CS112L - Object Oriented Programming Language

Lab Manual

Faculty of Computer Science and Engineering

Contents

I. LAB OUTLINE

CS112L Object Oriented Programming Lab (1 CH)
Pre-Requisite: CS101 Instructor: **Engr. Amna Arooj** Email: *amna.arooj@giki.edu.pk* Office #03, G35 FCSE. Ext. 2746 Office Hours: 10:00am ~ 01:00 pm

Lab Introduction

As a second lab. on programming, the emphasis would be that students should be able to write a program of reasonable size and complexity. Devising a solution to a problem will be encouraged and converting a design into a computer program would be stressed including the software reuse. The primary aspect of the lab is to introduce students with the object-oriented programming skills. This Lab will provide in-depth coverage of object-oriented programming principles and techniques using C++. Topics include classes, overloading, data abstraction, information hiding, encapsulation, inheritance, polymorphism, file processing, templates, exceptions, container classes, and low-level language features.

Lab Contents

Broadly, this Lab will cover following: Introduction to Classes and Objects, Control Structures, Methods, Arrays, Pointers, Classes Inheritance, Polymorphism, Templates, Exceptions, STL, and Operator Overloading, Dynamic Memory Allocation and Dynamic Arrays.

Mapping of CLOs and PLOs

Sr. No	Course Learning Outcomes[+]	PLOs*	Blooms Taxonomy
CLO_1	Utilize the basic techniques of an object-oriented programming language.	-	P2 (Set)
CLO_2	Implement programming structures to design solutions for the given problems.	PLO 1	P3 (Guided Response)
CLO_3	Apply the major object-oriented concepts to implement programs in C++ using encapsulation, inheritance, and polymorphism .	PLO 3	P4 (Mechanism)
	[+]Please add the prefix "Upon successful completion of this course, the student will be able to" [*]PLOs are for BS (CE) only		

CLO Assessment Mechanism (Tentative)

Assessment tools	CLO_1	CLO_2	CLO_3
Lab Performance	80%	45%	20%
Open Ended Lab	-	10%	-
Project	-	15%	20%
Midterm Exam	20%	-	20%
Final Exam	-	30%	40%

Overall Grading Policy (Tentative)

Assessment Items	Percentage
Lab Performance	25%
Midterm Exam	20%
Open Ended Lab	5%
Project	15%
Final Exam	35%

Text and Reference Books

Text books:

- Harvey M. Dietel and Paul J. Deitel, "How to Program C++", 9th Edition, Deitel & Associates, Inc. (2014)
- Lab Manual for CS112L

Lab Breakdown

Week	Contents/Topics
Week 1	User defined data types - Structures, Unions, Enumerations
Week 2	Pre-processor Directives, Bit Manipulation, Function Pointers
Week 3	Dynamic Memory Allocation and Dynamic Arrays
Week 4	C++ Classes-I Introduction
Week 5	C++ Classes-II Constructor, Destructor, Copy Constructor, this Pointer
Week 6	Friend Functions and Classes, Static Members, Constant Objects and Functions, Composition
Week 7	Type Casting – Static Vs Dynamic Casting
Week 8	Operator Overloading
Week 9	Inheritance
Week 10	Virtual Functions, Abstract Base Classes and Polymorphism
Week 11	Introduction to Standard Template Library (STL)
Week 12	Open-Ended Lab

II. BENCHMARK DETAILS

University Name: Massachusetts Institute of Technology

Department: Computer Science

Ranking: Ranked 4th best University in THE World University Rankings 2019, 3rd in National Universities and 3rd in Best Value Schools in USA. In QC world ranking, its rank No. 1 in 11 Subjects which Includes Computer Science and Information Systems.

III. SYSTEM REQUIREMENTS

HARDWARE REQUIREMENT

- Core i3 or Above
- 2 GB RAM
- 20 GB HDD

SOFTWARE REQUIREMENT

- Windows 8x or above
- Dev C++ IDE / Visual Studio

IV. EVALUATION RUBRICS

Weekly Evaluation	
Rubrics	**Marks (10)**
Code writing	1
Error/exception Handling	2
Code Comments / Necessary Documentation	3
Correctness	4

Detailed Rubric of weekly Evaluation:

Code writing (No credit, if not relevant implementation):
* Indentation: 0.5
* Meaningful/relevant variable/function names: 0.5

Correctness:
* Accurate methodology 2
* All tasks are done: 1
* Passes all test cases: 1

Code Comments / Necessary Documentation:
* Knowledge about the topic 2
* clarify meaning where needed 1

Error/exception Handling:
* Runs without exception: 1
* Interactive menu: 0.5
* Displays proper messages/operations clearly: 0.5

Criteria	Excellent (Well executed)	Good (Room for improvement exists)	Adequate (Meets minimum requirements)	Unsatisfactory (Does not meet minimum requirements)
	10	8	6	< 5
Code writing	Coding style guidelines are followed correctly, code is exceptionally easy to read and maintain. All names are consistent with prescribed style practices.	Coding style guidelines are almost always followed correctly. Code is easy to read. Names are consistent in style. Isolated cases exist of deviation from prescribed practices.	Coding style guidelines are not followed and/or code is less readable than it should be. Names are nearly always consistent, but occasionally verbose, overly terse, ambiguous or misleading.	Below minimum requirements. (See left column)
Correctness	Provided solution by the student correctly solves problem in all cases and exceeds problem specifications.	Provided solution by the student correctly solves problem in all or nearly all cases but may have minor problems in some instances.	Provided solution by the student solves problem in some cases but has one or more problems.	Below minimum requirements. (See left column)
Code comments or necessary documentation	Comments/documentation clarify meaning where needed.	Comments/documentation usually clarify meaning. Unhelpful comments may exist.	Comments/documentation exist but are frequently unhelpful or occasionally misleading.	Below minimum requirements. (See left column)
Error/exception Handling	Provided solution by the student handles erroneous or unexpected conditions gracefully and action is taken without surprising the user.	All obvious error conditions are checked for and appropriate action is taken.	Some obvious error conditions are checked for and some sort of action is taken.	Below minimum requirements. (See left column)

V. INSTRUCTIONS FOR STUDENTS

These are the instructions for the students attending the lab:

- Before entering the lab, the student should carry the following things (MANDATORY)
    1. Identity card issued by the college.
    2. Class note book
    3. Lab observation book/ Lab Manual
- Student must sign in in the Rubric/ Attendance sheet provided when attending the lab session.
- Come to the laboratory in time. Students, who are late more than 10 min., will not be allowed to attend the lab.
- Students need to maintain 100% attendance in lab if not a strict action will be taken.
- All students must follow a Dress Code while in the laboratory
- Foods, drinks are NOT allowed.
- Refer to the lab staff if you need any help in using the lab.
- Respect the laboratory and its other users.
- Workspace must be kept clean and tidy after experiment is completed.
- Read the Manual carefully before coming to the laboratory and be sure about what you are supposed to do.
- Do the experiments as per the instructions given in the manual.

Lab 01

User Defined Data Types – Structures, Unions, Enumerations

1.1 Structure

A structure is a collection of variable which can be same or different types. You can refer to a structure as a single variable, and to its parts as **members** of that variable by using the dot (.) operator. The power of structures lies in the fact that once defined, the structure name becomes a **user-defined data type** and may be used the same way as other built-in data types, such as int, double, char.

Have a look in the following example and the its each bit is explained below according to structures concepts:

Example 1

```
struct Student
{
   int rollno, age;
   char name[80];
   float marks;
};

int main()
{

   // declare two variables of the new type
   Student  s1, s3;

   //accessing of data members
   cin >> s1.rollno >> s1.age >> s1.name >> s1.marks;
   cout << s1.rollno << s1.age << s1.name << s1.marks;

   //initialization of structure variable
   Student s2 = {100, 17, "Aniket", 92};
   cout << s2.rollno << s2.age << s2.name << s2.marks;

   //structure variable in assignment statement
   s3 = s2;
   cout << s3.rollno << s3.age << s3.name << s3.marks;

   return 0;
}
```

1.1.1 Defining a structure

When dealing with the students in a school, many variables of different types are needed. It may be necessary to keep track of name, age, Rollno, and marks point for example.

```
struct Student
{
   int rollno, age;
   char name[80];
   float marks;
};
```

Student is called the **structure tag,** and is your brand new data type, like int, double or char. **rollno, name, age,** and **marks** are **structure members.**

1.1.2 Declaring Variables of Type struct

The most efficient method of dealing with structure variables is to define the structure **globally**. This tells "the whole world", namely main and any functions in the program, that a new data type exists. To declare a structure globally, place it **BEFORE** void main(). The structure variables can then be defined locally in main, for example…

```
struct Student
{
   int rollno, age;
   char name[80];
   float marks;
};


int main()
{
   // declare two variables of the new type
   Student  s1, s3;
    ……
    ……
    return 0;
}
```

Alternate method of declaring variables of type struct:

```
struct Student
{
   int rollno, age;
   char name[80];
   float marks;
} s1, s3;
```

1.1.3 Accessing of data members

The accessing of data members is done by using the following format:
structure variable.member name
for example

cin >> s1.rollno >> s1.age >> s1.name >> s1.marks;

1.1.4 Initialization of structure variable

Initialization is done at the time of declaration of a variable. For example

Student s2 = {100, 17, "Aniket", 92};

Structure variable in assignment statement

s3 = s2;
The statement assigns the value of each member of s2 to the corresponding member of s3. Note that one structure variable can be assigned to another only when they are of the same structure type, otherwise complier will give an error.

1.1.5 Nested structure (Structure within structure)

It is possible to use a structure to define another structure. This is called nesting of structure. Consider the following program

```
struct Day
{
   int month, date, year;
};

struct Student
{
   int rollno, age;
   char name[80];
   Day date_of_birth;
   float marks;
};
```

To access members of date_of_birth we can write the statements as below:

```
Student s;  // Structure variable of Student

s.date_of_birth.month = 11;
s.date_of_birth.date = 5;
s.date_of_birth.year = 1999;
```

Example 2

```
#include<iostream.h>
 struct Address
 {
      char HouseNo[25];
      char City[25];
      char PinCode[25];
 };
 struct Employee
 {
    int Id;
    char Name[25];
    float Salary;
    struct Address Add;

 }
 void main()

{
      int i;
      Employee E;

      cout << "\n\tEnter Employee Id : ";
      cin >> E.Id;

      cout << "\n\tEnter Employee Name : ";
      cin >> E.Name;

      cout << "\n\tEnter Employee Salary : ";
      cin >> E.Salary;

      cout << "\n\tEnter Employee House No : ";
      cin >> E.Add.HouseNo;

      cout << "\n\tEnter Employee City : ";
      cin >> E.Add.City;

      cout << "\n\tEnter Employee House No : ";
      cin >> E.Add.PinCode;

      cout << "\nDetails of Employees";
      cout << "\n\tEmployee Id : " << E.Id;
```

```
        cout << "\n\tEmployee Name : " << E.Name;
        cout << "\n\tEmployee Salary : " << E.Salary;
        cout << "\n\tEmployee House No : " << E.Add.HouseNo;
        cout << "\n\tEmployee City : " << E.Add.City;
        cout << "\n\tEmployee House No : " << E.Add.PinCode;

    }
```

Output:

```
            Enter Employee Id : 101
            Enter Employee Name : Suresh
            Enter Employee Salary : 45000
            Enter Employee House No : 4598/D
            Enter Employee City : Delhi
            Enter Employee Pin Code : 110056

            Details of Employees
            Employee Id : 101
            Employee Name : Suresh
            Employee Salary : 45000
            Employee House No : 4598/D
            Employee City : Delhi
            Employee Pin Code : 110056
```

1.1.6 Pointers to Structure

As we have learnt a memory location can be accessed by a pointer variable. In the similar way a structure is also accessed by its pointer.
For example, a pointer to **struct** employee may be defined by the statement **struct** employee ***sptr**; In other words the variable **sptr** can hold the address value for a structure of type **struct employee.** We can create several pointers using a single declaration, as follows:

Struct employee *sptr1, *sptr2, *sptr3;

We have a structure:

```
struct employee{
        char name[30];
        int age;
        float salary;
    };
```

We define its pointer and variable as follow:

struct employee *sptr1, emp1;

A pointer to a structure must be initialized before it can be used anywhere in program. The address of a structure variable can be obtained by applying the **address operator &** to the variable. For example, the statement **sptr1 = &emp1;**

The members of a structure can be accessed using an arrow operator through a structure pointer.

sptr1->name
sptr1->age
sptr1->salary

Example 3:

```
#include <iostream>
using namespace std;
struct Complex
{
int real;
float img;
};

int main()
{
  Complex var1;
  /* creating a pointer of Complex type & assigning address of var1 to this pointer */
  Complex* ptr = &var1;

  var1.real = 5;
  var1.img = 0.33;

cout<<"Real part: "<<  ptr->real  <<endl;
cout<<"Imaginary part: "<<   ptr-> img;
return 0;
}
```

1.2 Unions

Both structure and union are collection of different datatype. They are used to group number of variables of different type in a single unit.

