National University of Computer & Emerging Sciences, Karachi
Spring-2019 CS-Department

CS-217 Object-oriented Programming
Course Outline

Week	Topics	Lab Topic
1	Introduction to OO paradigm	Why C++, Data types, Looping and decision structures, scoping, types of operators
	Comparison from sequential & procedural paradigms	
	Intro to some of C++ IDE basics and program structure	
2	Structures vs. class	Pointers and functions
	Intro to Objects and its representation in memory	
3	Data Abstraction, Intro to Classes	Intro to classes & Objects
	Encapsulation	
	Concept of Abstract classes	
4	Access controls	Working with classes, Constructors & destructors
	Constructors and its types	
	Destructors	
5	Concept of Object reuse	Encapsulation, Access modifiers with data and functions
	Data and Object Casting	
6	Static data and member functions	Working with constants and casting
	Const data and member functions	
	Inline functions	
7	Inheritance	Working with abstract and static classes, functions
	Types of Inheritance	
8	Function overloading	Inheritance, abstract classes implementation
	Operator overloading	
9	Function overriding	Overloading
	Interface	
10	Multiple inheritance	Overriding
	Interfaces Vs. Abstract functions	
11	Friend classes and Functions	Multiple inheritance and friend functions.
12	Virtual functions	Virtual functions
13	IO stream	IO stream
14	Generic Programming	Working with templates
15	Exception Handling	Exception Handling
	Final Exam	

Books:

1- "Problem Solving with C++", 9e Global Edition, Walter Savitch, ISBN-13:9781292018249, Addison-Wesley, 2015.
2- C++ How to program By Deitel & Deitel.

Reference Books:

1- The C++ Programming Language by Bjarne Stroustrup.

2- Object Oriented Software Engineering by Jacobson.

Marks Distribution

Course:

Assignments	10%
Quizzes	10%
Mid Exam	30% (15% each)
Final Exam	50%
Total	100

Lab:

Lab activities	30%
Lab Mid exam	20%
Course Project	20% (Including viva exam)
Lab Final Exam	30%
Total	100

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 01

Outline

1. this Pointer
2. Constant Key word
3. Static Key Word
4. Examples
5. Exercise

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

- Object Oriented Programming (OOP) is a programming concept used in several modern programming languages, like C++, Java and Python.
- Object Oriented Programming works on the principle that objects are the most important part of a program. Manipulating these objects to get results is the goal of Object Oriented Programming.
- In OOP, the data is grouped together with the methods that operate upon it, which makes it easy to maintain the integrity of the data and provide a structure to the program.

Question: So what is an object oriented program exactly?

Answer: It is a program that contains objects, of course, which have certain properties and have methods linked to them. These methods are used to obtain a programming result.

WHY IS OOP NEEDED?

Problems with Procedural Languages

- Functions have unrestricted access to global data
- Unrelated Functions and data.

Before Object Oriented Programming programs were viewed as procedures that accepted data and produced an output. There was little emphasis given on the data that went into those programs.

DIFFERENCE BETWEEN C AND C++

The key differences include:

C	C++
It is a structural or procedural programming language.	It is an object oriented programming language.
Emphasis is on procedure or steps to solve a problem	Emphasis is on objects rather than procedure
Functions are the fundamental building blocks.	Objects are the fundamental building blocks.
In C, the data is not secured.	Data is hidden and can't be accessed by external functions.
C uses scanf() and printf() functions for standard input and output.	C uses cin>> and cout<< functions for standard input and output.
In C, namespace feature is absent.	In C++, namespace feature is present.
C program file is saved with .C extension.	C++ program file is saved with .CPP extension.

Table 1: Difference between C and C++

EXPLANATION OF BASIC C++ PROGRAM

An Example C++ Program

/* Comments can also be written starting with a slash followed by a star, and ending with a star followed by a slash. As you can see, comments written in this way can span more than one line. */ /* Programs should ALWAYS include plenty of comments! */ /* This program prints the table of entered number */

```
#include <iostream>
using namespace std;
int main()
{
        int input_num;
        //the number whose
        cout<<"Enter number";
        cin>>input_num;
        for (int i=0;i<=10;i++)
        {
                int output = input_num*i;
                cout<<input_num<<"*"<<i<<"="<<output<<endl;
        }
        return 0;
}
```

Program Output if 3 is entered as input to input_num

```
Enter number 3
3*0=0
3*1=3
3*2=6
3*3=9
3*4=12
3*5=15
3*6=18
3*7=21
3*8=24
3*9=27
3*10=30

--------------------------------
Process exited after 3.253 seconds with return value 0
Press any key to continue . . .
```

- ***The #include Directive***

The #include directive causes the contents of another file to be inserted into the program Preprocessor directives are not C++ statements and do not require semicolons at the end

- ***Using namespace std;***

The names cout and endl belong to the std namespace. They can be referenced via fully qualified namestd::cout and std::endl, or simply as cout and endl with a "using namespace std;" statement.

- ***return0;***

The return value of 0 indicates normal termination; while non-zero (typically 1) indicates abnormal

termination. C++ compiler will automatically insert a "return 0;" at the end of the the main () function, thus, it statement can be omitted.

- ***Output using cout***

- Cout is an object
- Corresponds to standard output stream
- << is called insertion or input operator

- ***Input With cin***

- Cin is an object
- Corresponds to standard input stream
- >> is called extraction or get from operator

Character	Name	Description
//	double slash	Marks the beginning of a comment
#	Pound sign	Marks the beginning of a preprocessor directive
<>	Opening and closing brackets	Encloses a filename when used with the #include directive
( )	Opening and closing parenthesis	Used in naming a function, as in int main ()
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
" "	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	Semicolon	Marks the end of a complete programming statement

Table 2: Mandatory symbols in basic program

COMMON ESCAPE SEQUENCES

Escape Sequence	Name	Description
\n	Newline	Causes the cursor to go to the next line for subsequent printing
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop
\b	Backspace	Causes the cursor to back up, or move left one position
\r	Return	Causes the cursor to go to the beginning of the current line, not the next line
\\	Backslash	Causes a backslash to be printed
\'	Single quote	Causes a single quotation mark to be printed
\"	Double quote	Causes a double quotation mark to be printed

Table 3: Escape Sequence

DATATYPES

There are many different types of data.
Variables are classified according to their data type, which determines the kind of information that may be stored in them. Integer variables only hold whole numbers.

Data Type	Size	Range
short	2 bytes	-32,768 to +32.767
unsigned short	2 bytes	0 to +65,535
Int	4 bytes	-2,147,4833,648 to +2,147,4833,647
unsigned int	4 bytes	0 to 4,294,967,295
Long	4 bytes	-2,147,4833,648 to +2,147,4833,647
Unsigned long	4 bytes	0 to 4,294,967,295

Table 4: Data types and size

Other Data Types

- ***Char Data Type***

- Usually 1 byte long
- Internally stored as an integer
- ASCII character set shows integer representation for each character
- 'A' == 65, 'B' == 66, 'C' == 67, etc
- Single quotes denote a character, double quotes denote a string

- ***Boolean Data Type***

Boolean variables are set to either true or false

OPERATORS

There are many operators in C++ for manipulating data which include arithmetic Operators, Relational Operators, Logical operators and many more which will be discussed accordingly.

