greater than sign. Simply saying: To access members of a structure, use the **dotoperator**. To access members of a structure through a pointer, use the **arrow operator**.

### Operator Overloading

3. Write a C++ program to perform matrix 2x2 **addition and subtraction using operator overloading concept.**
4. Create a class called **Martix** that represents a 3x3 matrix. Create a constructor for initializing the matrix with 0 values. Create another overloaded constructor for

initializing the matrix to the values sent from outside. **Overload the \* and / operators** for multiplication and division of two matrices. \*operator for finding the product of the two matrices. Define all the member functions outside the class.

5. Create a class called **Time** that has separate int member data for hours, minutes, and seconds. Provide the following member functions for this class:
- a) A **no-argument constructor** to initialize hour, minutes, and seconds to 0.
  - b) A **3-argument constructor** to initialize the members to values sent from the calling function at the time of creation of an object. Make sure that valid values are provided for all the data members. In case of an invalid value, set the variable to 0.
  - c) A member function **show** to display time in 11:59:59 format.
  - d) An overloaded **operator+** for addition of two Time objects. Each time unit of one object must add into the corresponding time unit of the other object. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.
  - d) An overloaded **operator-** for subtraction of two Time objects. one time unit of one object must be subtracted into the corresponding time unit of the other object. subtract minutes from minutes, seconds from seconds, hours from hours. Keep in view the fact that minutes and seconds of resultant should not exceed the maximum limit (60). If any of them do exceed, subtract 60 from the corresponding unit and add a 1 to the next higher unit.

A main() programs should create two initialized Time objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third Time variable. Finally it should display the value of this third variable.

# Lab No. 8

---

Target:

CLO3

**Operator overloading binary and unary operator, assignment operator, operator function as member functions and standalone functions, object conversion**

**Operator Overloading:** Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

## **Example Overloading "<<" Operator**

```
#include< iostream.h>
#include<    conio.h>
class time
{
int hr,min,sec;
public:
time()
{
hr=0, min=0; sec=0;
}
time(int h,int m, int s)
{
hr=h, min=m; sec=s;
}
friend ostream& operator << (ostream &out, time &tm); //overloading '<<' operator
};
ostream& operator<< (ostream &out, time &tm)    //operator function
{
out << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";
return out;
}
void main()
{
time tm(3,15,45);
cout << tm;
}
```

## **Example Overloading Relational Operators**

You can also overload Relational operator like `==` , `!=` , `>=` , `<=` etc. to compare two user-defined object.

class time



```

{
int    hr,min,sec;
public:
time()
{
    hr=0, min=0; sec=0;
}

time(int h,int m, int s)
{
    hr=h, min=m; sec=s;
}
friend bool operator==(time &t1, time &t2); //overloading '==' operator
};
bool operator== (time &t1, time &t2)          //operator function
{
return ( t1.hr == t2.hr &&
        t1.min == t2.min &&
        t1.sec == t2.sec );
}

```

**Binary Operator Overloading:** Feature in C++ programming that allows programmer to redefine the meaning of an **operator** (when they operate on class objects) is known as **operator overloading**.

#### **Example Overloading '+' Operator**

```

#include <iostream>
using namespace std;
class Box {
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
public:
    double getVolume(void) {
        return length * breadth * height;
    }
    void setLength( double len ) {
        length = len;
    }
    void setBreadth( double bre ) {
        breadth = bre;
    }
    void setHeight( double hei ) {
        height = hei;
    }
}

```

// Overload + operator to add two Box objects.

```

Box operator+(const Box& b) {
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}
};

// Main function for the program
int main() {
    Box Box1;          // Declare Box1 of type Box
    Box Box2;          // Declare Box2 of type Box
    Box Box3;          // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 specification
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // volume of box 1
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume << endl;

    // volume of box 2
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume << endl;

    // Add two object as follows:
    Box3 = Box1 + Box2;

    // volume of box 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume << endl;

    return 0;
}

```

### Operator functions that are class member functions

If you write an operator function as a member function, then it automatically has access to all the member variables and functions of the class. Friend declaration not needed! But the complication is that the left-hand-side operand becomes the "implicit argument" of the function - it is "this" object, the current one being worked on. So, the member operator function has only one argument, the right-hand-side operand. For example, the overload operator function for + becomes:

```
class CoinMoney
{
...//your code...

CoinMoney operator+ (CoinMoney rhs) //member function
{
CoinMoney sum;
sum.nickels = nickels + rhs.nickels;
etc
return sum;
}
...
};
```

The "nickels" is the member variable in the left-hand-side operand, "this" object. The function has only one parameter, the right-hand-side operand. Because this function is a member of the class, it has direct access to the member variables in "this" current object, and dot access to the member variables in the other objects in the same class.

Since a *member* operator function is just an ordinary *member* function, the following statements do the same thing:

```
m3 = m1 + m2;
m3 = m1.operator+ (m2);
```

The member version of an operator function is called to work on the left-hand operand, with the right-hand operand being the function argument. Again, calling operator functions explicitly is legal, but rare, and usually pointless.

### Object Conversion

**CONCEPT: Special operator functions may be written to convert a class object to any other type.**

As you've already seen, operator functions allow classes to work more like built-in data types. Another capability that operator functions can give classes is automatic type conversion. Data type conversion happens "behind the scenes" with the built-in data types. For instance, suppose a program uses the following variables:

```
int i;
double d;
```

The statement below automatically converts the value in *i* to a floating-point number and stores it in *d*:

```
d = i;
```

Likewise, the following statement converts the value in *d* to an integer (truncating the fractional part) and stores it in *i*:

```
i = d;
```

The same functionality can also be given to class objects. For example, assuming distance is a FeetInches object and *d* is a double, the following statement would conveniently convert distance's value into a floating-point number and store it in *d*, if FeetInches is properly written:

```
d = distance;
```

To be able to use a statement such as this, an operator function must be written to perform the conversion. Here is the code for the operator function that converts a FeetInches object to a double:

```
FeetInches::operator double()
```

```
{  
    double temp = feet;  
    temp += (inches / 12.0);  
    return temp;  
}
```

This function contains an algorithm that will calculate the decimal equivalent of a feet and inches measurement. For example, the value 4 feet 6 inches will be converted to 4.5. This value is stored in the local variable *temp*. The *temp* variable is then returned.

```
// This program demonstrates the the FeetInches class's  
// conversion functions.  
#include <iostream>  
#include "FeetInches.h"  
using namespace std;  
int main()  
{  
    double d; // To hold double input  
    int i; // To hold int input  
    // Define a FeetInches object.  
    FeetInches distance;  
    // Get a distance from the user.  
    cout << "Enter a distance in feet and inches:\n";  
    cin >> distance;  
    // Convert the distance object to a double.  
    d = distance;  
    // Convert the distance object to an int.  
    i = distance;  
    // Display the values.  
    cout << "The value " << distance;  
    cout << " is equivalent to " << d << " feet\n";
```

```

    cout << "or " << i << " feet, rounded down.\n";
    return 0;
}

```

### Output:

Enter a distance in feet and inches:

Feet: 8 [Enter]

Inches: 6 [Enter]

The value 8 feet, 6 inches is equivalent to 8.5 feet or 8 feet, rounded down.