1.2.1 Union declaration

Syntax for declaring union

```
union union-name
{
    datatype var1;
    datatype var2;
    - - - - - - - - - -
    - - - - - - - - - -
    datatype varN;
};
```

1.2.2 Accessing the union members

We have to create an object of union to access its members. Object is a variable of type union. Union members are accessed using the dot operator(.) between union's object and union's member name.
Syntax for creating object of union

```
union union-name obj;
```

Example 4

```
//Example for creating object & accessing union members

    #include<iostream.h>

    union Employee
    {
        int Id;
        char Name[25];
        int Age;
        long Salary;
    };

    void main()
    {

        Employee E;
```

```
        cout << "\nEnter Employee Id : ";
        cin >> E.Id;

        cout << "\nEnter Employee Name : ";
        cin >> E.Name;

        cout << "\nEnter Employee Age : ";
        cin >> E.Age;

        cout << "\nEnter Employee Salary : ";
        cin >> E.Salary;

        cout << "\n\nEmployee Id : " << E.Id;
        cout << "\nEmployee Name : " << E.Name;
        cout << "\nEmployee Age : " << E.Age;
        cout << "\nEmployee Salary : " << E.Salary;
}
```

Output :

```
Enter Employee Id : 1
Enter Employee Name : Kumar
Enter Employee Age : 29
Enter Employee Salary : 45000

Employee Id : -20536
Employee Name : ?$?$  ?
Employee Age : -20536
Employee Salary : 45000
```

In the above example, we can see that values of Id, Name and Age members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value of salary member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

Example 5

```
#include<iostream.h>

union Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
```

```
};

void main()
{

    Employee E;

        cout << "\nEnter Employee Id : ";
        cin >> E.Id;
        cout << "Employee Id : " << E.Id;

        cout << "\n\nEnter Employee Name : ";
        cin >> E.Name;
        cout << "Employee Name : " << E.Name;

        cout << "\n\nEnter Employee Age : ";
        cin >> E.Age;
        cout << "Employee Age : " << E.Age;

        cout << "\n\nEnter Employee Salary : ";
        cin >> E.Salary;
        cout << "Employee Salary : " << E.Salary;

}
```

Output :

```
Enter Employee Id : 1
Employee Id : 1

Enter Employee Name : Kumar
Employee Name : Kumar

Enter Employee Age : 29
Employee Age : 29

Enter Employee Salary : 45000
Employee Salary : 45000
```

Here, all the members are getting printed very well because one member is being used at a time.

1.2.3 Differences Between Structure & Union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

1.3 Enumerations

Enum is a user defined data type where we specify a set of values for a variable and the variable can only take one out of a small set of possible values. We use enum keyword to define a Enumeration. E.g.

enum direction {East, West, North, South}dir;

Here Enumeration name is direction which can only take one of the four specified values, the dir at the end of the declaration is an enum variable.

Example 6

```
#include <iostream>
using namespace std;
enum direction {East, West, North, South};
int main(){
  direction dir;
  dir = South;
  cout<<dir;
  return 0;
}
```

Output:

```
3
```

Here we have assigned the value South to the enum variable dir and when I displayed the value of dir it shows 3. This is because by default the values are in increasing order starting from 0, which means East is 0, West is 1, North is 2 and South is 3.

1.3.1 How to change default values of Enum

Example 7

```
#include <iostream>
using namespace std;
enum direction {East=11, West=22, North=33, South=44};
int main(){
  direction dir;
  dir = South;
  cout<<dir;
  return 0;
}
```

Output:

```
44
```

1.3.2 Why use enum in C++

Enums are used only when we expect the variable to have one of the possible set of values, for example, we have a dir variable that holds the direction. Since we have four directions, this variable can take any one of the four values, if we try to assign a another random value to this variable, it will throw a compilation error. This increases compile-time checking and avoid errors that occurs by passing in invalid constants.

Another important place where they are used frequently are switch case statements, where all the values that case blocks expect can be defined in an enum. This way we can ensure that the enum variable that we pass in switch parenthesis is not taking any random value that it shouldn't accept.

Lab 02

Pre-processor Directives, Bit Manipulation, Function Pointers

2.1 Pre-processor Directives

Preprocessors are programs that processes our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps:

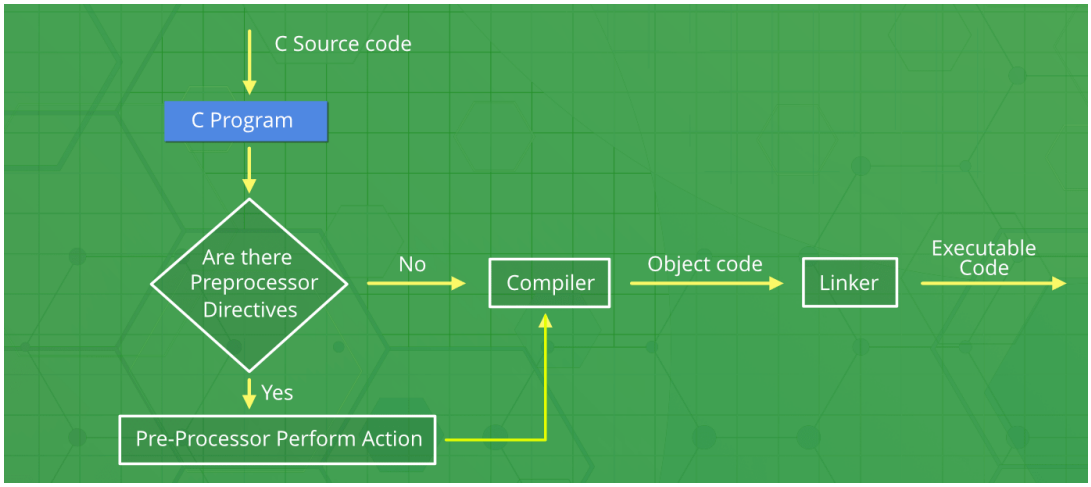


Fig 1: Execution of a code

All of these preprocessor directive begins with a '#' (hash) symbol. This ('#') symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these pre-processor directives anywhere in our program. Examples of some preprocessor directives are: #include, #define, #ifndef etc.

There are 4 main types of preprocessor directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

2.1.1 Macros

Macros are piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The

'#define' directive is used to define a macro. Let us now understand macro definition with the help of a program:

Example 1

```
#include <iostream>
  // macro definition
#define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        std::cout << i << "\n";
    }
    return 0;
}
```

Output

```
0
1
2
3
4
```

In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word 'LIMIT' in macro definition is called macro template and '5' is macro expansion.
Note: There is no semi-colon(';') at the end of macro definition. Macro definitions do not need a semi-colon to end.

Macros with arguments: We can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program:

Example 2

```
#include <iostream>

// macro with parameter
#define AREA(l, b) (l * b)
int main()
{
    int l1 = 10, l2 = 5, area;
```

```
    area = AREA(l1, l2);

    std::cout << "Area of rectangle is: " << area;

    return 0;
}
```

Output:

```
Area of rectangle is: 50
```

We can see from the above program that whenever the compiler finds AREA(l, b) in the program it replaces it with the statement (l*b) . Not only this, the values passed to the macro template AREA(l, b) will also be replaced in the statement (l*b). Therefore AREA(10, 5) will be equal to 10*5.

2.1.2 File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files which can be included by the user in the program:
Header File or Standard files: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example, standard I/O functions are in 'iostream' file whereas functions which perform string operations are in 'string' file.
Syntax:

#include< file_name >

where file_name is the name of file to be included. The '<' and '>' brackets tells the compiler to look for the file in standard directory.

User Defined files: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:
#include"filename"

2.1.3 Conditional Compilation

Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of two preprocessing commands 'ifdef' and 'endif'.

Syntax:

```
#ifdef macro_name
statement1;
statement2;
statement3;
.
.
.
statementN;
#endif
```

If the macro with name as ‘macroname‘ is defined then the block of statements will execute normally but if it is not defined, the compiler will simply skip this block of statements.

Example 3

```
#include <iostream>
using namespace std;
#define TABLE_SIZE 100

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
  int i, j;

  i = 100;
  j = 30;
  cout <<"The minimum is " << MIN(i, j) << endl;

#ifndef TABLE_SIZE
#define TABLE_SIZE 100
 int table[TABLE_SIZE];
#endif

#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif

#if TABLE_SIZE>200
```

```
        #undef TABLE_SIZE
        #define TABLE_SIZE 200
#elif TABLE_SIZE<50
        #undef TABLE_SIZE
        #define TABLE_SIZE 50
#else
        #undef TABLE_SIZE
        #define TABLE_SIZE 100
#endif
table[TABLE_SIZE];



#if 0
  /* This is commented part */
  cout << MKSTR(HELLO C++) << endl;
#endif



  return 0;
}
```

Output

```
The minimum is 30
```

2.1.4 Other directives

Apart from the above directives there are two more directives which are not commonly used. These are:

- **#undef Directive:** The #undef directive is used to un-define an existing macro. This directive works as:

  #undef LIMIT

  This statement will un-define the existing macro LIMIT. After this statement every "#ifdef LIMIT" statement will evaluate to false.

- **#pragma Directive:** This directive is used to turn on or off some features. This type of directives are compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:
  - **#pragma startup** and **#pragma exit**: These directives helps us to specify the functions that are needed to run before program startup (before the control

passes to main()) and just before program exit (just before the control returns from main()).

2.2 Bitwise Operations

~	complement	Bit **n** of **~x** is the opposite of bit **n** of **x**
&	Bitwise And	Bit **n** of **x&y** is 1 if bit **n** of **x** and bit **n** of **y** is 1.
|	Bitwise Or	Bit **n** of **x|y** is 1 if bit **n** of **x** or bit **n** of **y** is 1.
^	Bitwise Exclusive Or	Bit **n** of **x^y** is 1 if bit **n** of **x** or bit **n** of **y** is 1 but not if both are 1.
>>	Right Shift (divide by **2)**	Bit **n** of **x>>s** is bit **n-s** of **x**.
<<	Left Shift (multiply by 2)	Bit **n** of **x<<s** is bit **n+s** of **x**.

Bitwise Complement: The bitwise complement operator, the **tilde**, **~**, flips every bit. The tilde is sometimes called a **twiddle**, and the bitwise complement twiddles every bit: This turns out to be a great way of finding the **largest** possible value for an **unsigned** number.

```
unsigned int max = ~0;
```

Bitwise AND: The bitwise **AND** operator is a single ampersand: **&**:

```
01001000 &
10111000 =
----------
00001000
```

Bitwise OR: The bitwise **OR** operator is a **|**:

```
01001000 |
10111000 =
----------
11111000
```

Bitwise Exclusive OR (XOR): The **exclusive-or** operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a carrot, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated **XOR**.

```
01110010 ^
10101010
--------------
11011000
```

Suppose, we have some bit, either 1 or 0, that we'll call Z. When we take Z **XOR** 0, then we always get Z back: if Z is 1, we get 1, and if Z is 0, we get 0. On the other hand, when we take Z **XOR** 1, we flip Z. If Z is 0, we get 1; if Z is 1, we get 0:

- **myBits ^ 0** : No change
- **myBits ^ 1** : Flip

Example 4

```
//Binary Conversion

#include <iostream>
#include <iomanip>
#include <bitset>
using namespace std;
void binary( int u)
{
cout << setw(5) << u << ": ";
cout << bitset<16>((int)u);
cout << "\n";
}
int main()
{
binary(5);
binary(55);
binary(255);
binary(4555);
binary(14555);
system("PAUSE");
return 0;
}
```

Output:

```
    5: 0000000000000101
   55: 0000000000110111
  255: 0000000011111111
```

```
 4555: 0001000111001011
14555: 0011100011011011
```

2.3 Function Pointers

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

int func(int a, float b);

And simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

/* function returning pointer to int */
int *func(int a, float b); //Wrong

/* pointer to function returning int */
int (*func)(int a, float b); //Right

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function as:

(*func)(1,2);

Example 5

```
#include <stdio.h>
// A normal function with an int parameter and void return type
void fun(int a)
{
   printf("Value of a is %d\n", a);
}
int main()
{
   // fun_ptr is a pointer to function fun()
   void (*fun_ptr)(int) = &fun;

   /* The above line is equivalent of following two
      void (*fun_ptr)(int);
      fun_ptr = &fun;
   */
   // Invoking fun() using fun_ptr
   (*fun_ptr)(10);
   return 0;
```

```
}
```

Output:

```
Value of a is 10
```

Lab 03

Dynamic Memory Allocation and Dynamic Arrays

3.1 Memory Layout of C++ Programs

3.1.1 Introduction

The memory refers to the computer hardware integrated circuits that store information for immediate use in a computer. The computer memory is built to store bit patterns. Not only data but also instructions are bit patterns and these can be stored in memory. In systems software, they are stored in separate segment of memory. And the segments are also divided by data and program type.
When the program runs, the processing is performed in two spaces called **Kernel Space** and **User Space** on the system. The two processing spaces implicitly interfere with each other and the processing of the program proceeds.

- **Kernel Space**: The kernel space can be accessed by user processes only through the use of system calls that are requests in a Unix-like operating system such as input/output (I/O) or process creation.
- **User Space**: The user space is a computational resource allocated to a user, and it is a resource that the executing program can directly access. This space can be categorized into some segments.