➢ ***Arithmetic Operators***

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Table 5: Arithmetic Operators

➢ ***Relational Operators***

Operator	Description
= =	Equals to
!=	Not Equals to

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Table 6: Relational Operators

- ***Logical Operators***

Operator	Description
&&	Logical AND
||	Logical OR
!	NOT

Table 7: Logical Operators

- ***Increment and Decrement Operators***

C++ introduces increment and decrement operators which are ++ and – respectively. These operators increment/decrement 1 in the operand's value.

For example: x++ will be equivalent to x=x+1 or x+=1.

The special characteristic of these operators is that they can be used for pre-increment as well as post-increment. To understand, consider the following statements:

A=b++; *//The statement will assign the contents of b to A and then increments the value of b by 1*
A=++b; *//The statement will first increment the value of b by 1 and then assign the new value to A.*

- ***Assignment statements***

value = 5; *//This line is an assignment statement*

The assignment statement evaluates the expression on the right of the equal sign then stores it into the variable named on the left of the equal sign. The data type of the variable was in integer, so the data type of the expression on the right should evaluate to an integer as well.

DECISIONS

Sometimes, we want a program to choose among several possible alternative courses of action.
This means that some statements in the program may not be executed.
The choice between alternatives is based on some condition
A condition is either true or false
Use relational and/or logical operators to express a condition
Following are the main types of decision statements:

Statement	Description
If statement	An if statement consists of a boolean expression followed by one or more statement
If… else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false

nested if statements	You can use one if or else if statement inside another if or else if statement(s)
Switch statement	A switch statement allows a variable to be tested for equality against a list of values
Nested Switch statement	You can use one swicth statement inside another switch statement(s)

if statement

- ***Single statement if condition***

```
if (expression)
        statement;
```

Statement will be executed only if expression is true

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int x;
        cin >> x;
        if (x == 5)
                cout <<"Condition is true and the value of x is "<<x;
        return 0;
}
```

Program Output is 5 is enteredas input to x

```
5
Condition is true and the value of x is 5
--------------------------------
Process exited after 1.945 seconds with return value 0
Press any key to continue . . .
```

Program output if other than 5 is entered as input to x

```
3

--------------------------------
Process exited after 1.758 seconds with return value 0
Press any key to continue . . .
```

- ***Compound statement if condition***

Often, we want to execute several statements if a condition is true. Use braces to indicate the block of statements to be executed.

```
if (expression)
{
```

```
        statement1;
        statement2;
}
```

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int x;
        cin >> x;
        if (x == 5)
        {
                cout <<"Condition is true and the value of x is "<<x;
                cout <<"\nWelcome to NUCES-FAST";
        }

        return 0;
}
```

Program Output is 5 is enteredas input to x

```
5
Condition is true and the value of x is 5
Welcome to NUCES-FAST
--------------------------------
Process exited after 1.887 seconds with return value 0
Press any key to continue . . .
```

if else statement

```
if (expression)
        statement1;
else
        statement2;
```

Statement1 will be executed if expression is true Statement 2 will be executed if expression is false. Both statements will never be executed.

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int x;
        cin >> x;
        if (x == 5)
        {
```

```
                cout <<"Condition is true and the value of x is "<<x;
        }
        else
        {
                cout <<"Condition is false.";
        }
        return 0;
}
```

Program Output is 5 is enteredas input to x

```
5
Condition is true and the value of x is 5
--------------------------------
Process exited after 7.017 seconds with return value 0
Press any key to continue . . .
```

Program output if other than 5 is entered as input to x

```
3
Condition is false.
--------------------------------
Process exited after 1.671 seconds with return value 0
Press any key to continue . . .
```

Nested if statement

Possible to put one if or if-else statement inside another if or if-else statement

```
if (expression1)
{
        statement1;
        if (expression2)
        {
                statement2;
        }
}
```

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int age;
        cout<<"Enter your age: ";
        cin >> age;
        char gender;
        cout<<"\nEnter your gender i.e, M for male and F for female: ";
        cin >> gender;
```

```
        if (age < 4)
        {
                if(gender=='M')
                cout <<"\nA baby boy";

                else
                cout <<"\nA baby girl";
        }
        return 0;
}
```

Program output if age is less than 4 and gender is male

```
Enter your gender i.e, M for male and F for female: M

A baby boy
--------------------------------
Process exited after 5.248 seconds with return value 0
Press any key to continue . . .
```

Else if statement

```
if (expression1)
        statement 1;
else if (expression 2)
         statement 2;
...
else
        statement n;
```

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int percentage;
        cin >> percentage;
        if (percentage >= 50)
                cout << "You have passed";
        else if (percentage < 50)
                cout << "Try your best to clear the couse in next attempt."
        return 0;
}
```

Program output if percentage is greater than or equals to 50

```
55
You have passed
--------------------------------
Process exited after 3.106 seconds with return value 0
Press any key to continue . . .
```

Multiple conditions can be written by making several else-is clauses. Once a condition is true, control will never go to other else-if conditions. You can also add an else clause after else if statements.

ITERATIVE STATEMENTS (LOOPS)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.

- ***The for loop***

The for loop is designed to iterate a number of times.

Its syntax is:

for (initialization; condition; increase)
statement;

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        for (int n=10; n>0; n--)
        {
                cout << n << ", ";
        }
        cout << "\nEnd of for loop\n";
}
```

Program output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
End of for loop

--------------------------------
Process exited after 0.01807 seconds with return value 0
Press any key to continue . . .
```

- ***The while loop***

The simplest kind of loop is the while-loop.

Its syntax is:

while (expression
statement;

The while-loop simply repeats statement while expression is true. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop.

Sample Program

```
#include <iostream>
using namespace std;
int main()
{
        int n = 10;
        while (n>0)
        {
                cout << n << ", ";
                --n;
        }
        cout << "\nEnd of while loop";
}
```

Program output

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
End of while loop
--------------------------------
Process exited after 0.0153 seconds with return value 0
Press any key to continue . . .
```

- ***The do-while loop***

A very similar loop is the do-while loop, whose syntax is:

do
statement;
while (condition);

It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled.