### Problem Set

#### Question:1.

In the **Time** class (created in last lab) do the following :

- a) Overloaded operators for **pre- and post- increment**. These increment operators should add a 1 to the **seconds** unit of time. Keep track that **seconds** should not exceed 60.
- b) Overload operators for **pre- and post- decrement**. These decrement operators should subtract a 1 from **seconds** unit of time. If number of seconds goes below 0, take appropriate actions to make this value valid.

#### Question:2.

Create a class called **Fraction** for performing arithmetic with fractions. Write a driver program to test your class. Use integer variables to represent the private data of the class, the numerator and the denominator. Provide a constructor function that enables an object of this class to be initialized when it is declared.

The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form (use a utility function to reduce) (i.e., the fraction 2/4 would be stored in the object as 1 in the numerator and 2 in the denominator).

You have to implement functionalities to:

1. Add fractions using the operator +
2. Subtract fractions using the operator -
3. Multiply fractions using the operator \*
4. Divide fractions using the operator /
5. Increment fractions using post-increment and pre-increment operators ++
6. Decrement fractions using post-decrement and pre-decrement operators --
7. Take negative of fractions using operator unary -. Also overload the fraction unary +
8. Display fractions using the operator <<
9. Input fractions using the operator >>
10. Copy contents of a fraction in another fraction using operator =

## Lab No. 9

---

**Target:**

**CLO2**

### **Composition, Aggregation, Association**

**Composition:** In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc. Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called object composition.

To qualify as a **composition**, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

A good real-life example of a composition is the relationship between a person's body and a heart. Let's examine these in more detail.

Composition relationships are part-whole relationships where the part must constitute part of the whole object. For example, a heart is a part of a person's body. The part in a composition can only be part of one object at a time. A heart that is part of one person's body cannot be part of someone else's body at the same time.

In a composition relationship, the object is responsible for the existence of the parts. Most often, this means the part is created when the object is created and destroyed when the object is destroyed. But more broadly, it means the object manages the part's lifetime in such a way that the user of the object does not need to get involved. For example, when a body is created, the heart is created too. When a person's body is destroyed, their heart is destroyed too. Because of this, composition is sometimes called a "death relationship".

And finally, the part doesn't know about the existence of the whole. Your heart operates blissfully unaware that it is part of a larger structure. We call this a **unidirectional** relationship, because the body knows about the heart, but not the other way around.

Note that composition has nothing to say about the transferability of parts. A heart can be transplanted from one body to another. However, even after being transplanted, it still meets the requirements for a composition (the heart is now owned by the recipient and can only be part of the recipient object unless transferred again).

**Example:**

```

#include<iostream>
using namespace std;
class Engine
{
public:
    int power;
};

class Car
{
public:
    Engine e;
    string company;
    string color;
    void show_details()
    {
        cout<<"Compnay is: "<<<company<<<endl;
        cout<<"Color is: "<<<color<<<endl;
        cout<<"Engine horse power is: "<<<e.power;
    }
};

int main()
{
    Car c;
    c.e.power = 500;
    c.company = "hyundai";
    c.color = "black";
    c.show_details();
    return 0;
}

```

**Output:**

Company is: Hyundai

Color is: black

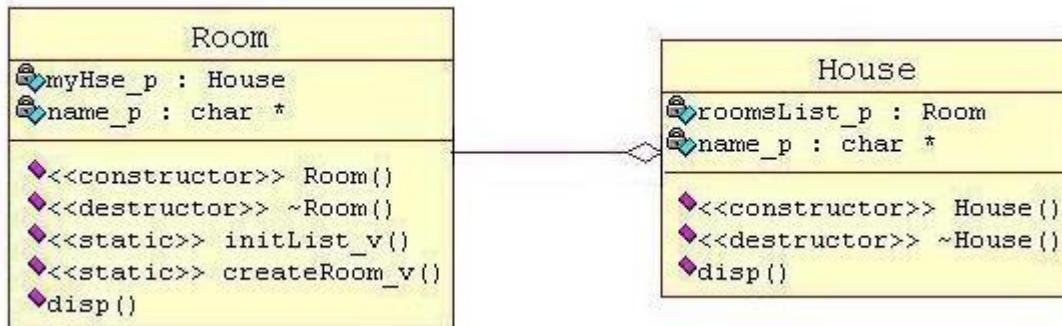
Engine horse power is: 500

**Relationship in Composition:**

House can contain multiple rooms there is no independent life for room and any room can not belong to two different house. If we delete the house room will also be automatically deleted.

Here is respective Model for the above example.

### Room class has Composition Relationship with House class

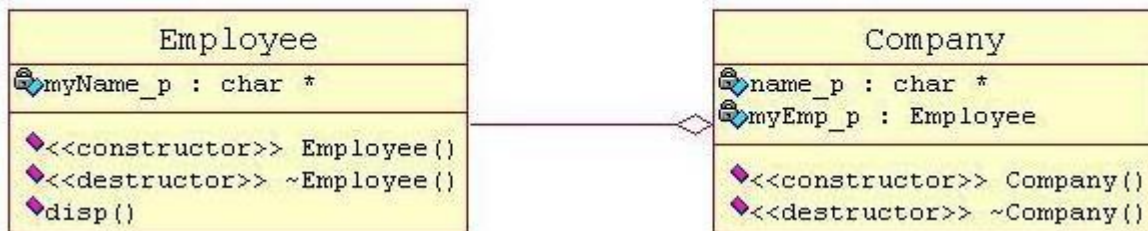


### Aggregation:

Aggregation is a specialized form of association between two or more objects in which each object has its own life cycle but there exists an ownership as well. Aggregation is a typical whole/part or parent/child relationship, but it may or may not denote physical containment. An essential property of an aggregation relationship is that the whole or parent (i.e. the owner) can exist without the part or child and vice versa.

As an example, an employee may belong to one or more departments in an organization. However, if an employee's department is deleted, the employee object would not be destroyed but would live on. Note that the relationships between objects participating in an aggregation cannot be reciprocal—i.e., a department may “own” an employee, but the employee does not own the department.