3.1.2 Memory Layout Schema

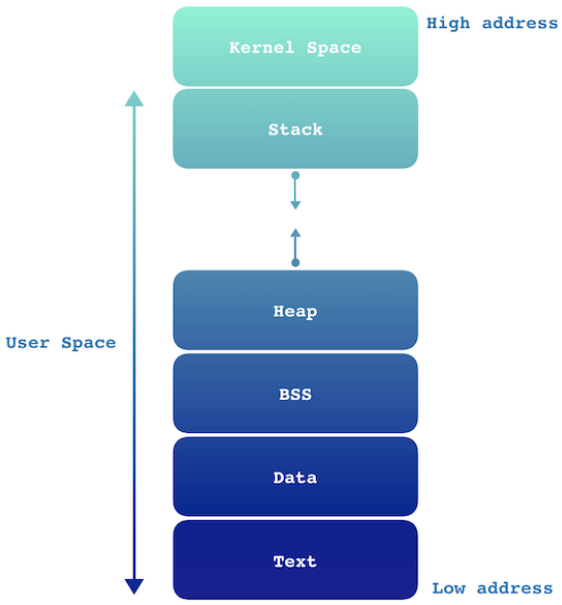


Fig 1. Memory Layout

The above picture shows virtual memory spaces of kernel space and user space. The user space part of the virtual space is categorized into **Stack** and **Heap**, **BSS**, **Data**, **Text**.

Stack

The stack space is located just under the OS kernel space, generally opposite the heap area and grows downwards to lower addresses. (it may grow the opposite direction on some other architectures). The stack is LIFO (last-in-first-out) data structure that serves as a collection of elements, with two principal operations:

- **push**, which adds an element to the collection, and
- **pop**, which removes the most recently added element that was not yet removed.

This area is devoted to storing all the data needed by a **function call** in a program. Calling a function is the same as pushing the called function execution onto the top of the stack, and once that function completes, the results are returned popping the function off the stack. The dataset pushed for function call is named a stack frame, and it contains the following data.

- the arguments (parameter values) passed to the routine
- the return address back to the routine's caller
- space for the local variables of the routine

The following is an example of C program and picture of stack memory allocation.

Example 1:

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

int getNum1() {
  return 10;
}
int getNum2() {
  return 20;
}

int getResult() {
  int num1 = getNum1();
  int num2 = getNum2();
  return num1 + num2;
}

int main() {
  int result = getResult();
  cout<<result;
}
```

When the function is called, the stack frame is pushed to the top of stack. Then the process is executed and the function goes out of scope, the stack frame pops from the top.

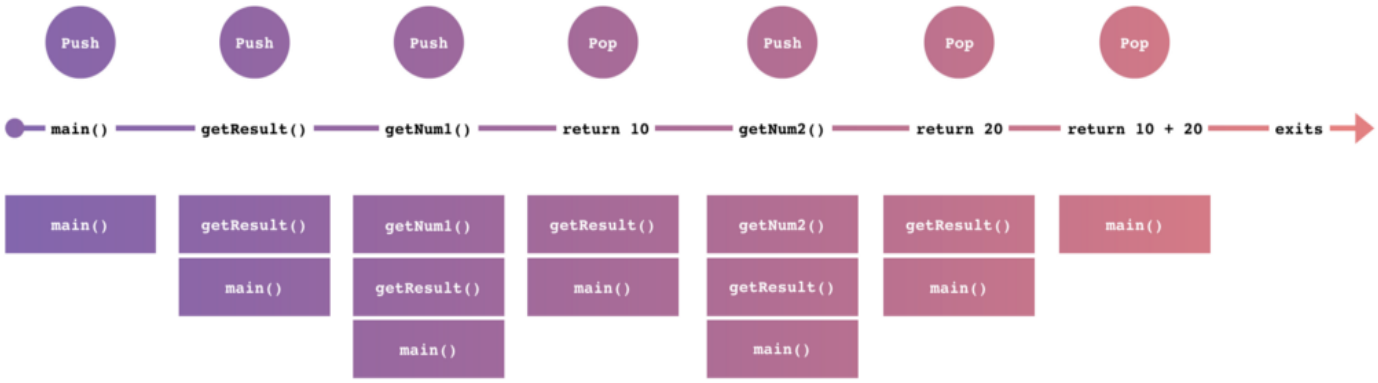


Fig 2: Stack operation

As described above, it can only store limited scope data. However, In memory management, **it runs very fast because the stack pointer register simply tracks the top of the stack**.

Heap

The Heap is the segment where dynamic memory allocation usually takes place. This area commonly begins at the end of the BSS segment and grows upwards to higher memory addresses. In C, it's managed by malloc / new, free / delete, which use the brk and sbrk system calls to adjust its size.

The allocation to the heap area occurs, in the following cases.

- memory size is dynamically allocated at run-time
- scope is not limited. (e.g. variables referenced from several places)
- memory size is very large.

It's our responsibility to free memory on the heap. The objects on the heap lead to memory leaks if they are not freed. In garbage-collected languages, the garbage collector frees memory on the heap and prevents memory leaks.

BSS (Block Started by Symbol)

Uninitialized data segment, often called the BSS segment. Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing. For instance, a variable declared as static int i; would be allocated to the BSS segment.

Data

The data segment contains initialized global and static variables which have a pre-defined value and can be modified. it's divided into a read-only and a read-write space. For example, the following C++ program outside the main

int val = 3;

char string[] = "Hello World";

Text

A text segment, also known as a code, is one of the sections of a program in an object file or in memory, which contains executable instructions. As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C++ compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

3.1.3 Stack vs Heap

The stack is faster because all free memory is always contiguous. Unlike heap, no list need to keep a list of all the free memory in stack, only one pointer to the current top of the stack. Each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast. Therefore, I recommend using stack as long as you don't need to use heap.

3.2 Memory Allocation

There are essentially two types of memory allocation:

1. Static - Done by the compiler automatically (implicitly).
2. Dynamic - Done explicitly by the programmer.

3.2.1 Static:

- Global variables or objects (memory is allocated at the start of the program, and freed when program exits; alive throughout program execution and can be access anywhere in the program)
- Local variables (memory is allocated when the routine starts and freed when the routine returns and cannot be accessed from another routine)
- Allocation and free are done implicitly. No need to explicitly manage memory is nice (easy to work with), but has limitations!
- Using static allocation, the array size must be fixed. E.g. Consider the grade roster for the class? What is the number of people in the class? Wouldn't it be nice to be able to have an array whose size can be adjusted depending on needs. Dynamic memory allocation deals with this situation.

3.2.2 Dynamic:

1. Programmer explicitly requests the system to allocate memory and return starting address of memory allocated (what is this?). This address can be used by the programmer to access the allocated memory.
2. When done using memory, it must be explicitly freed.

3.3 Dynamic Memory Allocation:

Dynamic allocation requires two steps:

1. Creating the dynamic space.
2. Storing its **address** in a **pointer** (so that the space can be accessed)

To dynamically allocate memory in C++, we use the **new** operator.
De-allocation:

- Deallocation is the "clean-up" of space being used for variables or other data storage
- Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
- It is the programmer's job to deallocate dynamically created space
- To de-allocate dynamic memory, we use the **delete** operator

3.3.1 Allocating space with new

To allocate space dynamically, use the unary operator **new**, followed by the *type* being allocated.

```
new int;       // dynamically allocates an int
new double;    // dynamically allocates a double
```

If creating an array dynamically, use the same form, but put brackets with a size after the type:

```
new int[40];      // dynamically allocates an array of 40 int
new double[size]; // dynamically allocates an array of size double and the size can be a variable
```

These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:

```
int * p;       // declare a pointer p
p = new int;   // dynamically allocate an int and load address into p

double * d;    // declare a pointer d
d = new double; // dynamically allocate a double and load address into d

// we can also do these in single line statements
int x = 40;
int * list = new int[x];
float * numbers = new float[x+10];
```

Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

3.3.2 Accessing dynamically created space

So once the space has been dynamically allocated, how do we use it? For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

```
int * p = new int;                    // dynamic integer, pointed to by p

*p = 10;                              // assigns 10 to the dynamic integer
cout << *p;                           // prints 10
```

For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:

```
double * numList = new double[size];          // dynamic array

for (int i = 0; i < size; i++)
   numList[i] = 0;                    // initialize array elements to 0

numList[5] = 20;                      // bracket notation
*(numList + 7) = 15;                  // pointer-offset notation. means same as
numList[7]
```

3.3.3 Deallocation of dynamic memory

To deallocate memory that was created with new, we use the unary operator **delete**. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int;                  // dynamically created int
// ...
delete ptr;                           // deletes the space that ptr points to
```

Note that the pointer ptr *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

```
ptr = new int[10];            // point p to a brand new array
```

To deallocate a dynamic array, use this form:

delete [] *name_of_pointer*;

Example:

```
int * list = new int[40];     // dynamic array
delete [] list;               // deallocates the array
list = 0;                             // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

To consider: So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

Example 2

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

int main () {
  double* pvalue  = NULL; // Pointer initialized with null
```

```
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```

Output:

```
Value of pvalue : 29495
```

Example 3

```
// Dynamically Allocate Memory for 1D Array in C++

#include <iostream>

#define N 10

int main()
{
        // dynamically allocate memory of size N
        int* A = new int[N];

        // assign values to allocated memory
        for (int i = 0; i < N; i++)
                A[i] = i + 1;

        // print the 1D array
        for (int i = 0; i < N; i++)
                std::cout << A[i] << " ";        // or *(A + i)

        // deallocate memory
        delete[] A;

        return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

3.3.4 Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

3.3.5 Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else. For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

I want to resize this so that the array called **list** has space for 5 more numbers (presumably because the old one is full). There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).
   ```
   int * temp = new int[size + 5];
   ```
2. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.
   ```
   for (int i = 0; i < size; i++)
       temp[i] = list[i];
   ```
3. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)
   ```
   delete [] list;  // this deletes the array pointed to by "list"
   ```
4. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.
   ```
   list = temp;
   ```

That's it! The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.

Example 4

```
#include<iostream>
using namespace std;
int main()
{
        int max = 5;          // no longer const
        int* a = new int[max];  // allocated on heap
        int n = max;
        char check;
```

```
        //--- Read into the array
        cout<<"Enter 5 values: "<<endl;
        for(int i= 0; i<max; i++)
        cin>>a[i];

        do{
                cout<<endl<<"Wanna enter more values??(y/n)  ";
                cin>>check;

                 if (check == 'y' || check == 'Y') {
                      max = max +1;           // increment in the previous size
                       int* temp = new int[max]; // create new bigger array.
                       for (int i=0; i<n; i++) {
                          temp[i] = a[i];       // copy values to new array.
                       }

                       delete [] a;             // free old array memory.
                       a = temp;                // now a points to new array.

                       cout<<"Enter another value. ";
                       cin>>a[n];
                       n++;
                }
               else break;
        } while (check=='y'|| check =='Y');

        //--- Write out the array etc.
        cout<<endl<<"Array Values:  ";
        for(int i=0;i<n;i++)
        cout<<a[i]<<" ";

}
```

Output:

```
Enter 5 values:
1
2
3
4
5

Wanna enter more values??(y/n)  y
Enter another value. 6

Wanna enter more values??(y/n)  y
Enter another value. 7
```

```
Wanna enter more values??(y/n)  n

Array Values:  1 2 3 4 5 6 7
```

Lab 04

Introduction to Object Oriented Programming and Classes

4.1 Object Oriented Programming - Introduction

Object oriented programming is a way of solving complex problems by breaking them into smaller problems using objects. Before Object Oriented Programming (commonly referred as OOP), programs were written in procedural language, they were nothing but a long list of instructions. For starters, there are two basic but vital concepts you have to understand in OOP namely Classes and Objects. Other important features of oop are data hiding, encapsulation, inheritance, overloading, abstraction, polymorphism etc.

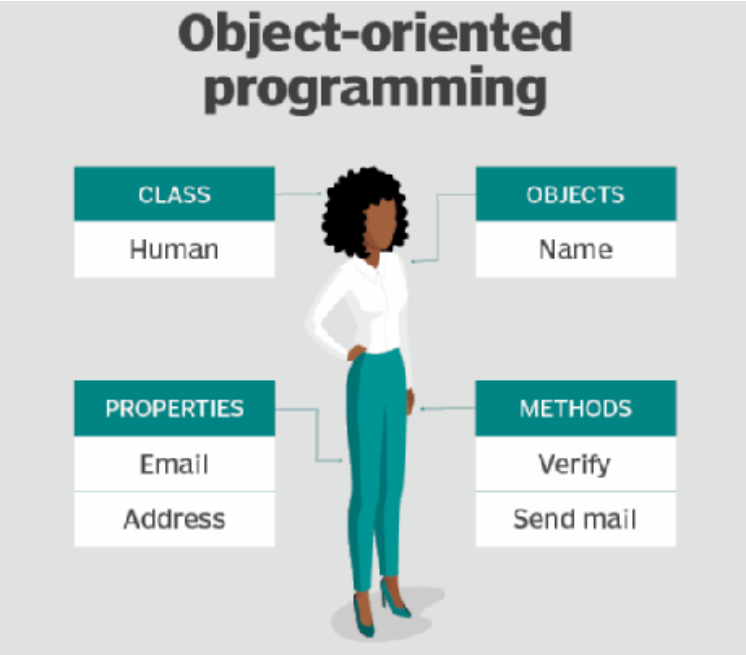


Fig 1: Oop concept

4.1.1 Classes and Objects

The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties. Now let's say I create an object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly, we can create as many objects as we want using the blueprint(class).