Sample Program

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
```

```
        string str;
        do
        {
                cout << "Enter text: ";
                getline (cin,str);
                cout << "You entered: " << str << '\n';
        }
        while (str != "goodbye");
}
```

Program Output

```
Enter text: ITC Section D
You entered: ITC Section D
Enter text: ITC Section A
You entered: ITC Section A
Enter text: ITC Section B
You entered: ITC Section B
Enter text: ITC Section C
You entered: ITC Section C
Enter text: goodbye
You entered: goodbye

--------------------------------
Process exited after 46.12 seconds with return value 0
Press any key to continue . . .
```

Activities

Q1. Write a program that generates the following output using /= and += operators.

100
50
52
26
28

Q2. If you have two fractions, a/b and c/d, their sum can be obtained from the formula a/b + c/d = a*d + b*c / b*d
For example, 1/4 + 2/3 is
1/4 + 2/3 = 1*3 + 4*2 / 4*3 = 3+8 / 12 = 11/12
Write a program that encourages the user to enter two fractions and then displays their sum in fractional form. The interaction with the user might look like this:

Enter first fraction:
1/2 Enter Second
Fraction: 2/5 Sum =
9/10

Q3. Write a program to find a student's average marks. The program should ask the user to enter the number of courses he has taken and the total marks he has obtained thus far. It should then display his average to 4 decimal places.

Q4. Write a function named "swap_floats" that takes two floating point arguments and interchanges the values that are stored in those arguments. The function should return no value. To take an example, if the following code fragment is executed

```
float x = 5.8, y = 0.9;
swap_floats (x, y);
cout << x << " " << y << endl;
```

then the output will be 0.9 5.8

Q5. Write a statement (or comment) to accomplish each of the following (assume that using directives have been used for cin, cout and endl):

a) State that a program calculates the product of three integers.
b) Declare the variables x,y,z and result to be of type int (in separate statements).
c) Prompt the user to enter three integers.
d) Read three integers from the keyboard and store them in the variables x, y and z.
e) Compute the product of the three integers contained in variables x, y and z, and assign the result to the variable result.
f) Print "The product is" followed by the value of the variable result.

Q6: Write a function named "digit_name" that takes an integer argument in the range from 1 to 9, inclusive, and prints the English name for that integer on the computer screen. No newline character should be sent to the screen following the digit name. The function should not return a value. The cursor should remain on the same line

as the name that has been printed. If the argument is not in the required range, then the function should print "digit error" without the quotation marks but followed by the newline character. Thus, for example:

The statement digit_name(7); should print seven on the screen; the statement digit_name(0); should print digit error on the screen and place the cursor at the beginning of the next line.

Q7. Create the equivalent of a four-function calculator. The program should ask the user to enter a number, an operator, and another number. (Use floating point.) It should then carry out the specified arithmetical operation: adding, subtracting, multiplying, or dividing the two numbers. Use a switch statement to select the operation. Finally, display the result.

Q8. Suppose you give a dinner party for six guests, but your table seats only four. In how many ways can four of the six guests arrange themselves at the table? Any of the six guests can sit in the first chair. Any of the remaining five can sit in the second chair. Any of the remaining four can sit in the third chair, and any of the remaining three can sit in the fourth chair. (The last two will have to stand.) So the number of possible arrangements of six guests in four chairs is 6*5*4*3, which is 360. Write a program that calculates the number of possible arrangements for any number of guests and any number of chairs. (Assume there will never be fewer guests than chairs.) Don't let this get too complicated. A simple for loop should do it.

Q9. Write a C program to read an amount (integer value) and break the amount into smallest possible number of bank notes. Note: The possible banknotes are 100, 50, 20, 10, 5, 2 and 1.

Q10. Take a number from user and add 1 to each digit.

e.g. Input : 58973

Output : 69084

Q11. Sequence is defined as: 2, 6, 14, 30, 62.....

Write a program to build above sequence.

Q12. write a program to convert a number into sentence

e.g. input : 123

Output : One hundred and twenty three

Q13. Write a program in C++ to read any Month Number in integer and display the number of days for this month.

Q14. Write a program which calculate a function F where F is equal to

$F = x+y$ *if* $a>0$

$x-y$ *if* $n<=0$ *where x, y and n are inputs.*

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 02

Outline

1. Pointers
2. Pointers to array
3. Double Pointers
4. Dynamic Memory Management (new and delete operators)
5. Pointer to function OR Function Pointer
6. Pointer and multidimensional array
7. Dynamic Memory Management
8. What is a function?
9. Void function
10. The Function returns a value
11. What is recursion?
12. Direct Vs Indirect Recursion
13. Exercise

POINTERS

Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value.

REFERENCE OPERATOR (&)

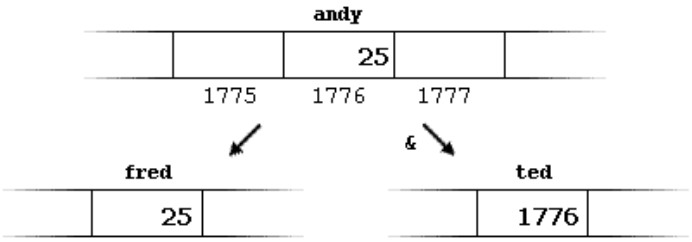
The address that locates a variable within memory is what we call a reference to that variable.

```
ted = &andy;
```

assume that andy is placed during runtime in the memory address 1776.

```
andy = 25;
fred = andy;
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:

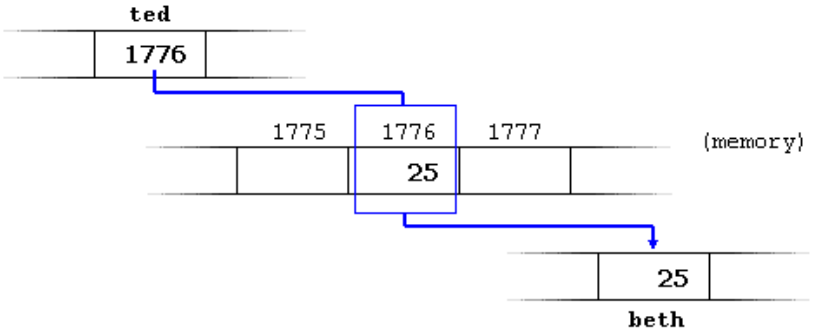


DE-REFERENCE OPERATOR (*)

The asterisk (*) acts as a dereference operator and that can be translated to "value pointed by".

beth = *ted;

We could read as: "beth equal to value pointed by ted", beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



POINTER TYPE DECLARATION

SYNTAX: type * variable ;

EXAMPLE: int *ptr, char * character, float * greatvalue;

EXPLANATION: The value of the pointer variable ***ptr*** is a memory address. A data item whose address is stored in this variable must be of the specified type.

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by
p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

POINTERS AND ARRAYS

The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to.

EXAMPLE: *Assume the following declaration:*
int numbers [20];
int * p;
The following assignment operation would be valid:
p = numbers;

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

POINTER INITIALIZATION

When declaring pointers, we can explicitly specify which variable we want them to point to.

EXAMPLE: int number;

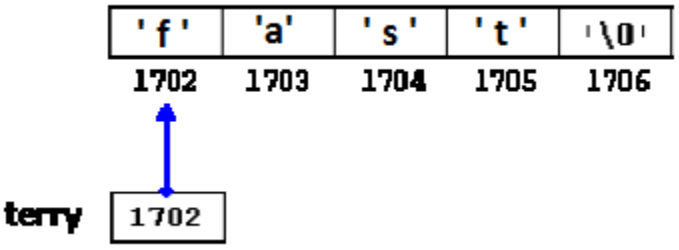
```
int *tommy = &number;
```

The above code is equivalent to:

```
int number;
int *tommy;
tommy = &number;
```

We can also initialize the content at which the pointer points with constants at the same moment the pointer is declared:

EXAMPLE: `char * terry = "fast";`
Assuming that "fast" is stored at the memory locations that start at addresses1702, we can represent the previous declaration as:



POINTER ARITHMETIC

Only addition and subtraction operations are allowed with pointers. Furthermore, when incrementing a pointer, the size in bytes of the type pointed to is added to the pointer.