### Employee class has Agregation Relationship with Company class



Example:

```

#include<iostream.h>

class Employee
{
public:

    Employee(char    *name){
        cout<<"Employee::ctor\n";
    }
};
  
```



```

        myName_p = new char(sizeof(strlen(name)));
        myName_p = name;
    }

    char* disp(){return(myName_p);};

    ~Employee()
    {
        cout<<"Employee:dtor\n\n";
        delete (myName_p);
    }

private:
    char *myName_p;
};

class Company
{
public:
    Company(char * compName, Employee* emp){
        cout<<"Company::ctor\n";
        name = new char(sizeof(strlen(compName)));
        name = compName;
        myEmp_p = emp;
    };

    ~Company()
    {
        cout<<"Company:dtor\n\n";
        myEmp_p = NULL;
    };

private:
    char *name;
    Employee *myEmp_p;
};

int main()
{
    cout<<"\nExample of Aggregation Relationship \n";
    cout<<".....\n\n";

    {
        cout<<"Here, an Employee-Keerti works for Company-MS \n";
        Employee emp("Keerti");
    }
}

```

```

{
    Company comp("MS", &emp);
} // here Company object will be deleted, whereas Employee object is still there

cout<<"At this point Company gets deleted...\n";
cout<<"\nBut Employee-"<<emp.disp(); cout<<"
is still there\n";

} //here Employee object will be deleted

return(0);
}

```

output:

-----

Example of Aggregation Relationship

-----

Here, an Employee-Keerti works for Company-MS

Employee::ctor

Company::ctor

Company:dtor

At this point Company gets deleted...

But Employee-Keerti is still there

Employee:dtor

## Association:

Association is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.

Association is a semantically weak relationship (a semantic dependency) between otherwise unrelated objects. An association is a “using” relationship between two or more objects in which the objects have their own lifetime and there is no owner.

As an example, imagine the relationship between a doctor and a patient. A doctor can be associated with multiple patients. At the same time, one patient can visit multiple doctors for treatment or consultation. Each of these objects has its own life cycle and there is no “owner” or parent. The objects that are part of the association relationship can be created and destroyed independently.

References:

1. <https://www.infoworld.com/article/3029325/exploring-association-aggregation-and-composition-in-oop.html>
2. <https://www.go4expert.com/articles/association-aggregation-composition-t17264/>

### **Lab Tasks:**

- 1- Show association between Bank class and its Employees class. The program should display the name of person as a Bank employee.
- 2- Make a library class that can have no. of books on same or different subjects. Identify the type of relation and create array of books with title and author names of each book.  
Hint: If Library gets destroyed then All books within that particular library will be destroyed. i.e. book cannot exist without library.
- 3- A car has an engine, at most 2 AC's, a handle to control the gears and a brake. If the car is manual, it has a clutch, otherwise it doesn't. Make 3 constructors of each class, a default and 2 parameterized. Call the overloaded constructors of the composed and aggregated classes from the container class.
- 4- A cup of tea consists of Milk, Tea, Sugar and Water. Code this scenario, again call the overloaded constructors of the composed and aggregated classes from the container class.
- 5- A pond has ducks and water in it.

# Lab No. 10

---

**Target:**

**CLO2**

**Composition, Aggregation, Association (UML diagrams)**

## **UML diagram:**

UML stands for Unified Modelling Language. It is actually a modern way of making models of software. Through UML diagrams we represent different software components, their working and relationship with other components. This approach is also used in modelling of business process.

## **Types of UML:**

There are two broad categories of UML diagrams that consist of all other types. Following are those types and their subtypes.

### **Behavioural UML Diagram:**

- Activity Diagram
- Use Case Diagram
- Interaction Overview Diagram
- Timing Diagram
- State Machine Diagram
- Communication Diagram
- Sequence Diagram

### **Structural UML Diagram:**

- Class Diagram
- Object Diagram
- Component Diagram
- Composite Structure Diagram
- Deployment Diagram
- Package Diagram
- Profile Diagram

In this lab we will focus on Class Diagrams. This is because OOP is based on classes and their relations within them.

In class diagram we specify attributes (data members) and their behaviour (member functions).

### **Representation of Class diagram:**

Class diagrams are represented by a box having 3 partitions. The top partition consists the name of your class, the middle part is for the attributes and the bottom part contains the member functions.

It is not necessary to have all the fields filled (as there may not be any data member or data function in any class) but still the class should have three partitions. You can leave that part blank.

What we call in oop    What we call in UML    How we show in UML



### UML Diagram Example 1:

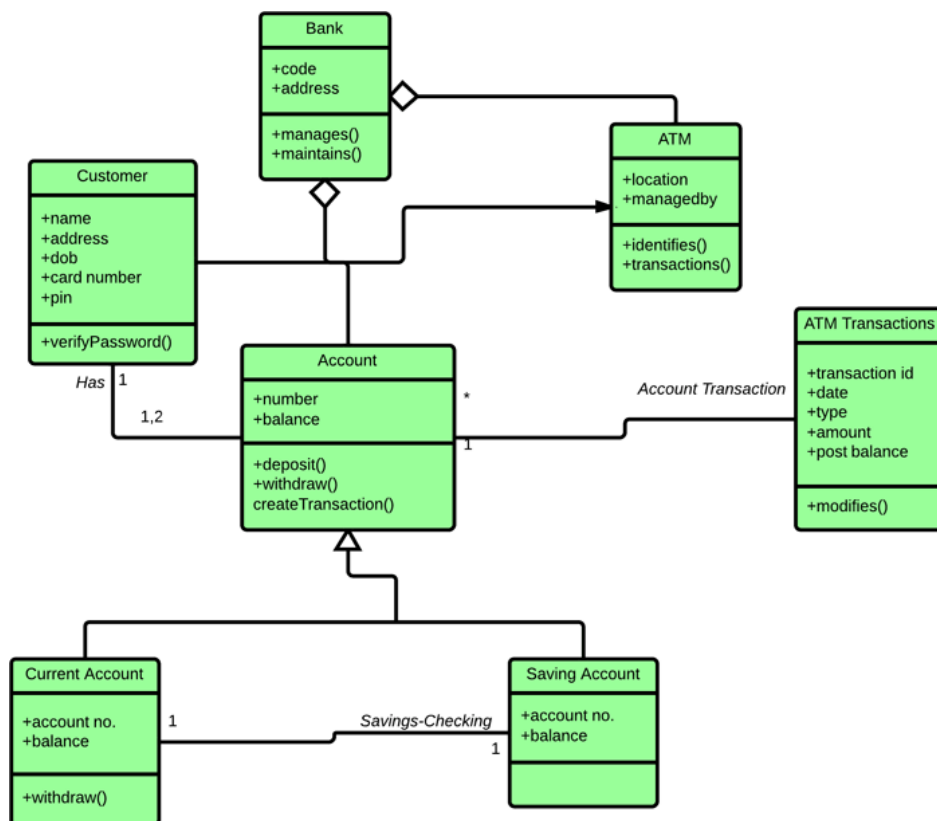
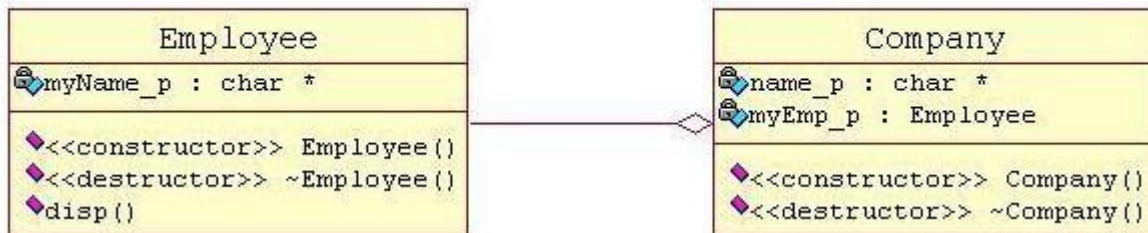