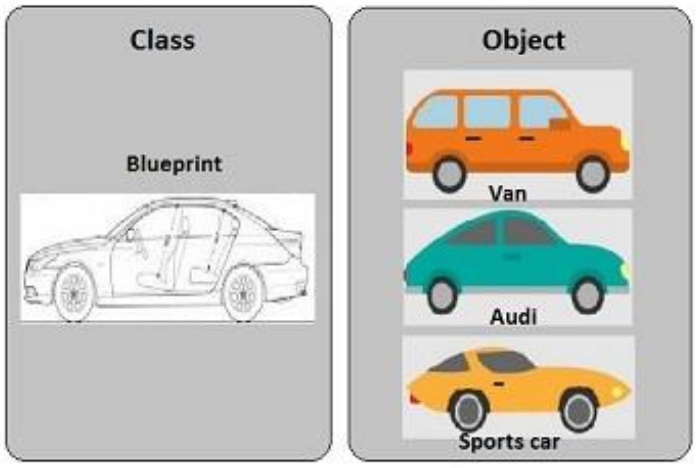


Fig 2: Classes and Objects

In the above example of class Car, the data member will be speed limit, mileage etc. and member functions can be applying brakes, increase speed etc.
An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

4.1.2 Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

4.1.3 Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.
In C++, we use Function overloading and Function overriding to achieve polymorphism.

4.1.4 Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. In C++, we use abstract class and interface to achieve abstraction.

4.1.5 Encapsulation

Wrapping up(combing) of data and functions into a single unit is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapping in the class can access it. This insulation of the data from direct access by the program is called data hiding or information hiding.

4.1.6 Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

4.2 Advantage of OOP over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

4.3 Classes Structure

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

4.3.1 A Class is a 3-Compartment Box encapsulating Data and Functions

A class can be visualized as a three-compartment box, as illustrated:

1. ***Classname*** (or identifier): identifies the class.
2. ***Data Members*** or ***Variables*** (or *attributes*, *states*, *fields*): contains the *static attributes* of the class.
3. ***Member Functions*** (or *methods*, *behaviors*, *operations*): contains the *dynamic operations* of the class.

In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

Class Members: The data members and member functions are collectively called class members.

The followings figure shows a few examples of classes:

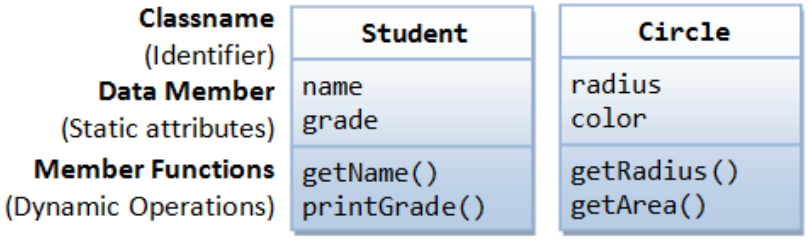


Fig 3: Class UML Diagram

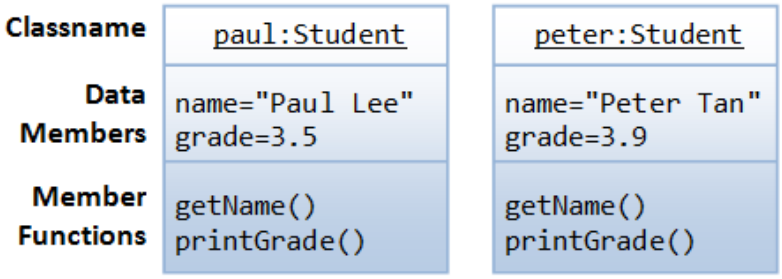


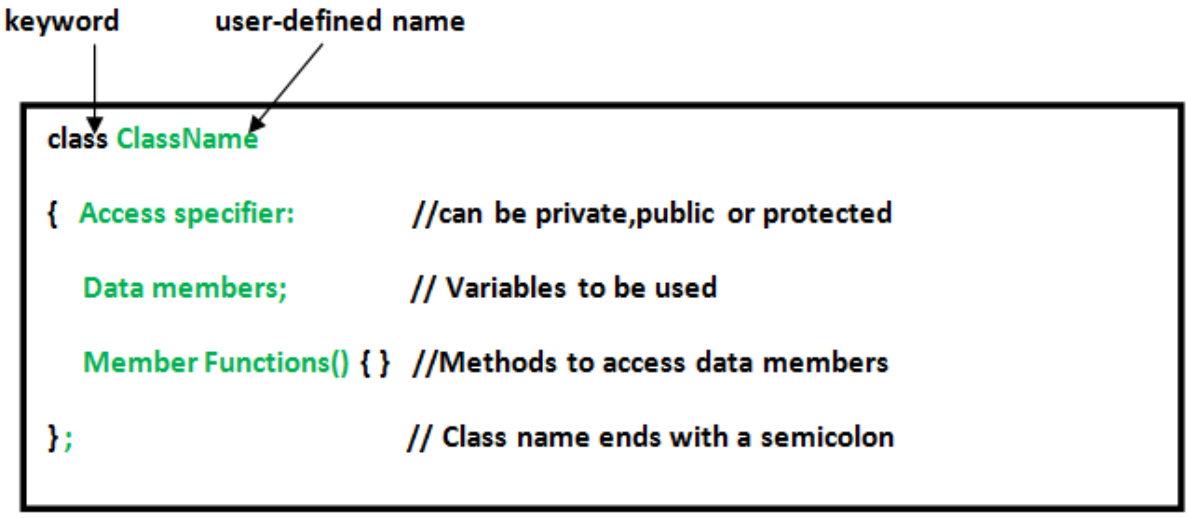
Fig 4: Instance UML Diagram

4.3.2 Unified Modeling Language (UML) Class and Instance Diagrams

The above class diagrams are drawn according to the UML notations. A class is represented as a 3-compartment box, containing name, data members (variables), and member functions, respectively. classname is shown in bold and centralized. An instance (object) is also represented as a 3-compartment box, with instance name shown as instanceName:Classname and underlined.

4.3.3 Class Definition

The syntax to declare a class is as under:



4.3.3.1 Data Members (Variables)

A *data member* (*variable*) has a *name* (or *identifier*) and a *type*; and holds a *value* of that particular type (as descried in the earlier chapter). A data member can also be an instance of a certain class (to be discussed later).

4.3.3.2 Member Functions

A member function:

1. receives parameters from the caller,
2. performs the operations defined in the function body, and
3. returns a piece of result (or void) to the caller.

4.3.3.3 Access Specifiers:

Access specifiers define how members of a class can be accessed. There are three access specifiers for a class in C++:

1. **public** — The public members are accessible from outside the class through an object of the class, typically with the use of the *dot operator*.
2. **protected** — The protected members can be accessed from outside the class but only in a class derived from it.
3. **private** — The private members are only accessible from within the class or in its member functions. Using a dot operator to access private members will result in the compiler throwing an error.

Example 1

```
/*  Simple Class Example Program In C++
    Understanding Class                          */

// Header Files
#include <iostream>
#include<conio.h>

using namespace std;

// Class Declaration

class Person {
  //Access - Specifier
public:
  //Member Variable Declaration
  string name;
  int age;

  //Member Functions read() and print() Declaration

   void print() {
     //Show the Output
     cout << "Name : " << name << endl<<"Age : " << age << endl;
  }
};

int main() {
  // Object Creation For Class
  Person obj;

  cout << "Simple Class and Object Example Program In C++\n";
  cout << "Enter the Name :";
  cin >> obj.name;
```

```
  cout << "Enter the Age :";
  cin >> obj.age;

  obj.print();

  getch();
  return 0;
}
```

Output:

```
Simple Class and Object Example Program In C++
Enter the Name :Ali
Enter the Age :15

Name : Ali
Age : 15
```

4.3.4 getter and setter methods

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 **getter** and **setter** methods.

Example 2

```
#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;
}
```

Output:

```
50000
```

4.3.5 Class Function Defined Outside the class definition

Member functions can be defined within the class definition or separately using **scope resolution operator, :** −. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. You can define the same function outside the class using the scope resolution operator (::) as follows –

Example 3

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void) {
   return length * breadth * height;
}

void Box::setLength( double len ) {
   length = len;
}
void Box::setBreadth( double bre ) {
   breadth = bre;
```

```
}
void Box::setHeight( double hei ) {
  height = hei;
}

// Main function for the program
int main() {
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 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;
  return 0;
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Lab 05

Introduction to Classes- Part II

5.1 Constructors

Consider the following example:

Example 1

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

class CRectangle {
        int x, y;
public:
        void set_values (int,int);
        int area ()
        {
        return (x*y);
        }
};
void CRectangle::set_values (int a, int b) {
        x = a;
        y = b;
}

int main () {
        CRectangle rect, rectb;
        rect.set_values (3,4);
        rectb.set_values (5,6);
        cout << "rect area: " << rect.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        return 0;
}
```

Output:

```
rect area: 12
rectb area: 30
```

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void. We are going to implement CRectangle including a constructor:

Example 2:

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

class CRectangle {

int width, height;
public:
CRectangle (int,int);
int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
width = a;
height = b;
}

int main () {
CRectangle rect (3,4);
CRectangle rectb (5,6);
cout << "rect area: " << rect.area() << endl;
cout << "rectb area: " << rectb.area() << endl;
return 0;
}
```

Output:

```
rect area: 12 rectb area: 30
```

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created ***(such constructor is also called parameterized constructor)***:

CRectangle rect (3,4);
CRectangle rectb (5,6);

A constructor function is different from an ordinary function in the following aspects:

- The name of the constructor is the same as the classname.
- Constructor has no return type (or implicitly returns void). Hence, no return statement is allowed inside the constructor's body.
- Constructor can only be invoked once to initialize the instance constructed. You cannot call the constructor afterwards in your program.
- Constructors are not inherited (to be explained later).

5.1.1 Default Constructors

Default constructor is the constructor which doesn't take any argument. It has no parameter.

For example:

```
class Cube
{
   public:
   int side;
   Cube()
   {
      side = 10;
   }
};

int main()
{
   Cube c;
   cout << c.side;
}
```

Output is 10 for the above code. In this case, as soon as the object is created the constructor is called which initializes its data members. A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor **implicitly.** Consider the following example:

```
class Cube
{
   public:
   int side;
};

int main()
{
   Cube c;
   cout << c.side;
}
```

In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in this case.

5.1.2 Constructors with Default Values

You can also add constructors with default values for its parameter in the prototype. See the example below:

Example 3

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

class Box{
	private:
		double length;
		double breadth;
		double height;

	public:
		Box(double le =1.0, double br =1.0, double he =1.0)
		{
			length = le;
			breadth = br;
			height = he;
		}
		double volume()
		{
			return length * breadth * height;
		}
};
int main()
{
	Box Box1(4);
	Box Box2( 4,5);
	Box Box3(8, 5, 1);

	cout<<"Volume of Box1: "<<Box1.volume();
	cout<<endl<<"Volume of Box2: "<<Box2.volume();
	cout<<endl<<"Volume of Box3: "<<Box3.volume();
	return 0;

}
```

Output:

```
Volume of Box1: 4
Volume of Box2: 20
Volume of Box3: 40
```

Common mistakes when using Default argument in constructors/functions

1. void add(int a, int b = 3, int c, int d = 4);

   The above function will not compile. You cannot miss a default argument in between two arguments. In this case, c should also be assigned a default value.

2. void add(int a, int b = 3, int c, int d);

   The above function will not compile as well. You must provide default values for each argument after b. In this case, c and d should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one.
   void add(int a, int b, int c, int d = 4);

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose. If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.

5.1.3 Constructor's Member Initializer List

Instead of initializing the private data members inside the body of the constructor, as follows:

```
Circle(double r = 1.0, string c = "red") {
        radius = r;
        color = c;
}
```

We can use an alternate syntax called member initializer list as follows:

```
Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }
```

Member initializer list is placed after the constructor's header, separated by a colon (:). Each initializer is in the form of data_member_name(parameter_name). For fundamental type, it is equivalent to data_member_name = parameter_name. For object, the constructor will be invoked to construct the object. The constructor's body (empty in this case) will be run after the completion of member initializer list.
It is recommended to use member initializer list to initialize all the data members, as it is often more efficient than doing assignment inside the constructor's body. **Also member initializer list syntax is mandatory for initiating the const data members in the classes (we will study const data members in later chapters.)**

5.1.4 Copy Constructor

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

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its own copy constructor.
- **User Defined Copy constructor:** The programmer defines the user-defined constructor.

A user defined copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

Copy constructor is initiated by the following line of code written outside the class definition.

ClassName NewObj(OldObj);
OR
ClassName NewObj = OldObj;

Following is a simple example of user defined copy constructor.

Example 4

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

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;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

Default Copy Constructor
If user doesn't provide any version of copy constructor, compiler itself performs the same operation by providing implicit version of copy constructors. But this version only provides shallow copy of the object. User defined copy constructor always make deep copy of the objects.

- **Shallow copy**: both object data members are pointing to same memory location therefore has same values.
- **Deep copy:** both object data members have their own copies of the values in the memory.

5.2 Destructor

A destructor, similar to constructor, is a special function that has the same name as the classname, with a prefix ~, e.g., ~Circle(). Destructor is called implicitly when an object is destroyed. If you do not define a destructor, the compiler provides a default, which does nothing. If your class contains data member which is dynamically allocated (via `new` or `new[]` operator), you need to free the storage via `delete` or `delete[]`.

Example 5

```
class A
{
   A()
   {
      cout << "Constructor called";
   }
   // destructor
   ~A()
   {
      cout << "Destructor called";
   }
};
int main()
{
   A obj1;   // Constructor Called
   int x = 1
   if(x)
   {
      A obj2;  // Constructor Called
   }   // Destructor Called for obj2
} //  Destructor called for obj1
```

Output:

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

When an object is created the constructor of that class is called. The object reference is destroyed when its scope ends, which is generally after the closing curly bracket `}` for the code block in which it is created.

The object `obj2` is destroyed when the `if` block ends because it was created inside the `if` block. And the object `obj1` is destroyed when the `main()` function ends.

5.3 'this' Pointer

The **this** pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class. Let's take an example to understand this concept.

Example 6