EXAMPLE:

```
char *mychar;       //points to memory location 1000
short *myshort;     //points to memory location 2000
long *mylong;       //points to memory location 3000

mychar++;           //mychar now contains 1001
myshort++;          //myshort now contains 2002
mylong++;           //mylong now contains 3004
```

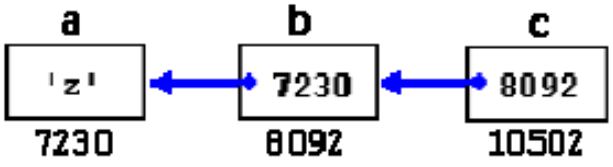
POINTERS TO POINTERS (DOUBLE POINTER)

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

EXAMPLE:

```
char a;
char * b;
char ** c;

a = 'z';
b = &a;
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could berepresented as:



- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

VOID POINTER

Void pointers are pointers that can point to any data type. One limitation of void pointers is that they cannot be directly de-referenced.

EXAMPLE:

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

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

```
y, 1603
```

NULL POINTER

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointingto any valid reference or memory address.

EXAMPLE:

```
int * p;
p = 0;          // p has a null pointer value
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to some where without a specific type.

POINTERS TO FUNCTIONS

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed de-referenced.

EXAMPLE:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int
(*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

8

DYNAMIC MEMORY ALLOCATION

MEMORY ALLOCATION

There are two ways through which memory can be allocated to a variable. They are ***static allocation*** of memory and ***dynamic memory allocation***. So far, we have declared variable and array statically. This means at compile time memory is allocated to variable or array.

Example of static memory allocation

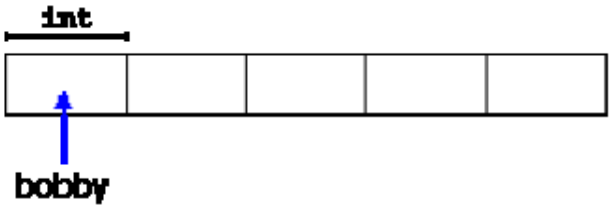
int a = 10;

Dynamic Memory Allocation examples

Dynamic memory allocation is needed if we want a variable amount of memory that can only be determined during runtime.
When we talk about dynamic memory allocation or runtime memory allocation, two keywords are important in ***C++.*** They are ***new*** and ***delete***.

- ***new and new[ ] operator***

new operator is used to dynamically allocate memory. When we allocate memory using new keyword a pointer is needed. This is so because new allocates memory and dynamically and it returns address to allocated memory and this returned memory address is stored in a pointer variable.



dynamically memory allocation for a variable

General Synatx for dynamically memory allocation for a variable
Pointer = new type;

Example
*int *ptr;* *ptr = new int;*// dynamically memory allocated for an integer and address of allocated memory is stored in ptr

Accessing value stored at the address in pointer variable (single variable)

Value at address stored in pointer variable will be accessed by dereference '*' operator as it is normally done.

Example
*int *ptr;* *ptr = new int;* **ptr = 10; // dereferencing pointer*

dynamically memory allocation for an array

General Synatx for dynamically memory allocation for an array
Pointer = new type[size];

Example
*int *ptr;* *ptr = new int[10];*// dynamically memory allocated for an integer array of size 10 and address of first element of array is stored in ptr

Accessing value stored at the address in pointer variable (array)

This can be done in two ways.

1. Use ***pointer subscript notation***. i.e ptr[0] = 10, ptr[1] = 20 etc.
   OR
2. Use ***deferencing operator.*** i.e *(ptr+0) = 10, *(ptr+1) = 20;

- ***delete and delete[ ] operator***

To free dynamic memory after it is used, **delete** operator is used so that the memory becomes available again for other requests of dynamic memory.

```
#include<iostream>
using namespace std;
int main()
{
	system("color 70");
	int i,n;
	int *p;
	cout<<"How many numbers would you like
to type? ";
	cin>>i;
	p = new int[i];
	for(n=0;n<i;n++)
	{
		cout<<"Enter Number: ";
		cin>>p[n];
	}
	cout<<endl<<"You have entered: ";
	for(n=0;n<i;n++)
	{
		cout<<p[n]<<",";

	}
	cout<<endl<<endl;

	delete[] p;

	cout<<"After deallocation, You have
entered: ";
	for(n=0;n<i;n++)
	{
		cout<<p[n]<<",";
	}
}
```

```
How many numbers would you like to type? 5
Enter Number: 1
Enter Number: 2
Enter Number: 3
Enter Number: 4
Enter Number: 5

You have entered: 1,2,3,4,5,

After deallocation, You have entered: 9198608,0,9175384,0,5,
--------------------------------
Process exited after 6.403 seconds with return value 0
Press any key to continue . . .
```

POINTER TO FUNCTION OR FUNCTION POINTER

The function pointer is actually a variable which points to the address of a function.

Function Pointers provide an extremely interesting, efficient and elegant programming technique. You can use them to replace switch/if-statements, and to realize late-binding.

Late binding refers to deciding the proper function during runtime instead of compile time.

Unfortunately, function pointers have complicated syntax and therefore are not widely used. If at all, they are addressed quite briefly and superficially in textbooks. They are less error prone than normal pointers because you will never allocate or deallocate memory with them.

Define a function pointer

Since a function pointer is nothing else than a variable, it must be defined as usual.

General Syntax

*Return_type (*name_of_pointer_variable) (data type of arguments separated by comma) = NULL;*

In the following example, we define a function pointer named function_ptr. It points to a function, which takes two integersand returns a float value.

Example – Declare function pointer

float (function_pointer) (float, float) = NULL;

Assign an address to function pointer

It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. Although it's optional for most compilers you should use the address operator & infront of the function's name in order to write portable code.

Example

```
float division(float a, float b)
{
        return a/b;
}

//function_ptr = division; //short form
//function_ptr = &division; //assignment using address operator

}
int main()
{
        float (*func_ptr)(float,float)=NULL;

        func_ptr = &division;
        float result = (*func_ptr)(9, 5);

        cout<<result <<endl;

        func_ptr = division;
        result = (*func_ptr)(9, 5);

        cout<<result <<endl;

        return 0;
}
```

Pass function pointer as an argument

You can pass a function pointer as a function's calling argument. You need this for example if you want to pass

a pointer to a callback function. The following code shows how to pass a pointer to a function which returns an int and takes a float and two char.