Figure 1 ATM's system [2]

## Example 2:

**Employee class has Agregation Relationship with Company class**



Example:

```

#include<iostream.h>

class Employee
{
public:

    Employee(char    *name){
        cout<<"Employee::ctor\n";
        myName_p = new char(sizeof(strlen(name)));
        myName_p = name;
    }

    char* disp(){return(myName_p);};

    ~Employee()
    {
        cout<<"Employee:dtor\n\n";
        delete (myName_p);
    }

private:
    char *myName_p;
};

class Company
{
public:
    Company(char    *    compName,    Employee*    emp){
        cout<<"Company::ctor\n";
    }
}
    
```

```

        name = new char(sizeof(strlen(compName)));
        name = compName;
        myEmp_p = emp;
    };

```

```

~Company()
{
    cout<<"Company:dtor\n\n";
    myEmp_p = NULL;
};

```

```

private:
    char        *name;
    Employee *myEmp_p;
};

```

```

int main()
{
    cout<<"\nExample of Aggregation Relationship \n";
    cout<<".....\n\n";

    {
        cout<<"Here, an Employee-Keerti works for Company-MS \n";
        Employee emp("Keerti");

        {
            Company comp("MS", &emp);
        } // here Company object will be deleted, whereas Employee object is still there

        cout<<"At this point Company gets deleted...\n";
        cout<<"\nBut Employee-"<<emp.disp();
        cout<<" is still there\n";

    } //here Employee object will be deleted

    return(0);
}

```

output:

-----

Example of Aggregation Relationship

-----

Here, an Employee-Keerti works for Company-MS  
Employee::ctor  
Company::ctor  
Company:dtor

At this point Company gets deleted...

But Employee-Keerti is still there  
Employee:dtor

References:

1. <https://app.createely.com/diagram/FetuN6gCs2H/>
2. <https://www.guru99.com/uml-class-diagram.html>

### **Lab Tasks:**

**Draw UML only from task 1 to 5.**

**Task 1:** A car has engine, AC, brakes, gear and clutch.

**Task 2:** A pond has ducks and water in it.

**Task 3:** A Hospital has 2 departments, Surgical and Medical. Surgical Department deals with surgeries and medical department deals with prescribing medicines. Each patient has a unique ID. The surgical department has 3 operation theatres available for surgeries and has 20 rooms which can be allocated to patients. The medical department has no rooms for patients; it deals with outdoor patients only. Surgical patients may go through many surgeries if required. There are doctors who perform surgeries. These doctors in the hospital are either surgeons or non-surgeons. A surgeon performs surgeries and non-surgeons check patients in the medical department.

**Task 4:** A cup of tea consists of Milk, Tea, Sugar and Water.

**Task 5:** A personal computer has a CPU, a motherboard, a RAM, a GPU, a HDD, a Display and a Sound, along with other additional components. Each of these components is an object in itself. Thus, a Personal computer is a complex object composed of several smaller level objects. Each of the smaller objects has its own attributes and functions. For example:



Object	Attributes	Object	Attributes
<b>CPU</b>	Model and Make Frequency Serial Number etc.	<b>Motherboard</b>	Model and Make Front Side Bus Built-in options (Sound, modem, VGA, NIC, ...) etc.
<b>RAM</b>	Model and Make Frequency Space Technology etc.	<b>GPU</b>	Internal memory Memory type Frequency Bus Interface No of cores etc.
<b>HDD</b>	Model and Make RPM Capacity Buffer	<b>Sound</b>	Model and Make No of channels Input Options (analog, digital, MIDI, mic)

6. Khursheed wants to book a ticket from Faisal Movers. He can book tickets online or in person. In both ways, he is provided with the schedule of bus, number of available seats and finally the payment. When the payment is completed the customer is given a confirmation.

Draw a UML diagram for the whole scenario of Khursheed making a reservation/booking. Make possible classes, their attributes and functions and then show their relations in diagram and code as well.