```
#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;
}
```

Output:

```
100
A
```

5.4 The Rule of Three

This rule basically states that if a class defines one (or more) of the following, it should probably explicitly define all three, which are:

- destructor
- copy constructor
- copy assignment operator

By default, compiler generated version of up above methods works until programmer has not defined any of them. If programmer has implemented any of them, he/she should implement rest of them to avoid possible (not always) exceptions.

Compilers implicitly-generated constructor, destructor and operators simply provide shallow copy. Mostly programmer implement these methods to change/avoid this specific feature. If default behavior is intended to be shallow copy, there is no need of implement any of them. But if intended behavior is deep copy (or something else), it need to be implemented in all above operator and constructor to avoid nondeterministic nature of class.

Now, supposing our class does not have a copy constructor. Then copying an object will copy all of its data members to the target object. What happens when these objects are destroyed? The destructor runs twice. Also the destructor has the same information available for each object being destroyed. That sums up as, in the absence of a copy constructor, the execution of the destructor, which is supposed to happen once, happens twice. This duplicate execution, is a source for trouble.

Lab 06

Friend Function and Class, Static Members, Constant Objects and Functions, Composition

6.1 Friend Function

Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes. It works as bridge between classes. Note that friend declarations can go in either the public, private, or protected section of a class--it doesn't matter where they appear.

- Friend function must be declared with **friend** keyword.
- Friend function must be declared in all the classes from which we need to access private or protected members.
- Friend function will be defined outside the class without specifying the class name.
- Friend function will be invoked like normal function, without any object.

Example 1

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
using namespace std;
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0) { }
      //friend function
      friend int addFive(Distance);
};
// friend function definition
int addFive(Distance d)
{
   //accessing private data from non-member function
   d.meter += 5;
   return d.meter;
}
int main()
{
   Distance D;
   cout<<"Distance: "<< addFive(D);
   return 0;
}
```

Output:

```
Distance: 5
```

Here, friend function `addFive()` is declared inside `Distance` class. So, the private data *meter* can be accessed from this function.

6.2 Friend Classes

We can also make a class a friend of another class. In that case, all the member function of the class declared as friend become the friend functions of the other class. Let's see how to make a class a friend of another.

```
class A
{
   friend class B;

};

class B
{

};
```

Class B is declared as a friend of class A in the above code. So, now all the member functions of class B became friend functions of class A. Let's see an example of a class friend.

Example 2

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

class Square;

class Rectangle {
        int width, height;

public:
        Rectangle(int w = 1, int h = 1):width(w),height(h){}
        void display() {
                cout << "Rectangle: " << width * height << endl;
        };
        void morph(Square &);
};

class Square {
```

```
        int side;

public:
        Square(int s = 1):side(s){}
        void display() {
                cout << "Square: " << side * side << endl;
        };
        friend class Rectangle;
};

void Rectangle::morph(Square &s) {
        width = s.side;
        height = s.side;
}

int main () {
        Rectangle rec(5,10);
        Square sq(5);
        cout << "Before:" << endl;
        rec.display();
        sq.display();

        rec.morph(sq);
        cout << "\nAfter:" << endl;
        rec.display();
        sq.display();
        return 0;
}
```

Output:

```
Before:
Rectangle: 50
Square: 25

After:
Rectangle: 25
Square: 25
```

We declared **Rectangle** as a friend of **Square** so that **Rectangle** member functions could have access to the private member, **Square::side**

In our example, **Rectangle** is considered as a friend class by **Square** but **Rectangle** does not consider **Square** to be a friend, so **Rectangle** can access the private members of **Square** but not the other way around.

6.3 Static Members in the Class

When we declare a normal variable (data member) in a class, different copies of those data members create with the associated objects.
In some cases, when we need a common data member that should be same for all objects, we cannot do this using normal data members. To fulfill such cases, we need static data members. Now we are going to learn about the static data members and static member functions, how they declare, how they access with and without member functions?

6.3.1 C++ static data member

It is a variable which is declared with the static keyword, it is also known as class member, thus only single copy of the variable creates for all objects.

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. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to. Any changes in the static data member through one member function will reflect in all other object's member functions.

Declaration

static data_type member_name;

Defining the static data member

It should be defined outside of the class following this syntax:

data_type class_name :: member_name =value;

If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

Example 3

```
#include <iostream>
using namespace std;
class Box {
  public:

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
```

```
      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }
    static int getCount() {
      return objectCount;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
    static int objectCount;
};

int Box::objectCount = 0;
// Initialize static member of class Box

int main(void) {
  // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;

  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects after creating object.
  cout << "Final Stage Count: " << Box::getCount() << endl;

  return 0;
}
```

Output:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

6.3.2 Static Member Functions

By declaring a function member as static, you make it independent of any particular 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.

Let us try the following example to understand the concept of static function members −

Example 4

```
#include <iostream>

using namespace std;

class Box {
  public:
    static int objectCount;

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;

      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }
    static int getCount() {
      return objectCount;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
```

```
  // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;

  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects after creating object.
  cout << "Final Stage Count: " << Box::getCount() << endl;

  return 0;
}
```

Output:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

6.4 "Const" Keyword in Classes

Constant is something that doesn't change. In C/C++ we use the keyword const to make program elements constant. const keyword can be used in many contexts in a C++ program e.g.:

1. Variables
2. Pointers
3. Function arguments and return types
4. Class Data members
5. Class Member functions
6. Objects

6.4.1 Defining Class Data members as const

These are data variables in class which are defined using const keyword. They are not initialized during declaration. Their initialization is done in the constructor.

Example 5

```
class Test
{
   const int i;
   public:
   Test(int x):i(x)
   {
      cout << "\ni value set: " << i;
   }
};
```

```
int main()
{
   Test t(10);
   Test s(20);
}
```

In this program, i is a constant data member, in every object its independent copy will be present, hence it is initialized with each object using the constructor. And once initialized, its value cannot be changed. The above way of initializing a class member is known as Initializer List in C++ because const data members can only be initialized using **member initializer list** syntax.

6.4.2 Defining Class Object as const

When an object is declared or created using the const keyword, its data members can never be changed, during the object's lifetime.

Syntax:

const class_name object;

For example, if in the class Test defined above, we want to define a constant object, we can do it like:

const Test r(30);

6.4.3 Const member functions in Classes

A function becomes const when const keyword is used in function's declaration. The idea of const functions does not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided. When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects only otherwise will give compile time error if called by const objects.

Following is a simple example of const function.

Example 6

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

class Test {
   int value;
public:
   Test(int v = 0) {value = v;}

   // We get compiler error if we add a line like "value = 100;"
   // in this function.
   int getValue() const {return value;}
```

```
};

int main() {
   Test t(20);
   cout<<t.getValue();
   return 0;
}
```

Output:

```
20
```

6.5 Types of Relationships in Object Oriented Programming (OOP)

One of the advantages of Object-Oriented programming language is code reuse. This reusability is possible due to the relationship b/w the classes. It generally supports 4 types of relationships that are: inheritance, association, aggregation and composition.
Inheritance is based on "is a" relationship, association and aggregation normally based on "has a" relationship and composition belongs to "part of" relationship. We will study inheritance in later chapters.

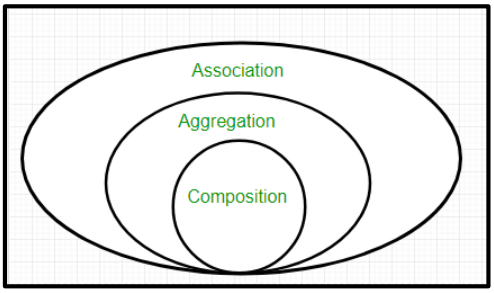


Fig 1: Relationship in OOP

6.5.1 Association, Aggregation and Composition

Association refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each other's functionality. Composition and aggregation are two types of association. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

Aggregation

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, a Team object and a Player object. The team contains multiple players but a player can exist without a team.

Example 7

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

```
   public:
  string addressLine, city, province;
    Address(string addressLine, string city, string province)
   {
      this->addressLine = addressLine;
      this->city = city;
      this->province = province;
   }
};
class Employee
   {
      private:
      Address* address;  //Employee HAS-A Address
      public:
      int id;
      string name;
      Employee(int id, string name, Address* address)
      {
         this->id = id;
         this->name = name;
         this->address = address;
      }
    void display()
      {
         cout<<id <<" "<<name<< " "<<
           address->addressLine<< " "<< address->city<< " "<<address->province<<endl;
      }
  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Swabi","KPK");
   Employee e1 = Employee(101,"Ali",&a1);
         e1.display();
  return 0;
}
```

Output:

```
101 Ali C-146, Sec-15 Swabi KPK
```

Composition

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human. E.g.

```
//Car must have Engine
public class Car {
```

```
  //engine is a mandatory part of the car
  private final Engine engine;

  public Car () {
    engine = new Engine();
  }
}

//Engine Object
class Engine {}
```

Example 8

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

class Id {
  public:
                Id(double i) :id(i) {}
                void printId() {
        cout << "your Id is : " << id << endl;
                }
  private:
                double id;
};

class Money {
public :
        Money (int m):money(m) {}
        void printMoney() {
                cout << "you have " << money << "$ " << endl;
        }
private :
        int money;
};

class Country {
  public :
                Country( string c):country(c) {}
           void printCountry() {
          cout << country << endl;
                }

  private :
                string country;
};
```

```
class Person {
  public :
                Person(string n, Id ii, Money mm, Country cc):name(n), identification(ii),
cash(mm),state(cc) {}

                void printInfoPerson() {
                        cout << name << endl;
                        state.printCountry();
                        identification.printId();
                        cash.printMoney();
                }
        private:
                string name;
                Id identification;
                Money cash;
                Country state;
};
int main()
{
   Country state("Italy");
   Id identification( 0.001);
   Money cash( 55);
   Person p("David", identification, cash, state);
   p.printInfoPerson();
}
```

Output:

```
David
Italy
your Id is : 0.001
you have 55$
```

Lab 07

Operator Overloading

7.1 Operators Overloading

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively. When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

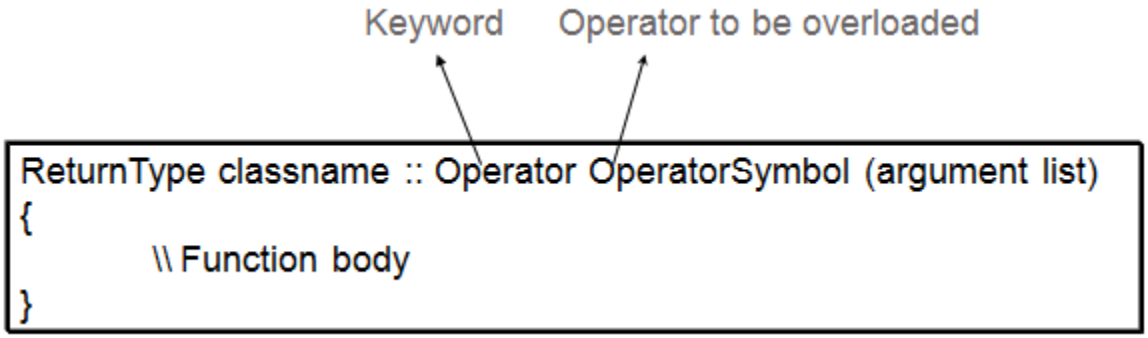
The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used. However, for user-defined types (like: objects), you can redefine the way operator works.

7.1.1 Why is operator overloading needed?

You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example, You can replace the code like:

calculation = add(multiply(a, b),divide(a, b));
to
calculation = (a*b)+(a/b);

7.1.2 Operator Overloading Syntax



For example, to overload the binary + operator for the class Time is as followed:

Time Time::operator+(Time t1)
or
Time operator+(Time t1)

7.1.4 Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

- Member Function
- Non-Member Function
- Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function. Operator overloading function can be made friend function if it needs access to the private and protected members of class.

7.1.5 Restrictions on Operator Overloading

- The overloaded operator must be an existing and valid operator. You cannot create your own operator such as ⊕.
- Certain C++ operators cannot be overloaded, such as sizeof, dot (. and .*), scope resolution (::) and conditional (?:).
- The overloaded operator must have at least one operands of the user-defined types. You cannot overload an operator working on fundamental types. That is, you can't overload the '+' operator for two ints (fundamental type) to perform subtraction.
- You cannot change the syntax rules (such as associativity, precedence and number of arguments) of the overloaded operator.

7.2 Binary Operator Overloading

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

Example 1

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

class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i =0)  {real = r;   imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
```

```
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

```
12 + i9
```

7.3 Unary Operator Overloading

The unary operators operate on a single operand and following are the examples of Unary operators −

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Example 2

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

class Counter{
        private:
                int count;
        public:
                Counter(){ count=0; }
                Counter(int c):count(c){ }
                int getCount(){  return count;   }

                //PreFix Unary Operator
                Counter operator++(){
                        Counter temp;
```

```
                temp.count = ++count;
                return temp;
        }
        //postfix unary operator
        Counter operator++(int){
                return Counter(count++);
        }
};
int main(){
        Counter c1,c2;

        cout<<"\nc1="<<c1.getCount();
        cout<<"\nc2="<<c2.getCount();

        ++c1;           //c1=1
        c2=++c1;        //c1=2 , c2=2 (prefix)

        cout<<"\nc1="<<c1.getCount();
        cout<<"\nc2="<<c2.getCount();

        c2 = c1++;     //c1=3; c2=2 (postfix)

        cout<<"\nc1="<<c1.getCount();
        cout<<"\nc2="<<c2.getCount();
}
```

Output:

```
c1=0
c2=0
c1=2
c2=2
c1=3
c2=2
```

7.4 Relational Operators Overloading

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.
You can overload any of these operators, which can be used to compare the objects of a class.