Example

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

int DoIt (float a, char b, char c)
{
        printf("DoIt\n");
        return a+b+c;
}
void PassPtr(int (*pt2Func)(float, char, char))
{
        int result = (*pt2Func)(12, 'a', 'b');
        // call using function pointer printf("%d", result);
}
void Pass_A_Function_Pointer()
{
        printf("Executing 'Pass_A_Function_Pointer'\n");
        PassPtr(&DoIt);
}
int main()
{
        Pass_A_Function_Pointer();
}
```

FUNCTION

A function is a group of statements that together perform a particular task. Every C++ program has at least one function and that is a main function.

Based on the nature of task, we can divide up our program into several functions.

What are local variables?

These are the variables that are declared in the function. Their lifetime ends when the execution of the function finishes and are only known in the function in which they are declared.

Function returns a value

Value returning functions are used when only one result is returned and that result is used directly in an expression.

General Format of a function return a value

```
datatype nameOfFunction()
{
        return variable;
}
```

Void Function

Void functions are used when function doesn't return a value.

General Format of a void function

```
void nameOfFunction()
{
        Statement1;
        Statement2;
        …
        Statement n;
}
```

RECURSION

When a function repeatedly calls itself, it is called a recursive function and the process is called recursion.

It seems like a never ending loop, or more formally it seems like our function will never finish. In some cases, this might true, but in practice we can check if a certain condition becomes true then return from the function.

Base Case

The case/condition in which we end our recursion is called a base case.

Example of finite recursion

```
#include<iostream>
using namespace std;
void myFunction( int counter)
{
        if(counter == 0)
                return;
        else
   {
                cout <<counter<<endl;
                myFunction(--counter);
                return;
   }
```

```
}

int main()
{
        myFunction(10);
}
```

Characteristics of Recursion

Every recursion should have the following characteristics.

1. A simple ***base case*** which we have a solution for and a return value. Sometimes there are more than one base cases.
2. A way of getting our problem closer to the base case. i.e. a way to chop out part of the problem to get a somewhat simpler problem.
3. A recursive call which passes the simpler problem back into the function.

General Format

```
returntype recursive_func ([argument list])
{
   statements;
   recursive_func ([actual argument])
}
```

Flow chart for Recursion

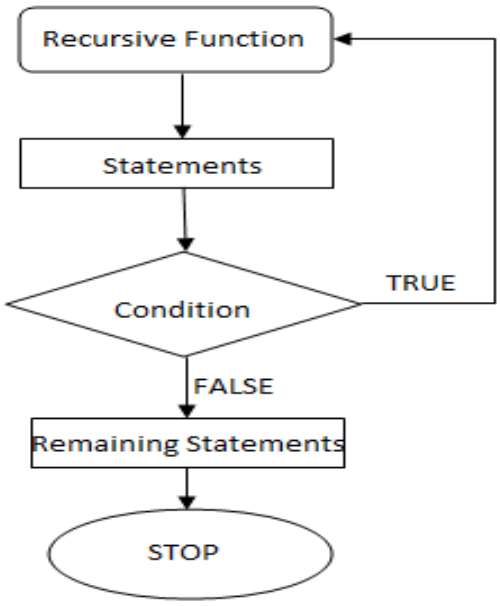


Fig: Flowchart showing recursion

DIRECT Vs INDIRECT RECURSION

There are two types of recursion, direct recursion and indirect recursion.

1. Direct Recursion

A function when it calls itself directly is known as Direct Recursion.

Example of Direct Recursion

```
#include<iostream>
using namespace std;
int factorial (int n)
{
   if (n==1 || n==0)
      return 1;
   else
      return n*factorial(n-1);
}

int main()
{
        int f = factorial(5);
        cout << f;
}
```

2. Indirect Recursion

A function is said to be indirect recursive if it calls another function and the new function calls the first calling function again.

Example of Indirect Recursion

```
#include<iostream>
using namespace std;
int func1(int);
int func2(int);
int func1(int n)
{
   if (n<=1)
      return 1;
   else
      return func2(n);
}
int func2(int n)
{
   return func1(n-1);
}
int main()
{
        int f = func1(5);
        cout << f;
}
```

Here, recursion takes place in 2 steps, unlike direct recursion.

- First, *func1* calls *func2*
- Then, *func2* calls back the first calling function *func1*.

Disadvantages of Recursion

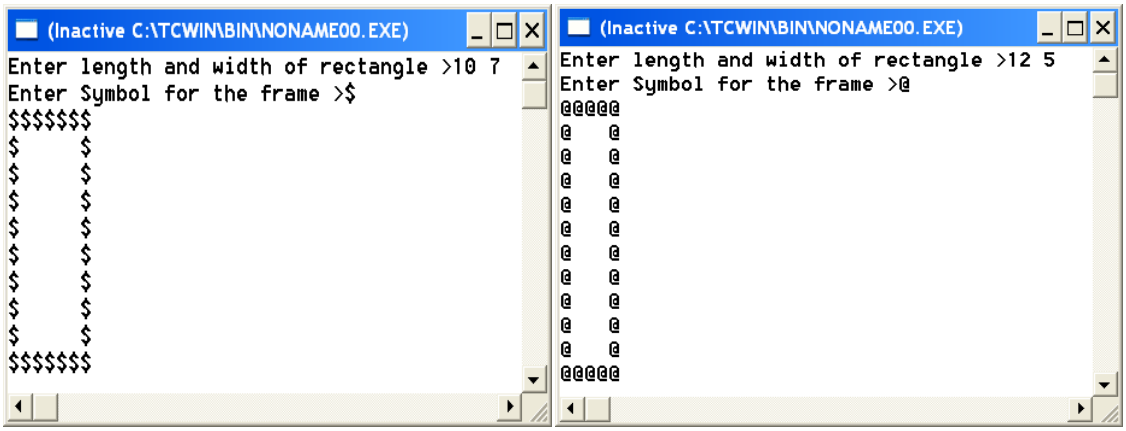
- Recursive programs are generally slower than non recursive programs. This is because, recursive function needs to store the previous function call addresses for the correct program jump to take place.
- Requires more memory to hold intermediate states. It is because, recursive program requires the allocation of a new stack frame and each state needs to be placed into the stack frame, unlike non-recursive(iterative) programs.

EXERCISE

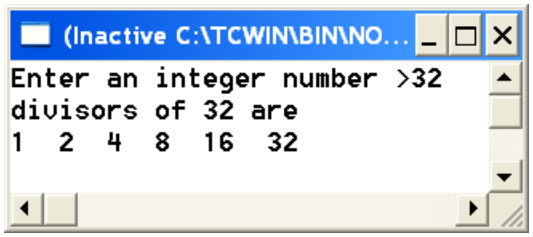
```
//draws a rectangle using functions
#include <stdio.h>
#include <iostream>
using namespace std;
void draw_solid_line(int size);
void draw_hollow_line(int size);
void draw_rectangle(int len, int wide);

int main(void) {
   int length, width;

   cout<<"Enter length and width of rectangle >";
   cin>>length;
   cin>>length>>width;

   draw_rectangle(length, width);

   return 0;
}
void draw_solid_line(int size) {
    //Write your Code Here
}
void draw_hollow_line(int size) {
    //Write your Code Here
}
void draw_rectangle(int len, int wide) {
    int i;
    draw_solid_line(wide);
    if (len > 2) {
```