7. You are given the UML diagram of Hospital Management system. Write a program according to the given scenario

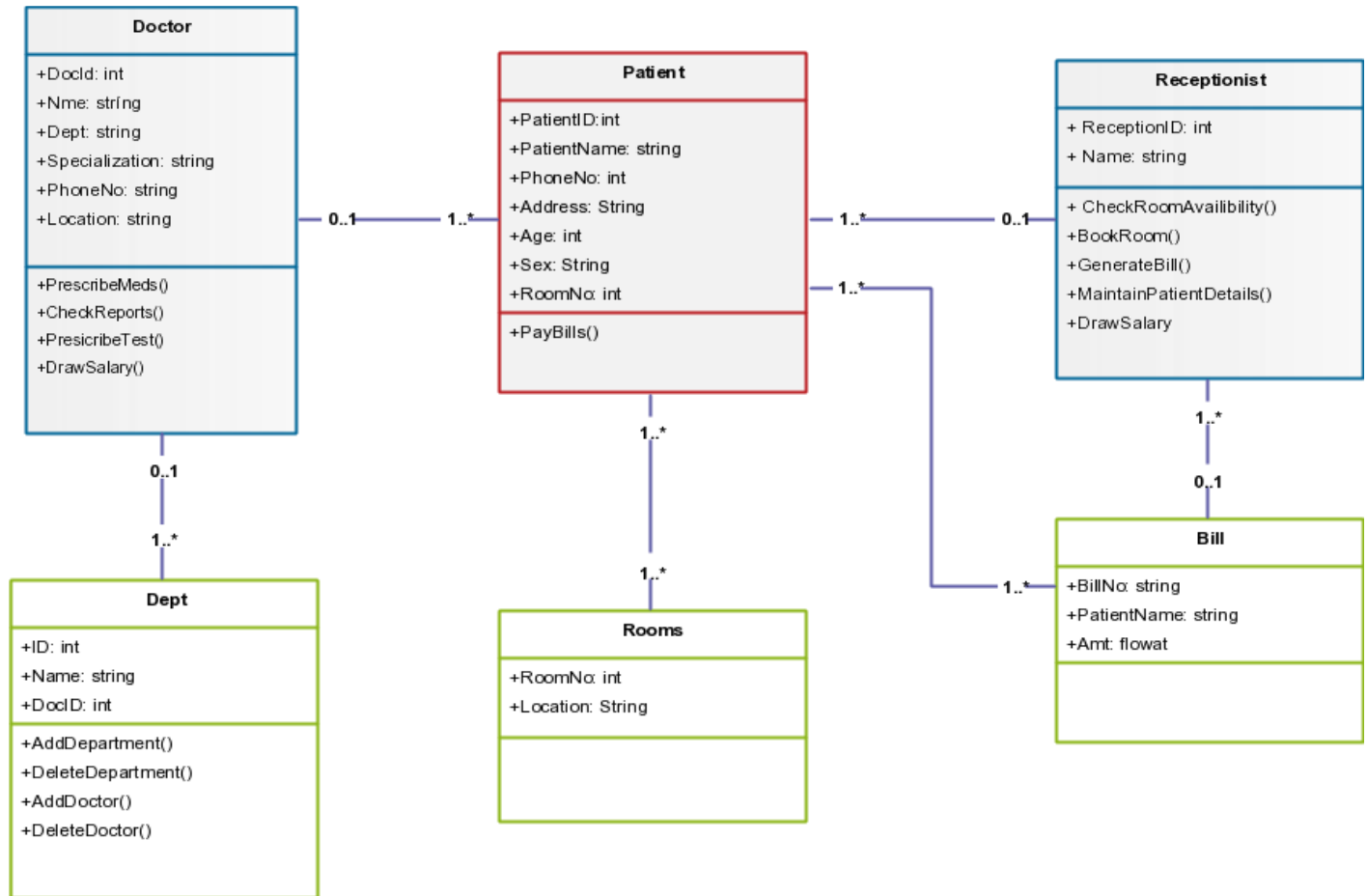


Figure 2 Hospital Management System [1]

8. Similar to the Ticket Booking System (task 1), create a UML diagram and write program for “Hotel Management system” according to your own understanding. The classes can include following: Receptionist, Rooms, Customer, Bill, Food Items, Stock, Chef, Manager (or others if you want).

# Lab No. 11

---

**Target:**

**CLO1**

## **Inheritance**

**Inheritance:** The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

**Derived Class:** The class that inherits properties from another class is called Sub class or Derived Class.

**Base Class:** The class whose properties are inherited by sub class is called Base Class or Super class.

### **Example 1:**

Implementing Inheritance  
using namespace std;

```
//Base class
class Parent
{
    public:
        int id_p;
};
// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
        int id_c;
};
//main function
int main()
{
    Child obj1;
    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is " << obj1.id_c << endl;
    cout << "Parent id is " << obj1.id_p << endl;
    return 0;
}
```

Access Modifiers:

1. Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

**Example 2:**

```
class A
{
public:
int x;
protected:
int y;
private:
int z;
};
class B : public A
{
// x is public
// y is protected
// z is not accessible from B
};
class C : protected A
{
// x is protected
// y is protected
// z is not accessible from C
};
class D : private A // 'private' is default for classes
{
// x is private
// y is private
// z is not accessible from D
}
```

**UML diagram of Inheritance**

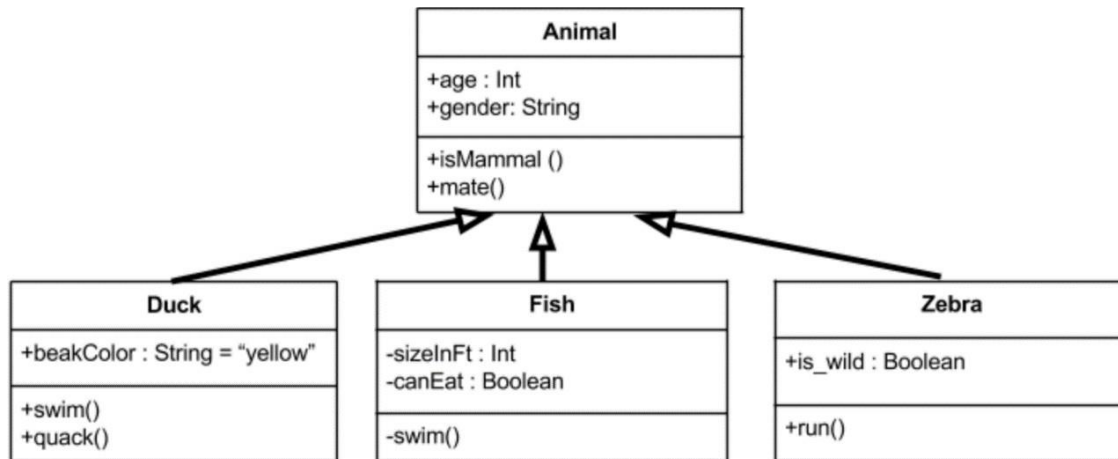


Figure 3 Inheritance among Animals [1]

### Lab Tasks:

- 1- Write a program that defines a shape class with a constructor that gives value to width and height. Then define two sub-classes triangle and rectangle, that calculate the area of the shape area (). In the main, define two variables a triangle and a rectangle and then call the area() function in this two variables.
- 2- Write a program with a mother class and an inherited daughter class. Both of them should have a method void display() that prints a message (different for mother and daughter). In the main define a daughter and call the display() method on it.
- 3- Write a problem with a mother class animal. Inside it define a name and an age variables, and set\_value() function. Then create two bases variables Zebra and Dolphin which write a message telling the age, the name and giving some extra information (e.g. place of origin).

### References:

1. [https://medium.com/@smagid\\_allThings/uml-class-diagrams-tutorial-step-by-step-520fd83b300b](https://medium.com/@smagid_allThings/uml-class-diagrams-tutorial-step-by-step-520fd83b300b)

# Lab No. 12

---

**Target:**

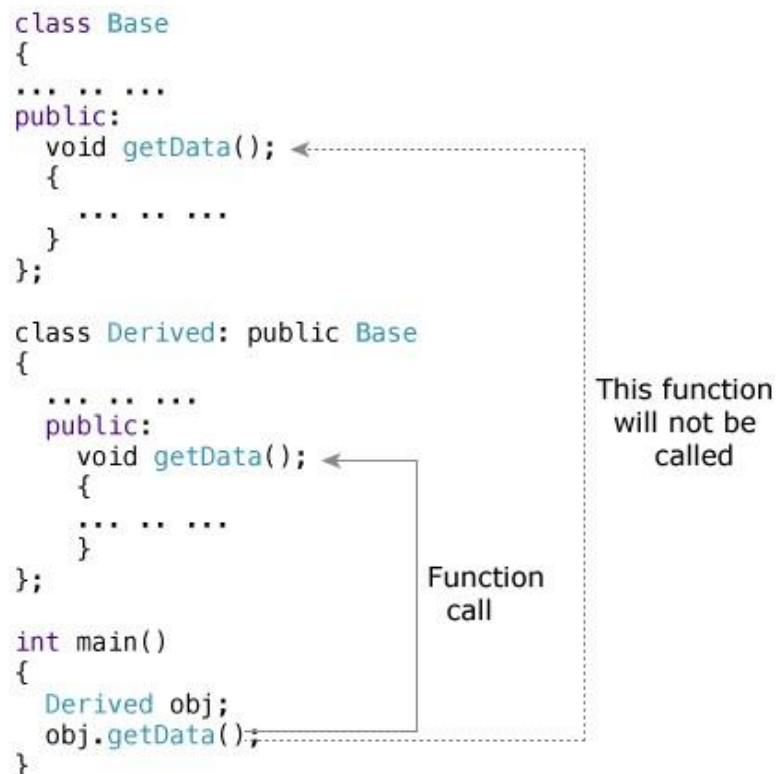
**CLO1**

**Use of constructors and destructors in case of inheritance, function overriding, class hierarchy**

**Function Overriding:** Giving new implementation of base class method into derived class is called function overriding.