Example 3

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

class Distance {
```

```
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }

    // overloaded < operator
    bool operator <(const Distance& d) {
      if(feet < d.feet) {
        return true;
      }
      if(feet == d.feet && inches < d.inches) {
        return true;
      }
      return false;
    }
};
int main() {
  Distance D1(11, 10), D2(5, 11);

  if( D1 < D2 ) {
    cout << "D1 is less than D2 " << endl;
  } else {
    cout << "D2 is less than D1 " << endl;
  }
  return 0;
}
```

Output:

```
D2 is less than D1
```

7.5 Input/Output Operators Overloading

The first question before learning how to override the I/O operator should be, why we need to override the I/O operators. Following are a few cases, where overloading the I/O operator proves useful:

- We can overload output operator << to print values for user defined datatypes.
- We can overload output operator >> to input values for user defined datatypes.

In case of input/output operator overloading, left operand will be of types ostream& and istream&. Also, when overloading these operators, we must make sure that the functions must be a Non-Member function because left operand is not an object of the class and it must be a friend function to access private data members.

You have seen above that << operator is overloaded with ostream class object cout to print primitive datatype values to the screen, which is the default behavious of << operator, when used with cout. In other words, it is already overloaded by default.

Similarly we can overload << operator in our class to print user-defined datatypes to screen. For example we can overload << in our Time class to display the value of Time object using cout rather than writing a custom member function like show() to print the value of Time class objects.
Time t1(3,15,48);
// like this, directly
cout << t1;

NOTE: When the operator does not modify its operands, the best way to overload the operator is via friend function.

Example 4

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

class Distance {
  private:
    int feet;            // 0 to infinite
    int inches;          // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      f = i;
```

```
    }
    friend ostream &operator<<( ostream &output, const Distance &D ) {
      output << "F : " << D.feet << " I : " << D.inches;
      return output;
    }

    friend istream &operator>>( istream  &input, Distance &D ) {
      input >> D.feet >> D.inches;
      return input;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11), D3;

  cout << "Enter the value of object : " << endl;
  cin >> D3;
  cout << "First Distance : " << D1 << endl;
  cout << "Second Distance :" << D2 << endl;
  cout << "Third Distance :" << D3 << endl;

  return 0;
}
```

Output:

```
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

7.6 Assignment Operator Overloading

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.
Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. And assignment operator is called when an already initialized object is assigned a new value from another existing object. The example shows the difference between two:

Example 5

```
#include<iostream>
#include<stdio.h>

using namespace std;
```

```
class Test
{
public:
   Test() {}
   Test(const Test &t)
   {
      cout<<"Copy constructor called "<<endl;
   }
   Test& operator = (const Test &t)
   {
      cout<<"Assignment operator called "<<endl;
   }
};

int main()
{
  Test t1, t2;
  t2 = t1;
  Test t3 = t1;
  getchar();
  re*/*turn 0;
}
```

Output:

```
Assignment operator called
Copy constructor called
```

t2 = t1; // calls assignment operator, same as "t2.operator=(t1);"
Test t3 = t1; // calls copy constructor, same as "Test t3(t1);"

7.7 Subscripting [] Operator Overloading

The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays. Following example explains how a subscript operator [] can be overloaded.

Example 6

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
   private:
      int arr[SIZE];
```

```
  public:
    safearay() {
        register int i;
        for(i = 0; i < SIZE; i++) {
        arr[i] = i;
      }
    }

    int &operator[](int i) {
      if( i > SIZE ) {
         cout << "Index out of bounds" <<endl;
         // return first element.
         return arr[0];
      }

      return arr[i];
    }
};

int main() {
  safearay A;

  cout << "Value of A[2] : " << A[2] <<endl;
  cout << "Value of A[5] : " << A[5]<<endl;
  cout << "Value of A[12] : " << A[12]<<endl;

  return 0;
}
```

Output:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

Lab 08

Inheritance

8.1 What is Inheritance?

Reusability is one of the important characteristics of Object Oriented Programming (OOP). Instead of trying to write programs repeatedly, using existing code is a good practice for the programmer to reduce development time and avoid mistakes. In C++, reusability is possible by using Inheritance. The technique of deriving a new class from an old one is called inheritance. The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

8.1.1 Sub Class

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

8.1.2 Super Class

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

8.1.3 Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance, then we have to write all of these functions in each of the three classes as shown in below figure:

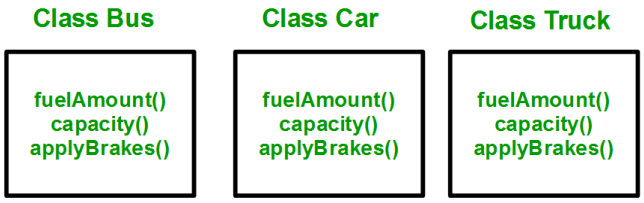


Fig 1: classes without inheritance

You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. Look at the below diagram in which the three classes are inherited from vehicle class:

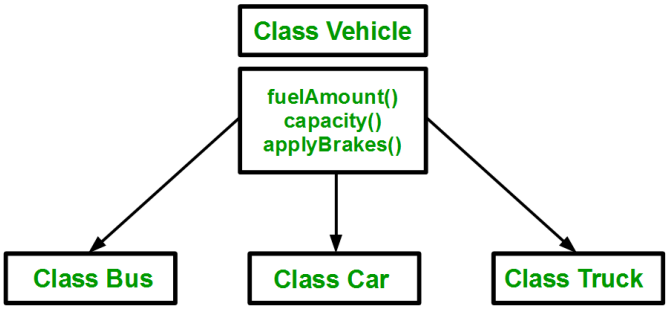


Fig 2: classes with inheritance

Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

8.1.4 Syntax for Implementing inheritance

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here access_mode **i**s the mode in which you want to inherit this sub class for example: public, private etc.

8.1.5 Modes of Inheritance

- **Public mode:** If we drive 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.
- **Protected mode:** If we drive 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.
- **Private mode:** If we drive 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.

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

Example 1

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};
```

```
// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;

  return 0;
}
```

Output:

```
Total area: 35
Total paint cost: $2450
```

8.2 Types of Inheritance

1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritance
4) Hierarchical inheritance

8.2.1 Single Inheritance:

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.

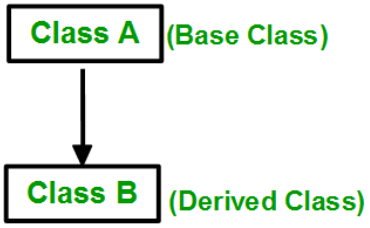


Fig 3: Single inheritance

Syntax:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};
```

Example 2

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

};
// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a vehicle
```

8.2.2 Multiple Inheritance:

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

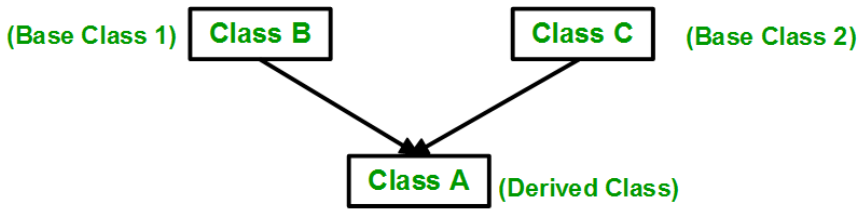


Fig 4: Multiple inheritance

Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

Example 3

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
```

```
class Car: public Vehicle, public FourWheeler {

};

int main()
{
   // creating object of sub class will
   // invoke the constructor of base classes
   Car obj;
   return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

8.2.3 Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class.

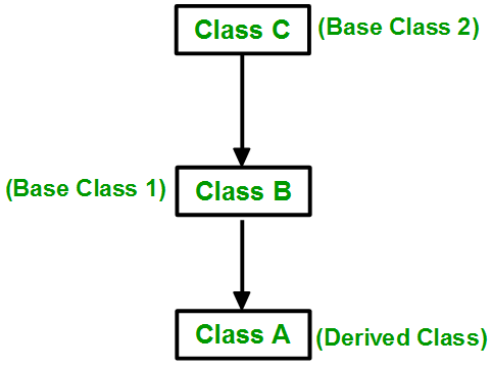


Fig 5: Multilevel inheritance

Example 4

```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
```

```
{  public:
   fourWheeler()
   {
     cout<<"Objects with 4 wheels are vehicles"<<endl;
   }
};
// sub class derived from two base classes
class Car: public fourWheeler{
  public:
    car()
    {
s       cout<<"Car has 4 Wheels"<<endl;
    }
};

int main()
{
   //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return 0;
}
```

Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

8.2.4 Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

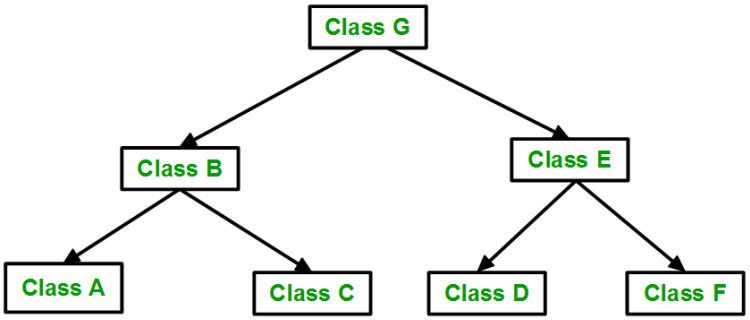


Fig 6: Hierarchical Inheritance

Example 5

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

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

Lab 09

Type Casting- Implicit Casting, Explicit Casting

9.1 Type Conversion

The process of converting one predefined type into another is called as type conversion. When constants and variables of different types are mixed in an expression, they are converted to the same type. When variables of one type are mixed with variables of another type, a type conversion will occur. As an operator, a cast is unary and has the same precedence as any other unary operator. C++ facilitates the type conversion into the following two forms:

- Implicit Type Conversion
- Explicit Type Conversion

9.2 Implicit Type Conversion

An implicit type conversion is a conversion performed by the compiler without programmer's intervention. An implicit conversion is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information. The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable).

Therefore, the types of right side and left side of an assignment should be compatible so that type conversion can take place. The compatible data types are mathematical data types i.e., char, int ,float, double. For example, the following statement:

`ch = x ;` (where ch is char and x is int)
Converts the value of x i.e., integer to bit into ch, a character. Assuming that word size is 16-bit (2 bytes), the left higher order bits of the integer variable x are looped off, leaving ch with lower 8-bits. Say, if x was having value 1417 (whose binary equivalent is 0000010110001001) the ch will have lower 8-bits i.e., 10001001 resulting in loss of information which the compiler can signal with a warning. This can be avoided with an explicit conversion.

9.2.1 Arithmetic Conversion Hierarchy

All the data types of the variables are upgraded to the data type of the variable with largest data type.

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

The compiler first proceeds with promoting a character to an integer. If the operands still have different data types, then they are converted to the highest data type that appears in the following hierarchy chart:

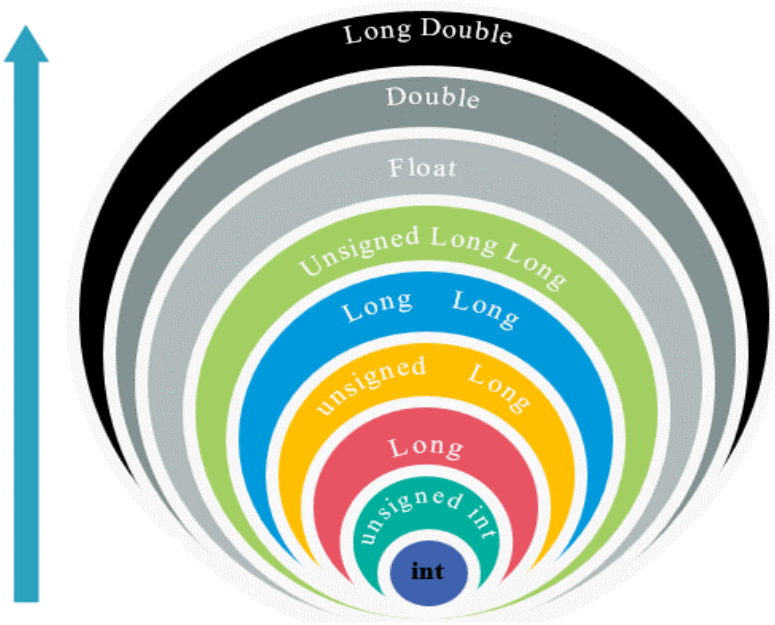


Fig 1: Conversion Hierarchy

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore, implicit conversions from A to B are allowed.

Example 1

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

int main()
{
   int x = 10; // integer x
   char y = 'a'; // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;
```

```
    // x is implicitly converted to float
    float z = x + 1.0;

    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;

    return 0;
}
```

Output:

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

9.3 Explicit Type Conversion

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

1) Converting by assignment

This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
Syntax: (type) expression
where type indicates the data type to which the final result is converted.