```
    for (i=1; i<=len - 2; i++)
      draw_hollow_line(wide);
  }
  draw_solid_line(wide);
}
```

1. Modify above code to get following output.



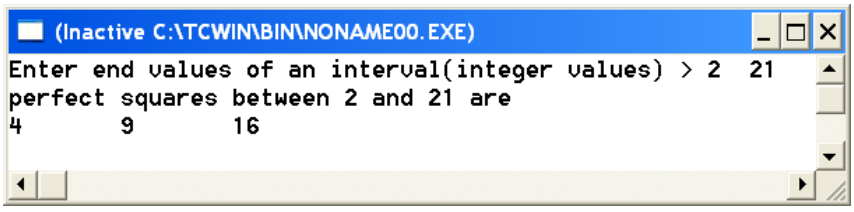
2. Write function divisors that receive an integer number and return its divisors using pointer array on main including 1 and itself.



3. Write a logical function perfect_Square that receives a positive integer number and checks if it is a perfect square or not.

Note: perfect square numbers are 4, 9,16,25,36 etc….

Write a main function that makes use of the perfect_Square function to find and print all perfect squares between n1 and n2. n1 and n2 are end values of a range introduced by the user.



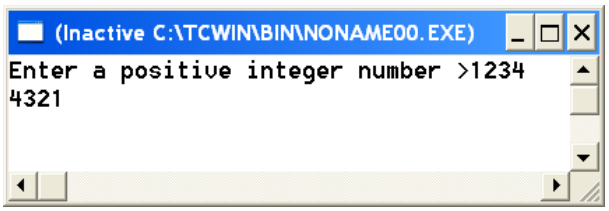
4. Write a logical function, is_prime, that takes an integer number and determines if the number is prime or not.

Note: A prime number is one that does not have proper factors.
Write a main function that makes use of the is_prime function to find and print all the prime numbers from 2 to 100.

5. Write a program that computes the area and perimeter of a rectangle using 2 functions. One function is used to read the width and length, and the other to compute the area and perimeter. Write a main function to test your functions.

6. Write a function that receives a time in seconds and returns the equivalent time in hours, minutes, and seconds. Write a main function to test your function. For example if the received time is 4000 seconds, the function returns 1 hour, 6 minutes, and 40 seconds.
Use integer division and remainder.

7. Write a recursive function that receives a positive integer number and prints it on the screen in reverse order as shown below. Write a main function for testing.



8. Write a recursive function countdigits that receives an integer number and returns how many digits it contains. Write a main function to read an integer number and tests the written function countdigits.
Input: 12345
Output: 5 Digits

9. Write a function that reverses the elements of an array. In other words, the last element must become the first; the second from last must become the second and so on. The function must accept only one pointer value and return void.

10. Write a program that will read 10 integers from the keyboard and place them in an array. The program then will sort the array into ascending and descending order and print the sorted list. The program must not change the original array or create any other integer arrays.
Hint: It requires two pointer arrays.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 03

Outline

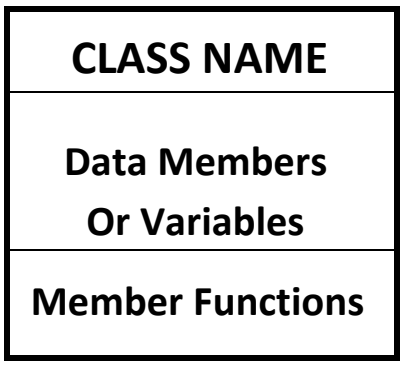
1. Classes
2. Objects
3. Structures VS Classes
4. Transformation from Procedural to Object Oriented Programming
5. Example Programs
6. Exercise

CLASSES

A class is a programmer-defined data type that describes what an object of the class will look like when it is created. It consists of a set of variables and a set of functions.

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class.

In the UML, a class icon can be subdivided into compartments. The top compartment is for the name of the class, the second is for the variables of the class, and the third is for the methods of the class.




```
class class-name
{
access-specifier:
data

access-specifier:
functions

};
```

CLASS NAME

By convention, the name of a user-defined class begins with a capital letter and, for readability, each subsequent word in the class name begins with a capital letter.

DATA MEMBERS

Consider the attributes of some real world objects:

RADIO – station setting, volume setting.
CAR – speedometer readings, amount of gas in its tank and what gear it is in.

These attributes form the data in our program. The values that these attributes take (the blue color of the petals, for example) form the state of the object.

MEMBER FUNCTIONS

Consider the operations of some real world objects:

RADIO – setting its station and volume (invoked by the person adjusting the radio's controls)
CAR – accelerating (invoked by the driver), decelerating, turning and shifting gears.

These operations form the functions in program. Member functions define the class's behaviors.

OBJECTS

In C++, when we define a variable of a class, we call it **instantiating** the class. The variable itself is called an **instance** of the class. A variable of a class type is also called an **object**. Instantiating a variable allocates memory for the object.

RADIO r;
CAR c

STRUCTURES VS. CLASSES

By default, all structure fields are public, or available to functions (like the main() function) that are outside the structure. Conversely, all class fields are private. That means they are not available for use outside the class. When you create a class, you can declare some fields to be private and some to be public. For example, in the real world, you might want your name to be public knowledge but your Social Security number, salary, or age to be private.

TRANSFORMATION FROM PROCEDURAL TO OBJECT ORIENTED PROGRAMMING

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

double calculateBMI(double w, double h)
{
 return w/(h*h)*703;
}

string findStatus(double bmi)
{
 string status;
if(bmi < 18.5)
  status = "underweight";
else if(bmi < 25.0)
  status = "normal";// so on.
return status;
}

int main()
{
    double bmi, weight, height;
    string status;
    cout<<"Enter weight in Pounds ";
    cin>>weight;
    cout<<"Enther height in Inches ";
    cin>>height;
    bmi=calculateBMI(weight,height);
    cout<<"Your BMI is "<<bmi<<" Your status is "<<findStatus(bmi);

}
```

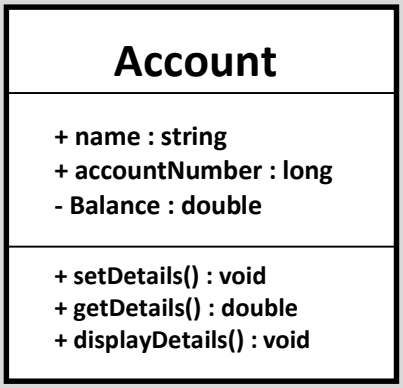
PROCEDURE ORIENTED APPROACH

```cpp
#include<iostream>
using namespace std;
class BMI
{
    double weight, height,bmi;
    string status;
    public:
        void getInput() {
            cout<<"Enter weight in Pounds ";
            cin>>weight;
            cout<<"Enther height in Inches ";
            cin>>height;   }
        double calculateBMI() {
             return weight / (height*height)*703; }
         string findStatus() {
            if(bmi < 18.5)
                status = "underweight";
            else if(bmi < 25.0)
                status = "normal";// so on.
            return status; }
        void printStatus()  {
            bmi = calculateBMI();
            cout<< "You BMI is "<< bmi<< "your status is " << findStatus(); }
};

int main()
{
    BMI bmi;
    bmi.getInput();
    bmi.printStatus();
}
```