Signature of base class method and derived class must be same. Signature involves:

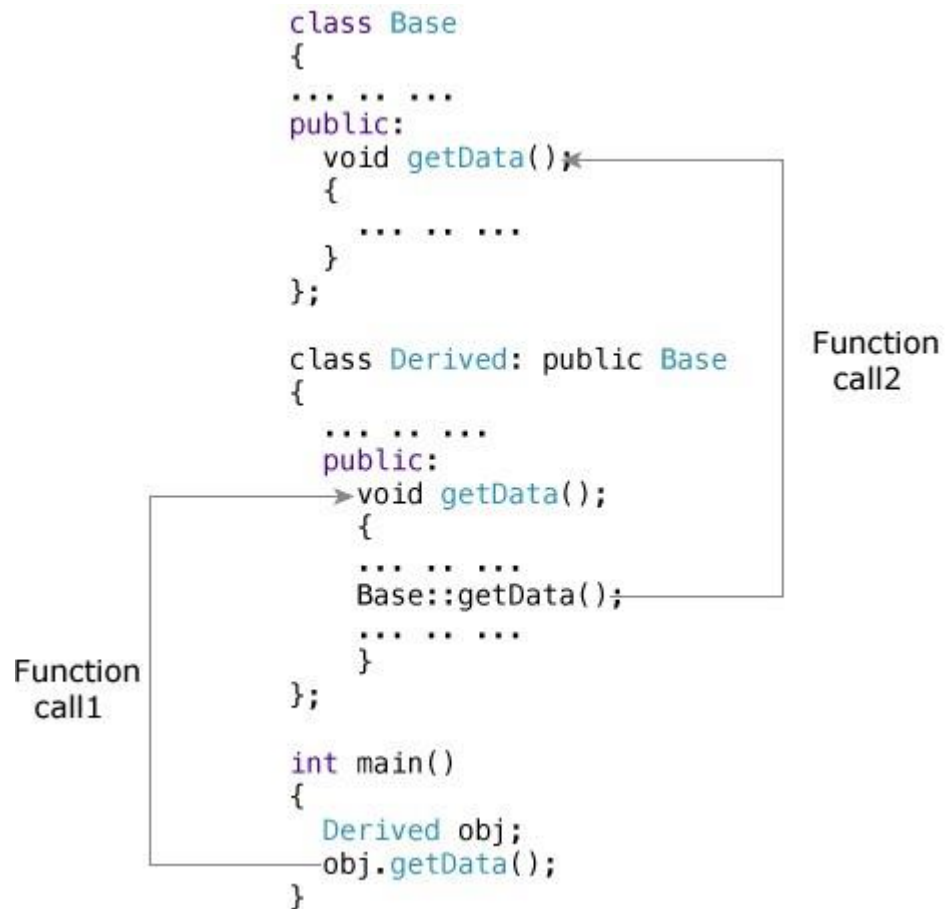
- Number of arguments
- Type of arguments
- Sequence of arguments



To access the overridden function of the base class from the derived class, scope resolution operator `::` is used. For example,

If you want to access `getData()` function of the base class, you can use the following statement in the derived class.

`Base :: getData();`



**Example:**

```

#include<iostream.h>
#include<conio.h>
class BaseClass
{
    public:
    void Display()
    {
        cout<<"\n\tThis is Display() method of BaseClass";
    }
    void Show()
    {
        cout<<"\n\tThis is Show() method of BaseClass";
    }
};
class DerivedClass : public BaseClass
{
    public:

```

```

        void Display()    //overriding method - new working of
        {                //base class's display method
            cout<<"\n\tThis is Display() method of DerivedClass";
        }
    };
    void main()
    {
        DerivedClass      Dr;
        Dr.Display();
        Dr.Show();
    }

```

### Function Overloading:

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

```

#include <iostream>
using namespace std;
void print(int i){
    cout << " Here is int " << i << endl;
}
void print(double f) {
    cout << " Here is float " << f << endl;
}
void print(char* c) {
    cout << " Here is char* " << c << endl;
}
int main() {
    print(10);
    print(10.10);
    print("ten");
    return 0;
}

```

Function Overload	Function Override
The scope is the same	The scope is different
Signatures must differ (ex: parameter)	Signatures must be same
Number of overloading functions possible	Only one overriding function possible
May occur without inheritance	It mainly occurs due to inheritance

### Constructors and Destructors in Inheritance

The base class's constructor is called before the derived class's constructor. The destructors are called in reverse order, with the derived class's destructor being called first.

In inheritance, the base class constructor is called before the derived class constructor. Destructors are called in reverse order. Program below shows a simple set of demonstration classes, each with a default constructor and a destructor. The DerivedClass class is derived from the BaseClass class. Messages are displayed by the constructors and destructors to demonstrate when each is called.



```

// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;
*****

BaseClass declaration *
*****

class BaseClass
{
public:
BaseClass() // Constructor
{ cout << "This is the BaseClass constructor.\n"; }
~BaseClass() // Destructor
{ cout << "This is the BaseClass destructor.\n"; }
};
//*****

// DerivedClass declaration *
//*****

class DerivedClass : public BaseClass
{
public:
DerivedClass() // Constructor
{ cout << "This is the DerivedClass constructor.\n"; }
~DerivedClass() // Destructor
{ cout << "This is the DerivedClass destructor.\n"; }
};
//*****

// main function *
//*****

int main()
{
cout << "We will now define a DerivedClass object.\n";
DerivedClass object;
cout << "The program is now going to end.\n";
return 0;
}

```

### Program Output

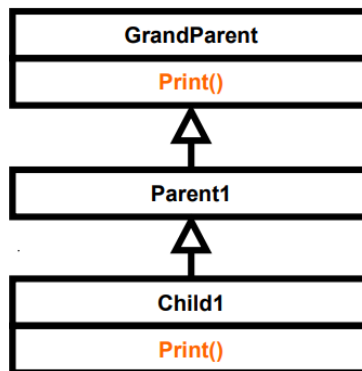
```

We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.