Example 2

```
// C++ program to demonstrate
// explicit type casting

#include <iostream>
using namespace std;

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    cout << "Sum = " << sum;

    return 0;
}
```

Output:

```
Sum = 2
```

2) Conversion using Cast operator

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream.h>

class CDummy {
   int i;
};

class CAddition {
        int x,y;
  public:
        CAddition (int a, int b) { x=a; y=b; }
        int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * ptrAdd;
  ptrAdd = (CAddition*) &d;
  cout << ptrAdd->result();
  return 0;
}
```

Although the previous program in syntactically correct in C++ (in fact it will compile with no warnings on most compilers) it is code with not much sense since we use function result, that is a member of CAddition, without having declared an object of that class: ptrAdd is not an object, it is only a pointer which we have assigned the address of a non-related object. When accessing its result member, it will produce a run-time error or, at best, just an unexpected result.

In order to control these types of conversions between classes, ANSI-C++ standard has defined four new casting operators.

reinterpret_cast <new_type> (expression)
dynamic_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)

Where new_type is the destination type to which expression has to be casted. To make an easily understandable parallelism with traditional type-casting operators these expression mean:

(new_type) expression
new_type (expression)

but with their own special characteristics. A Cast operator is an unary operator which forces one data type to be converted into another data type.

9.3.1 Dynamic Cast

Dynamic casts are only available in C++ and only make sense when applied to members of a class hierarchy ("polymorphic types"). Dynamic casts can be used to safely cast a superclass pointer (or reference) into a pointer (or reference) to a subclass in a class hierarchy. If the cast is invalid because the real type of the object pointed to is not the type of the desired subclass, the dynamic will fail gracefully.

Syntax: `dynamic_cast < type-id > (expression)`
Converts the operand expression to an object of type type-id.

Example 3

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {

   CBase * pba = new CDerived;
   CBase * pbb = new CBase;
   CDerived * pd;

   pd = dynamic_cast<CDerived*>(pba);
   if (pd==0) cout << "Null pointer on first type-cast" << endl;

   pd = dynamic_cast<CDerived*>(pbb);
   if (pd==0) cout << "Null pointer on second type-cast" << endl;


  return 0;
}
```

Output:

```
Null pointer on second type-cast
```

The code tries to perform two dynamic casts from pointer objects of type CBase* (pba and pbb) to a pointer object of type CDerived*, but only the first one is successful. Notice their respective initializations:

```
CBase * pba = new CDerived;
CBase * pbb = new CBase;
```

Even though both are pointers of type CBase*, pba points to an object of type CDerived, while pbb points to an object of type CBase. Thus, when their respective type-castings are performed using dynamic_cast, pba is pointing to a full object of class CDerived, whereas pbb is pointing to an object of class CBase, which is an incomplete object of class CDerived.

When dynamic_cast cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown instead.

dynamic_cast can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (void*).

9.3.2 Static Cast

The static_cast operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

In general, you use static_cast when you want to convert numeric data types such as enums to ints or ints to floats, and you are certain of the data types involved in the conversion. static_cast conversions are not as safe as dynamic_cast conversions, because static_cast does no run-time type check, while dynamic_cast does. A dynamic_cast to an ambiguous pointer will fail, while a static_cast returns as if nothing were wrong; this can be dangerous. Although dynamic_cast conversions are safer, dynamic_cast only works on pointers or references, and the run-time type check is an overhead.
In the example that follows, the line D* pd2 = static_cast<D*>(pb); is not safe because D can have fields and methods that are not in B. However, the line B* pb2 = static_cast<B*>(pd); is a safe conversion because D always contains all of B.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

static_cast can also be used to perform any other non-pointer conversion that could also be

performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

9.3.3 Reinterpret Cast

The Reinterpret cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

9.3.4 Const Cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

Example 4

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

void print (char * str)
{
```

```
  cout << str << endl;
}

int main () {
  const char * c = "Sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

Output:

```
Sample text
```

9.3.5 type_id

typeid allows to check the type of an expression:

typeid (expression)

This operator returns a reference to a constant object of type type_info that is defined in the standard header file <typeinfo>. This returned value can be compared with another one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

Example 5

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Output:

```
a and b are of different types:
a is: int *
b is: int
```

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

Example 6

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class CBase { virtual void f(){} };
class CDerived : public CBase {};

int main () {
   CBase* a = new CBase;
   CBase* b = new CDerived;
   cout << "a is: " << typeid(a).name() << '\n';
   cout << "b is: " << typeid(b).name() << '\n';
   cout << "*a is: " << typeid(*a).name() << '\n';
   cout << "*b is: " << typeid(*b).name() << '\n';

  return 0;
}
```

Output:

```
a is: class CBase *
b is: class CBase *
*a is: class CBase
*b is: class CDerived
```

Note: The string returned by member name of type_info depends on the specific implementation of your compiler and library. It is not necessarily a simple string with its typical type name, like in the compiler used to produce this output.
Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object).

If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.

The compiler in the examples above generates names with type_info::name that are easily readable by users, but this is not a requirement: a compiler may just return any string.

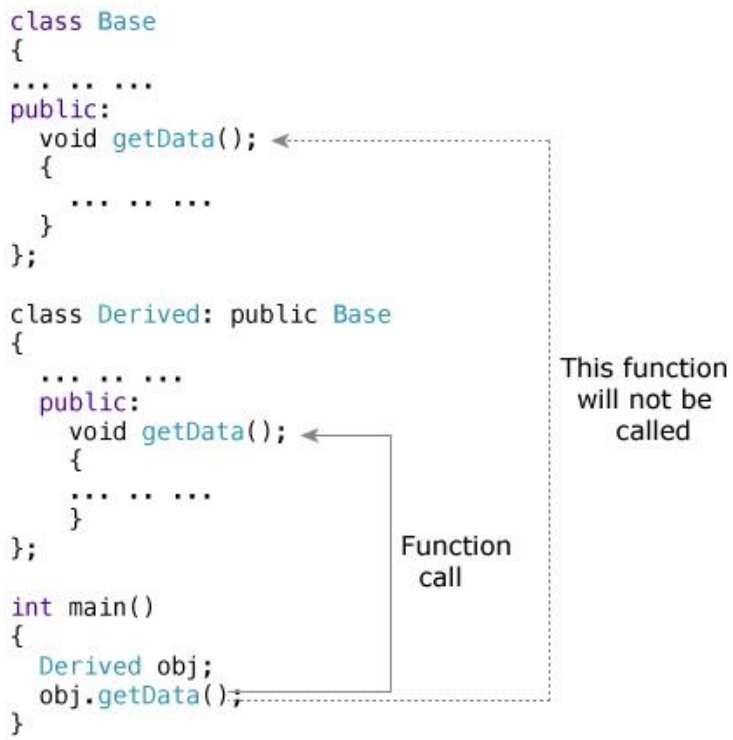
Lab 10

Overridden Functions, Virtual Functions, Abstract Base Class, Polymorphism

10.1 Overridden Functions

Inheritance allows software developers to derive a new class from the existing class. The derived class inherits features of the base class (existing class). Suppose, both base class and derived class have a member function with same name and arguments (number and type of arguments).

If you create an object of the derived class and call the member function which exists in both classes (base and derived), the member function of the derived class is invoked and the function of the base class is ignored. This feature in C++ is known as function overriding.
For example:

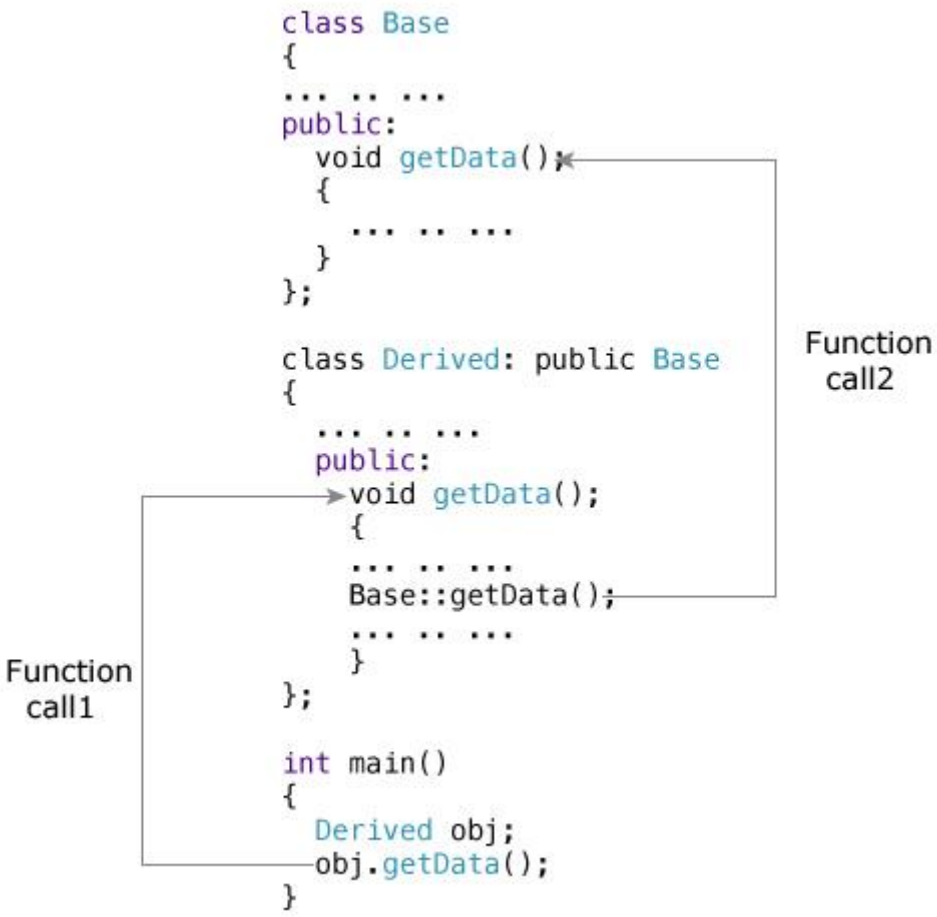


10.1.1 How to access the overridden function in the base class from the derived class?

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();



10.1.2 Function Call Binding with class Objects

Connecting the function call to the function body is called Binding. When it is done before the program is run, its called Early Binding or Static Binding or Compile-time Binding.

Example 1

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

class Base
{
   public:
   void shaow()
   {
```

```
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class\n";
    }
}

int main()
{
    Base b;       //Base class object
    Derived d;     //Derived class object
    b.show();     //Early Binding Ocuurs
    d.show();
}
```

Output:

```
Base class
Derived class
```

In the above example, we are calling the overrided function using Base class and Derived class object. Base class object will call base version of the function and derived class's object will call the derived version of the function.

10.1.3 Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

Example 2

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

class Base
{
    public:
    void show()
    {
        cout << "Base class\n";
    }
};
```

```
class Derived:public Base
{
   public:
   void show()
   {
      cout << "Derived Class\n";
   }
}

int main()
{
   Base* b;       //Base class pointer
   Derived d;     //Derived class object
   b = &d;
   b->show();     //Early Binding Occurs
}
```

Output:

```
Base class
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing Base class's pointer, set call to Base class's show() function, without knowing the actual object type. **Here there is a need of virtual functions.**

10.2 Virtual Functions in C++ and Polymorphism

A virtual function is a member function of the base class, that is overridden in derived class. The classes that have virtual functions are called polymorphic classes.

Polymorphism is the ability for objects of different classes related by inheritance to respond differently to the same member function call. The word polymorphism means having many forms.

Virtual Function tells the compiler to perform Late Binding on this function. In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called Dynamic Binding or Runtime Binding.

Example 3

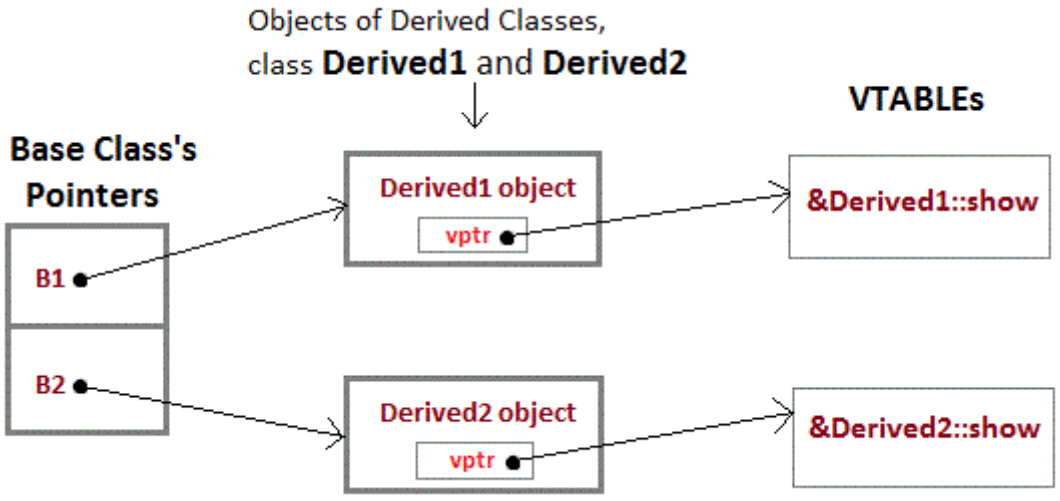
```
class Base
{
   public:
   virtual void show()
```

```
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
}

int main()
{
    Base* b;       //Base class pointer
    Derived d;     //Derived class object
    b = &d;
    b->show();     //Late Binding Ocuurs
}
```

Output:

```
Derived class
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.



10.2.1 Mechanism of Late Binding in C++

Vptr, is the vpointer, which points to the Virtual Fucntion for that object.
VTABLE, is the table containing address of virtual functions of each class.

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

10.3 Abstract Base Class and Pure Virtual Function in C++

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

10.3.1 Pure Virtual Functions in C++

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

Example 4

```
//Abstract base class
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
   public:
   void show()
   {
      cout << "Implementation of Virtual Function in Derived class\n";
   }
```

```
};

int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Output:

```
Implementation of Virtual Function in Derived class
```

In the above example Base class is abstract, with pure virtual show() function, hence we cannot create object of base class. One important thing to note is that, you should override the pure virtual function of the base class in the derived class. If you fail the override it, the derived class will become an abstract class as well.