OBJECT ORIENTED APPROACH

EXAMPLE PROGRAM

```cpp
#include<iostream>

using namespace std;

class Account
{
private:
        double balance; // Account balance

public: //Public interface:
        string name; // Account holder
         long accountNumber; // Account number
```



```
void setDetails(double bal)
{
        balance = bal;
}
double getDetails()
{
        return balance;
}
void displayDetails()
{
        cout<<"Details are: "<<endl;
        cout<<"Account Holder: "<<name<<endl;
        cout<<"Account Number: "<< accountNumber <<endl;
        cout<<"Account Balance: "<<getDetails()<<endl;
}

};

int main()
{
        double accBal;
        Account currentAccount;

        currentAccount.getDetails();

        cout<<"Please enter the details"<<endl;
        cout<<"Enter Name:"<<endl;
        getline(cin, currentAccount.name);
        cout<<"Enter Account Number:"<<endl;
        cin>>currentAccount.accountNumber;


        cout<<"Enter Account Balance:"<<endl;
        cin>>accBal;
        currentAccount.setDetails(accBal);
        cout<<endl;

        currentAccount.displayDetails();
        return 0;
}
```

Set and get functions to manipulate private data member

Publically available data:
Assigning values in main

Private data:
Can only assign values from main

```
Please enter the details
Enter Name:
Dummy
Enter Account Number:
9256432
Enter Account Balance:
25000

Details are:
Account Holder: Dummy
Account Number: 9256432
Account Balance: 25000
```

OUTPUT OF EXAMPLE PROGRAM

LAB 03 EXERCISES

INSTRUCTIONS:

NOTE: Violation of any of the following instructions may lead to the cancellation of your submission.

1) Create a folder and name it by your student id (k16-1234).
2) Paste the .cpp file for each question with the names such as Q1.cpp, Q2.cpp and so on into that folder.
3) Submit the zipped folder on slate.

QUESTION#1

Create a class Rectangle with attributes length and width, each of which defaults to 1. Provide member functions that calculate the perimeter and the area of the rectangle. Also, provide set and get functions for the length and width attributes. The set functions should verify that length and width are each floating-point numbers larger than 0.0 and less than 20.0.

QUESTION#2

Create a class called Employee that includes three pieces of information as data members—a first name (type char*), a last name (type string) and a monthly salary (type int). Your class should have a setter function that initializes the three data members. Provide a getter function for each data member. If the monthly salary is not positive, set it to 0.Write a test program that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10 percent raise and display each Employee's yearly salary again. Identify and add any other related functions to achieve the said goal.

QUESTION#3

Create a class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four data members—a part number (type string), a part description (type string), a quantity of the item being purchased (type int) and a price per item (type float). Your class should have a functions that initializes the four data members. Provide a get function for each data member. In addition, provide a member function named getInvoiceAmount that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a float value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0. Write a test program that demonstrates class Invoice's capabilities.

QUESTION#4

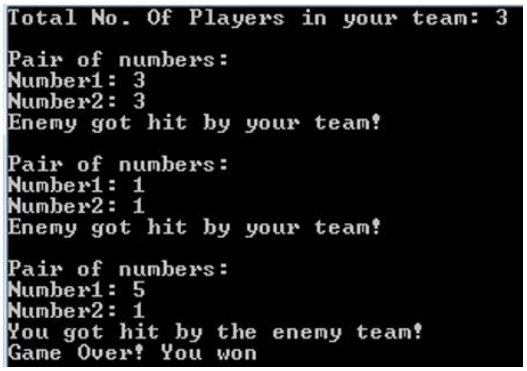
Write C++ code to represent a hitting game. The details are as follows:
This game is being played between two teams (i.e. your team and the enemy team).
The total number of players in your team is randomly generated and stored accordingly.
The function generates a pair of numbers and matches each pair. If the numbers get matched, the following message is displayed:

"Enemy got hit by your team!"

Otherwise, the following message is displayed:

"You got hit by the enemy team!"

The number of hits should be equal to the number of players in your team.
The program should tell the final result of your team by counting the hits of both the teams.
Consider the following sample output:



QUESTION#5

MyJava Coffee Outlet runs a catalog business. It sells only one type of coffee beans. The company sells the coffee in 2-lb bags only and the price of a single 2-lb bag is $5.50 when a customer places an order, the company ships the order in boxes. The boxes come in 3 sizes with 3 different costs:

	Large Box	Medium Box	Small Box
Capacity	20 Bags	10 Bags	5 Bags
Cost	$1.80	$1.00	$0.60

The order is shipped using the least number of boxes. For example, the order of 52 bags will be shipped in 2 boxes: 2 large boxes, 1medium and 1 small. Develop an application that computes the total cost of an order.

Number of Bags Ordered: 52
The Cost of Order: $ 286.00
Boxes Used:
2 Large - $3.60
1 Medium - $1.00
1 Small - $0.60
Your total cost is: $ 291.20

QUESTION#6

Write a class named Vehicle that can represent both the Rickshaw and Bike on the basis of number of wheels it has. Each vehicle has the following details

- **year**. An int that holds the vehicle's model year.
- **manufacturer**. A string that holds the manufacturer name of that vehicle.
- **speed**. An int that holds the vehicle's current speed.

In addition, the class should have the following member functions.

- **accelerate**. The accelerate function should add 5 to the speed member variable each time it is called.
- **brake**. The brake function should subtract 5 from the speed member variable each time it is called.

Demonstrate the class in a program that creates a Vehicle object for a Rickshaw and for a Bike both, and then calls the accelerate function five times. After each call to the accelerate function, get the current speed of the car and display it. Then, call the brake function two times. After each call to the brake function, get the current speed of the car and display it.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 04

Outline

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Destructor

CONSTRUCTORS

A constructor is a special function that is a member of a class.

- C++ requires a constructor call for each object that's created, which helps ensure that each object is initialized properly before it's used in a program. The constructor call occurs implicitly when the object is created. This ensures that objects will always have valid data to work on.
- Normally, constructors are declared public.
- Constructors can be overloaded, just like other functions.

DIFFERENCE BETWEEN CONSTRUCTORS AND OTHER FUNCTIONS

CONSTRUCTOR	FUNCTION
Can NOT return any value	Can return value
Constructor name is always class name	Function name can NOT be class name

DEFAULT CONSTRUCTOR

- A constructor without parameters is referred to as a default constructor.
- If a class does not contain a constructor definition, the compiler will create a minimal version of the default constructor as a public member. However, this constructor will not perform initialization. An uninitialized variable typically contains a "garbage" value.

By contrast, if a class contains at least one constructor, a default constructor must be defined explicitly.