```

### Class Hierarchies

A base class can also be derived from another class. Sometimes it is desirable to establish a hierarchy of classes in which one class inherits from a second class, which in turn inherits from a third class

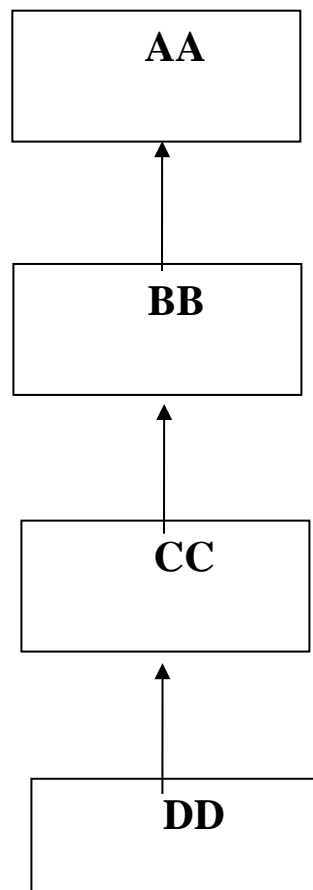


### Lab Tasks:

#### Task 1:

##### Order of Construction and Destruction

Implement the following scenario and show the execution order for construction and destruction of objects in inheritance hierarchy. Class AA is serving as the base class. Class BB is publicly inherited from class AA. Class CC is publicly inherited from class BB and finally the class DD is publicly inherited from class CC.



## Task 2:

**Create a base class Card** with the following attributes

- Card number: private
- Owner name: protected
- Expiry date: public

Derive the following classes from Card, with mentioned additional attributes

- Calling card (public inheritance)
  - Amount: private
  - Company name: private
  - PIN: private
- ID card (public Inheritance)
  - CNIC Number: private
  - Age: private
- Driving license card(public Inheritance)
  - Driving license type (heavy, light, bike) : private
  - Issued in city : private

Hint: Use Inheritance

Your tasks:

1. In the derived classes, write the getters and setters of every member variable (including the derived variables).main() is required to add card of each type, and then to display their information. The object of the base class will not be instantiated.
2. According to the rules of inheritance, clearly specify (by adding comments in the derived classes) which of the members are inherited and clearly mention their access specifiers in the derived classes.
3. Draw the UML diagram of the above-mentioned scenario in any software of your choice.

## Task3:

Write a class Book that has 3 attributes id, name & author. Create new class novel that inherits Book class. It has some additional attributes of publisher, price, published date, total copies. Display a complete set of information regarding Novel id, name, author, publisher, price, published date, total copies.

- You are required to set & display information for at least 3 novels.
- In the end, calculate the total cost for each novel.
- Display information of a novel with a highest individual price.
- Display information of a novel with a maximum total cost.

- Display information of a novel with a maximum total cost.

Hint: Use Inheritance

# Lab No. 13

---

**Target:**

**CLO1**

## **Polymorphism**

**Polymorphism:** The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, an employee. So, a same person can have different behavior in different situations. This is called polymorphism.

### **Example:**

```
#include <iostream>
using namespace std;
class Shape {
protected:
    int width, height;
public:
    Shape(int a=0, int b=0){
        width = a;
        height = b;
    }
    int area(){
        cout << "Parent class area:" << endl;
        return 0;
    }
};

class Rectangle: public Shape {
public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
        cout << "Rectangle class area :" << endl;
        return (width * height);
    }
};

class Triangle: public Shape {
public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
        cout << "Triangle class area :" << endl;
```

```

        return (width * height / 2);
    }
};

```

```

// Main function for the program
int main() {
    Shape      *shape;
    Rectangle  rec(10,7);
    Triangle   tri(10,5);

    // store the address of Rectangle
    shape = &rec;

    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;

    // call triangle area.
    shape->area();

    return 0;
}

```

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

- Parent class area
- Parent class area

The reason for the incorrect output is that the call of the function `area()` is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the `area()` function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of `area()` in the `Shape` class with the keyword **virtual** so that it looks like this –

```

class Shape {
protected:
    int width, height;

public:

```

```

Shape( int a = 0, int b = 0) {
    width = a;
    height = b;
}
virtual int area() {
    cout<<"Parent class area:"<<endl;
    return 0;
}
};

```

After this slight modification, when the previous example code is compiled and executed, it produces the following result –

- Rectangle class area
- Triangle class area

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

### Virtual Function:

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

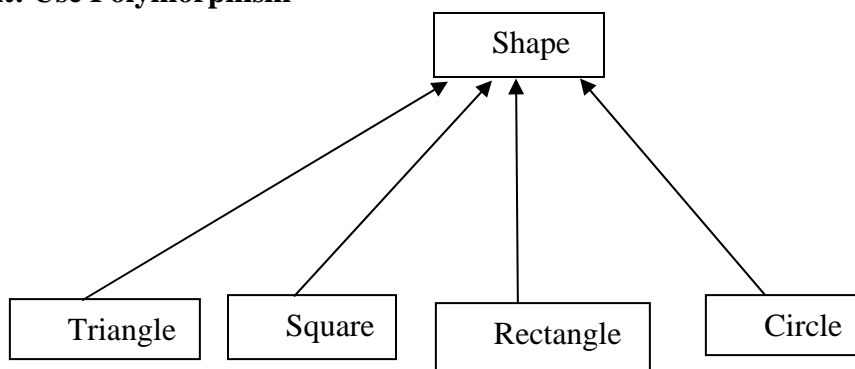
What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

### Lab Tasks:

#### Task 1:

Implement the following class hierarchy. Write a function calculate Area to calculate area of each object of any shape.

Hint: Use Polymorphism



### **Task 2:**

Make a vehicle class with its weight and wheels as data members and message as member function. The message would display that it is a vehicle. Make 3 derived classes i.e. boat, car, and truck. All the derived classes will display the type of vehicle they are. (you can use data members in derived classes also)

### **Task 3:**

```
class Employee {  
private:  
String name;  
double taxRate;  
public:  
Employee( String&, double );  
String getName();  
virtual double calcSalary();}
```

Write a program that calculates the salaries of employees in a software company. The employees include the Manager of the firm, HR, Engineers, Front End Developers and Programmers. You need to create an employee class, that is inherited by all other classes and display their salaries. The criteria of salaries are as under:

- Salary of Manager is fixed i.e. 1.5 lac
- Salary of HR is 1 lac
- Salary of engineer is 75k
- Salary of front-end developer is 60k
- Salary of programmer is 60k

The net salary of all employees except manager is calculated on the basis of Over Time (OT) hours. For each OT hour a bonus of 1.5k is added in the salary of each employee. Now you have to display the salaries of all employees using above formula with OT of your own choice.



# Lab No. 14

---

**Target:**

**CLO1**

**Calling virtual functions from constructor/destructors, override specifier and final specifier**

**Calling virtual functions from constructor/destructors:**

All the C++ implementations need to call the version of the function defined at the level of the hierarchy in the current constructor and not further.