10.3.3 Why can't we create Object of an Abstract Class?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an error message whenever you try to do so.

Example 5

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

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;
};
```

```
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;
}
```

Output:

```
Rectangle class area :
Triangle class area :
```

Lab 11

Introduction to Standard Template Library

11.1 C++ Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

11.1.1 What are generic functions or classes?

Many times while programming, there is a need for creating functions which perform the same operations but work with different data types. So C++ provides a feature to create a single generic function instead of many functions which can work with different data type by using the template parameter.

11.1.2 What is the template parameter?

The way we use normal parameters to pass as a value to function, in the same manner template parameters can be used to pass type as argument to function. Basically, it tells what type of data is being passed to the function.

11.1.3 Syntax for creating a generic function

```
template <class  type> return-type function-name (parameter-list)
```

Here, 'type' is just a placeholder used to store the data type when this function is used you can use any other name instead class is used to specify the generic type of template, alternatively typename can be used instead of it.

Let's try to understand it with an example:

Assume we have to swap two variables of int type and two of float type. Then, we will have to make two functions where one can swap int type variables and the other one can swap float type variables. But here if we use a generic function, then we can simply make one function and can swap both type of variables by passing their different type in the arguments. Let's implement this:

Example 1

```
#include <iostream>
using namespace std ;
// creating a generic function 'swap (parameter-list)' using template :
```

```
template <class X>
void swap( X &a, X &b) {
   X tp;
   tp = a;
   a = b;
   b = tp;
   cout << " Swapped elements values of a and b are  " << a << " and  " << b << " respectively "
<< endl;
}

int main( ) {
   int a = 10, b = 20 ;
   float c = 10.5, d = 20.5 ;
         swap(a , b);         // function swapping 'int' elements
   swap(c , d);               // function swapping 'float' elements
   return 0;
}
```

Output:

```
Swapped elements values of a and b are  20 and 10 respectively.
Swapped elements values of a and b are  20.5 and 10.5 respectively.
```

After creating the generic function, compiler will automatically generate correct code for the type of data used while executing the function.

11.1.4 Syntax for creating a generic class

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here −

template <class type> class class-name {

.

.

.

}

Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack –

Example 2

```
#include <iostream>
#include <vector>
#include <cstdlib>
```

```
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;    // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element

    bool empty() const {      // return true if empty.
      return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem) {
  // append copy of passed element
  elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
  if (elems.empty()) {
    throw out_of_range("Stack<>::pop(): empty stack");
  }

  // remove last element
  elems.pop_back();
}

template <class T>
T Stack<T>::top () const {
  if (elems.empty()) {
    throw out_of_range("Stack<>::top(): empty stack");
  }

  // return copy of last element
  return elems.back();
}
```

```
int main() {
  try {
    Stack<int>        intStack;  // stack of ints
    Stack<string> stringStack;    // stack of strings

    // manipulate int stack
    intStack.push(7);
    cout << intStack.top() <<endl;

    // manipulate string stack
    stringStack.push("hello");
    cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
  } catch (exception const& ex) {
    cerr << "Exception: " << ex.what() <<endl;
    return -1;
  }
}
```

Output:

```
7
hello
Exception: Stack<>::pop(): empty stack
```

11.2 STL in C++

STL is an acronym for standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms. STL is mainly composed of:

- Algorithms
- Containers
- Iterators

These are all generic class which can be used to represent collection of any data type.

11.2.1 Iterator

An iterator is any object that, points to some element in a range of elements (such as an array or a container) and has the ability to iterate through those elements using a set of operators (with at least the increment (++) and dereference (*) operators).

A pointer is a form of an iterator. A pointer can point to elements in an array, and can iterate over them using the increment operator (++). There can be other types of iterators as well. For each

container class, we can define iterator which can be used to iterate through all the elements of that container.
Example:
For Vector**:** **vector <int>::iterator it;**
For List**:** **list <int>::iterator it;**
You will see the implementations using iterators in the topics explained below.

11.2.2 Vector

Vectors are sequence containers that have dynamic size. In other words, vectors are dynamic arrays. Just like arrays, vector elements are placed in contiguous storage location so they can be accessed and traversed using iterators. To traverse the vector we need the position of the first and last element in the vector which we can get through begin() and end() or we can use indexing from 0 to size(). Let us see how to construct a vector.

```
vector<int> a;                          // empty vector of ints
vector<int> b (5, 10);                  // five ints with value 10
vector<int> c (b.begin(),b.end());      // iterating through second
vector<int> d (c);                      // copy of c
```

Some of the member functions of vectors are:
at(): Returns the reference to the element at a particular position (can also be done using '[ ]' operator). Its time complexity is O(1).

back(): Returns the reference to the last element. Its time complexity is O(1).

begin(): Returns an iterator pointing to the first element of the vector. Its time complexity is O(1).

clear(): Deletes all the elements from the vector and assign an empty vector. Its time complexity is O(N) where N is the size of the vector.

empty(): Returns a boolean value, true if the vector is empty and false if the vector is not empty. Its time complexity is O(1).

end(): Returns an iterator pointing to a position which is next to the last element of the vector. Its time complexity is O(1).

erase(): Deletes a single element or a range of elements. Its time complexity is O(N + M) where N is the number of the elements erased and M is the number of the elements moved.

front(): Returns the reference to the first element. Its time complexity is O(1).

insert(): Inserts new elements into the vector at a particular position. ts time complexity is O(N + M) where N is the number of elements inserted and M is the number of the elements moved .

pop_back(): Removes the last element from the vector. Its time complexity is O(1).

push_back(): Inserts a new element at the end of the vector. Its time complexity is O(1).

resize(): Resizes the vector to the new length which can be less than or greater than the current length. Its time complexity is O(N) where N is the size of the resized vector.

size(): Returns the number of elements in the vector. Its time complexity is O(1).

Example 3

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

int main() {

  // create a vector to store int
  vector<int> vec;
  int i;

  // display the original size of vec
  cout << "vector size = " << vec.size() << endl;

  // push 5 values into the vector
  for(i = 0; i < 5; i++) {
    vec.push_back(i);
  }

  // display extended size of vec
  cout << "extended vector size = " << vec.size() << endl;

  // access 5 values from the vector
  for(i = 0; i < 5; i++) {
    cout << "value of vec [" << i << "] = " << vec[i] << endl;
  }

  // use iterator to access the values
  vector<int>::iterator v = vec.begin();
  while( v != vec.end()) {
    cout << "value of v = " << *v << endl;
    v++;
  }
```

```
  return 0;
}
```

Output:

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

11.2.3 List

List is a sequence container which takes constant time in inserting and removing elements. List in STL is implemented as Doubly Link List. The elements from List cannot be directly accessed. For example, to access element of a particular position, you have to iterate from a known position to that particular position.

Declaration: list <int> LI;
Here LI can store elements of int type.

For example: list<int> LI(5, 100)
// here LI will have 5 int elements of value 100.

Some of the member function of List:
begin( ): It returns an iterator pointing to the first element in list.Its time complexity is O(1).
end( ): It returns an iterator referring to the theoretical element(doesn't point to an element) which follows the last element.Its time complexity is O(1).
empty( ): It returns whether the list is empty or not.It returns 1 if the list is empty otherwise returns 0.Its time complexity is O(1).
assign( ): It assigns new elements to the list by replacing its current elements and change its size accordingly.It time complexity is O(N).
back( ): It returns reference to the last element in the list.Its time complexity is O(1).
erase( ): It removes a single element or the range of element from the list.Its time complexity is O(N).
front( ): It returns reference to the first element in the list.Its time complexity is O(1).
push_back( ): It adds a new element at the end of the list, after its current last element. Its time complexity is O(1).
push_front( ): It adds a new element at the beginning of the list, before its current first element. Its time complexity is O(1).

remove( ): It removes all the elements from the list, which are equal to given element. Its time complexity is O(N).
pop_back( ): It removes the last element of the list, thus reducing its size by 1. Its time complexity is O(1).
pop_front( ): It removes the first element of the list, thus reducing its size by 1. Its time complexity is O(1).
insert( ): It insert new elements in the list before the element on the specified position. Its time complexity is O(N).
reverse ( ): It reverses the order of elements in the list. Its time complexity is O(N).
size( ): It returns the number of elements in the list. Its time complexity is O(1).

Example 4

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list <int> LI;
   list <int>::iterator it;
   //inserts elements at end of list
   LI.push_back(4);
   LI.push_back(5);

   //inserts elements at beginning of list
   LI.push_front(3);
   LI.push_front(5);

   //returns reference to first element of list
   it = LI.begin();

   //inserts 1 before first element of list
   LI.insert(it,1);

   cout<<"All elements of List LI are: " <<endl;
   for(it = LI.begin();it!=LI.end();it++)
   {
      cout<<*it<<" ";
   }
   cout<<endl;

   //reverse elements of list
   LI.reverse();

   cout<<"All elements of List LI are after reversing: " <<endl;
   for(it = LI.begin();it!=LI.end();it++)
```

```
    {
        cout<<*it<<" ";
    }
    cout<<endl;

    //removes all occurences of 5 from list
    LI.remove(5);

    cout<<"Elements after removing all occurence of 5 from List"<<endl;
    for(it = LI.begin();it!=LI.end();it++)
    {
        cout<<*it<<" ";
    }
    cout<<endl;

    //removes last element from list
    LI.pop_back();
    //removes first element from list
    LI.pop_front();
    return 0;
}
```

Output:

```
All elements of List LI are:
1 5 3 4 5
All elements of List LI are after reversing:
5 4 3 5 1
Elements after removing all occurrence of 5  from List
4 3 1
```

11.2.4 Stacks

Stack is a container which follows the LIFO (Last In First Out) order and the elements are inserted and deleted from one end of the container. The element which is inserted last will be extracted first.

Declaration: stack <int> s;
Some of the member functions of Stack are:
push( ): Insert element at the top of stack. Its time complexity is O(1).
pop( ): removes element from top of stack. Its time complexity is O(1).
top( ): access the top element of stack. Its time complexity is O(1).
empty( ): checks if the stack is empty or not. Its time complexity is O(1).
size( ): returns the size of stack. Its time complexity is O(1).

Example 5

```
#include <iostream>
#include <stack>

using namespace std;
int main( )
{
   stack <int> s;  // declaration of stack

   //inserting 5 elements in stack from 0 to 4.
   for(int i = 0;i < 5; i++)
   {
      s.push( i ) ;
   }

   //Now the stack is {0, 1, 2, 3, 4}

   //size of stack s
   cout<<"Size of stack is: " <<s.size( )<<endl;

   //accessing top element from stack, it will be the last inserted element.
   cout<<"Top element of stack is: " <<s.top( ) <<endl ;

   //Now deleting all elements from stack
   for(int i = 0;i < 5;i++)
   {
      s.pop( );
   }

   //Now stack is empty,so empty( ) function will return true.

   if(s.empty())
   {
      cout <<"Stack is empty."<<endl;
   }
   else
   {
      cout <<"Stack is Not empty."<<endl;
   }

   return 0;

}
```

Output:

```
Size of stack is: 5
Top element of stack is: 4
Stack is empty.
```

Lab 12

Open Ended Lab

An open-ended lab is where students are given the freedom to develop their own experiments, instead of merely following the already set guidelines from a lab manual or elsewhere.

Making labs open-ended pushes students to think for themselves and think harder. The students here have to devise their own strategies and back them with explanations, theory and logical justification. This not only encourages students to come up with their experiments, but requires them to defend themselves and their experiment, if questioned.

Different lab classes may vary in their degrees of open-endedness. However, there are three general areas that can be made open-ended:

Concept

In order to make this stage open-ended, the teacher may give the students a Problem Statement with a purpose and not the procedure. The students would then have to come up with their own experiments to back the theory or fulfil the purpose. This helps boost confidence in students, as they can proudly say that they did the experiment on their own.

Table. Schwab/Herron Levels of Laboratory Openness

LEVEL	PROBLEM	WAYS & MEANS/METHODS	ANSWERS
0	Given	Given	Given
1	Given	Given	Open
2	Given	Open	Open
3	Open	Open	Open

We use Schwab and Heron's scale of laboratory openness. In the first level the teacher posed a problem, which student then tried to answer by using different methods. In the second level the teacher posed a problem, but this time the students had to develop a method themselves. In the third level of inquiry the students had to both pose the problem and develop a suitable method.

Level

Our open-ended lab conforms to level 2 of Schwab/Herron Levels of Laboratory Openness.

VI. REFERENCES

Text Books:

- C++ How to Program by H M Deitel and P J Deitel.
- Object Oriented Programming in Turbo C++ by Robert Lafore
- Programming with C++ by D Ravichandran

Reference Books:

- Object Oriented Programming with C++ by E Balagurusamy
- Computing Concepts with C++ Essentials by Hortsmann
- The Complete Reference in C++ by Herbert Schildt

Designed by:

Amna Arooj,
Computer Engineer, FCSE
amna.arooj@giki.edu.pk