You can call a virtual function in a constructor. The Objects are constructed from the base up, “base before derived”.

**Example:**

```
// CPP program to illustrate
// calling virtual methods in
// constructor/destructor
#include <iostream>
using namespace std;
class dog
{
public:
    dog()
    {
        cout << "Constructor called" << endl;
        bark();
    }
    ~dog()
    {
        bark();
    }
    virtual void bark()
    {
        cout << "Virtual method called" << endl;
    }
    void seeCat()
    {
        bark();
    }
};
class Yellowdog : public dog
{
public:
    Yellowdog()
    {
```

```

        cout<< "Derived class Constructor called" <<endl;
    }
    void bark()
    {
        cout<< "Derived class Virtual method called" <<endl;
    }
};
int main()
{
    Yellowdog d;
    d.seeCat();
}

```

Output:

```

Constructor called
Virtual method called
Derived class Constructor called
Derived class Virtual method called
Virtual method called

```

## Override specifier

Specifies that a virtual function overrides another virtual function.

### Syntax

The identifier `override`, if used, appears immediately after the declarator in the syntax of a member function declaration or a member function definition inside a class definition.

1) In a member function declaration, `override` may appear in *virt-specifier-seq* immediately after the declarator, and before the *pure-specifier*, if used.

2) In a member function definition inside a class definition, `override` may appear in *virt-specifier-seq* immediately after the declarator and just before *function-body*.

In both cases, *virt-specifier-seq*, if used, is either `override` or `final`, or `final override` or `override final`.

In a member function declaration or definition, `override` specifier ensures that the function is virtual and is overriding a virtual function from a base class. The program is ill-formed (a compile-time error is generated) if this is not true.

`override` is an identifier with a special meaning when used after member function declarators: it's not a reserved keyword otherwise.

### Example

```

struct A
{
    virtual void foo();
    void bar();
};

```

```

struct B : A
{

```

```

void foo() const override; // Error: B::foo does not override A::foo
                        // (signature mismatch)
void foo() override; // OK: B::foo overrides A::foo
void bar() override; // Error: A::bar is not virtual
};

```

### Final specifier:

We can use final for a function to make sure that it cannot be overridden. We can also use final in Java to make sure that a class cannot be inherited. Sometimes you don't want to allow derived class to override the base class' virtual function.

#### Example

```

#include <iostream>
using namespace std;
class Base
{
public:
virtual void myfun() final
{
    cout << "myfun() in Base";
}
};
class Derived : public Base
{
void myfun()
{
    cout << "myfun() in Derived\n";
}
};
int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}

```

#### Lab Tasks:

1- a)

Write a C++ program to explain virtual function (polymorphism) by creating a base class `c_polygon` which has virtual function `area()`. Two classes `c_rectangle` and `c_triangle` derived from `c_polygon` and they have `area()` to calculate and return the area of rectangle and triangle respectively.

b) now using the same implementation, use “final” word with virtual function in base class and see the results.

c) now using the same implementation, use “override” word with overridden function prototype in child class and see the results.

d) using (c) part, change the prototype of overridden function by just passing an argument in it (which was not in passed in function of base class) only, keep the “override” with it, again compile and see the results.

2-

- a) Create a simple “shape” hierarchy: a base class called **Shape** and derived classes called **Circle**, **Square**, and **Triangle**. In the base class, make a virtual function called **draw()**, and override this in the derived classes. Make an array of pointers to **Shape** objects that you create on the heap (and thus perform upcasting of the pointers), and call **draw()** through the base-class pointers, to verify the behavior of the virtual function.

3-

Design a Ship class that has the following members:

- A member variable for the name of the ship (a string)
- A member variable for the year that the ship was built (a string)
- A constructor and appropriate setter and getter
- A virtual print function that displays the ship’s name and the year it was built.

Design a CruiseShip class that is derived from the Ship class. The CruiseShip class should have the following members:

- A member variable for the maximum number of passengers (an int)
- A constructor and appropriate setter and getter.
- A print function that overrides the print function in the base class. The CruiseShip class’s print function should display only the ship’s name and the maximum number of passengers.

Design a CargoShip class that is derived from the Ship class. The CargoShip class should have the following members:

- A member variable for the cargo capacity in tonnage (an int).
- A constructor and appropriate setter and getter.
- A print function that overrides the print function in the base class. The CargoShip class’s print function should display only the ship’s name and the ship’s cargo capacity.

In main call the relevant classes’s print functions using the base class pointer and see the results.

# Lab No. 15

---

**Target:**

**CLO2**

## **Early and Late Binding, Pointer to functions, Pure virtual functions, Virtual table Late Binding**

In the case of late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as Dynamic Binding or Runtime Binding.

In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

By default, early binding takes place. So, if by any means we tell the compiler to perform late binding, then the problem in the previous example can be solved.

This can be achieved by declaring a virtual function.

Virtual Function is a member function of the base class which is overridden in the derived class. The compiler performs late binding on this function.

To make a function virtual, we write the keyword virtual before the function definition.

### **Example**

```
#include <iostream>

using namespace std;

class Animals
{
public:
    virtual void sound()
    {
        cout << "This is parent class" << endl;
    }
};

class Dogs : public Animals
{
public:
    void sound()
    {
        cout << "Dogs bark" << endl;
    }
}
```

```

    }
};

int main()
{
    Animals *a;
    Dogs d;
    a= &d;
    a->sound();
    return 0;
}

```

### **Output:**

Dogs bark

### **Pure Virtual Functions:**

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following

```

class Shape {
protected:
    int width, height;

public:
    Shape(int a=0, int b=0){
        width = a;
        height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};

```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

### **The Virtual Table:**

The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

**Example:**

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we'll put it in the next example just to show where it's added:

```
class Base
{
public:
    FunctionPointer *_vptr;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base
{
public:
    virtual void function1() {};
};

class D2: public Base
{
public:
    virtual void function2() {};
```

```
};
```

**Lab Tasks:**

4- What will be the output of the following C++ code?

```
#include <iostream>
#include<type_traits>
using namespace std;
class Base
{
    public:
        virtual void function1() {};
        virtual void function2() {};
};
int main()
{
    Base      b;
    cout<<sizeof(b);
    return 0;
}
```

- a) 1
- b) 4
- c) 8
- d) 16

2-

Given below classes which of the following are the possible row entries in vtable of Base class?

```
class Base
{
    public:
        virtual void function1() {};
        virtual void function2() {};
};
class D1: public Base
{
    public:
        virtual void function1() {};
};
class D2: public Base
{
    public:
        virtual void function2() {};
};
```



- a) Base::function1() and Base::function2()
- b) Base::function1() and D1::function2()
- c) D1::function1() and Base::function2()
- d) D1::function1() and D1::function2() or D2::function1() and D2::function2()

3-

- a) Write a program for simple calculator using Dynamic/ Late binding.
- b) Write the same program using Static/ Early Binding