See following example:

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

class Employee
{
 int salary;
 string name;

public:
Employee()
   {
      salary = 60000;
            name = "ABCD";
   }
void display()
{
            cout<<"name  :"<<name<<"  "<<"salary  :"<<salary<<endl;
}

};
 int main()
{
   Employee e1;
   e1.display();
    return 0;
   }
```

Parameterized Constructor:

The following example demonstrates how we can pass argument in a constructor.

Example 1:

```
#include<iostream>
#include<string>
using namespace std;
class Employee
{
 double salary;
public:
Employee(double annualSalary)
   {
      salary = annualSalary;
   }

void display()
{
                cout<<"salary  :"<<salary<<endl;
}

};
 int main()
{
   Employee e1(19876);
   e1.display();
   return 0;
   }
```

In the following example, we have used constructor with 2 arguments to initialize class object.

Example 2:

```
#include<iostream>
using namespace std;
class CoOrds
{
public:
 int x, y;
// constructor with two arguments:
public:
CoOrds(int x, int y)
   {
(*this).x = x;
(*this).y = y;
   }
 void display()
   {
cout<<"x = {0}, y = {1} : "<<x<<"  "<<y<<endl;
   }
};

int main()
{
```

```
    CoOrds p1(5, 3);
            p1.display();

}
```

MULTIPLE CONSTRUCTORS

When a class have two or more constructors, the concept is said to be is said to be overloaded. Any class member function may be overloaded, including the constructor. For example, one constructor might take an integer argument, while another constructor takes a double.

Example:

```
#include<iostream>
using namespace std;
class CoOrds
{
public:
 int x, y;
// constructor with two arguments:
public:
CoOrds(int x, int y)
   {
(*this).x = x;
(*this).y = y;
   }
   // constructor with one arguments:
CoOrds(int cor)
   {
      x = cor;
      y = cor;
   }

 void display()
   {
cout<<"x = {0}, y = {1} : "<<x<<"  "<<y<<endl;
   }
};

int main()
{           CoOrds p(15);
            p.display();
            CoOrds p1(5, 3);
            p1.display();

}
```

COPY CONSTRUCTOR:

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.

- Copy an object to return it from a function.

Syntax:

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

If a copy constructor is not defined in a class, the compiler itself defines one.

Example 1:

```
#include <iostream>

using namespace std;

class oneD {
             private:
     int i;
   public:
     int getLength(  )
      {
        return i;
       }
           // simple constructor
     oneD ( int len ){
   cout << "Normal constructor allocating i" << endl;

   i = len;
            }
             // copy constructor
     oneD (oneD &obj)
     {
   cout << "Copy constructor allocating i" << endl;

   i = obj.i; // copy the value
      }
     // destructor

~ oneD ()
  {
   cout << "Freeing memory!" << endl;

}


};

 void display(oneD obj) {
   cout << "Length of line : " << obj.getLength() <<endl;
}

int main() {

   oneD a(10);

  oneD b=a;
  display(a);// This also calls copy constructor
```

```
  display(b);

  return 0;
}
```

If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

Example 2:

```
#include <iostream>

using namespace std;

class oneD {
          private:
          int *ptr;
         public:

oneD (int len) {
  cout << "Normal constructor allocating ptr" << endl;

  // allocate memory for the pointer;
  ptr = new int;
  *ptr = len;
}


oneD (const oneD &obj) {
  cout << "Copy constructor allocating ptr." << endl;
  ptr = new int;
  *ptr = *obj.ptr; // copy the value
}

~ oneD () {
  cout << "Freeing memory!" << endl;
  delete ptr;
}

int getLength( ) {
  return *ptr;
}
};
void display(oneD obj) {
  cout << "Length of line : " << obj.getLength() <<endl;
}


int main() {
  oneD a(10);
  oneD b=a;
  display(a);
  display(b);

  return 0;
}
```

DESTRUCTOR

A destructor is a special function that is a member of a class.

- A destructor is a special member function that is called when the lifetime of an object ends.
- The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.
- Normally, destructors are declared public.
- Destructor have the same name as class name followed by a ~.

Called when an object is destroyed.

Example:

```
class Example
 {
        public:
        Example()
{
cout<< “Welcome”;
}

        ~Example()
        {

                cout<<”Good Bye”;
        }
};
```

EXERCISE

Question # 1

Write a program using class to process shopping list for a departmental store. The list includes details such as code no, price, qty and total and perform operations like adding and deleting items from the list. The program will also print the total value of an order. You are supposed to add a constructor, destructor and other necessary functions to perform the said tasks.

Question # 2

A phone number, such as (021) 38768214, can be thought of as having three parts: the area code (021) the exchange (3876) and the number (8214). Write a program that uses a class Phone to store these three parts of a phone number in specific attributes. Add a constructor that accept a number and separate these elements from that number. Write a display function that display the details of the number.
Sample Program output:

```
Please enter Your No: 02134567893
Your Area code is: 021
Your Exchange Code is: 3456
Your Consumer No is: 7893
```

QUESTION # 3

Create a class distance that stores distance in feet and inches. Add a constructor that initializes the object with default values. There must be a function that ask user to enter distance in meters and stores accordingly. Add two functions to display the distance in meters and in feet. Add a destructor that will notify the user when an object is killed.

QUESTION # 4

Create a class Sales with 3 private variables SaleID of type integer, ItemName of type string , and Quantity of type integer.
Part (a) Use a default constructor to initialize all variables with any values.
Part (b) Use a constructor to take user input in all variables to display data.
Part (c) Use a parameterized constructor to initialize the variables with values of your choice.
Part (d) Use copy constructor to copy the quantity of previously created object to current one.

QUESTION# 5

Write a program in which a class named student has member variables name, roll_no, semester and section. Use a parameterized constructor to initialize the variables with your name, roll no, semester and section. Print all data calling some public method.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 05

Outline

1. this Pointer
2. Constant Key word
3. Static Key Word
4. Examples
5. Exercise

this POINTER

- Bydefault,thecompilerprovideseachmemberfunctionofaclasswithanimplicitparameterthatpointstothe object through which the member function is called. The implicit parameter is this pointer.
- One copy of each member function in a class is stored no matter how many objects exist, and each instance of a class uses the same function code. When you call a member function, it knows which object to use because you use the object's name. The address of the correct object is stored in this pointer and automatically passed to the function.
- Within any member function, you can explicitly use this pointer to access the object's data fields. You can use the C++ pointer-to-member operator, which looks like an arrow(->).

CONSTANT DATA MEMBERS IN CLASSES

If there is a need to initialize some data members of an object when it is created and cannot be changed afterwards, use const keyword with data members.

CONSTANT MEMBER FUNCTIONS

- Constant member function is the function that cannot modify the datamembers.
- To declare a constant member function, write the const keyword after the closing parenthesis of the parameter list. If there is separate declaration and definition, then the const keyword is required in both the declaration and the definition.
- Constant member functions are used, so that accidental changes to objects can be avoided. A constant member function can be applied to a non-constobject.
- Keyword, const can't be used for constructors and destructors because the purpose of a constructor is to initialize data members, so it must change the object. Same goes fordestructors.

CONSTANTOBJECTS

As with normal variables we can also make class objects constant so that their value can't change during program execution. Constant objects can only call constant member functions. The reason is that only constant member function will make sure that it will not change value of the object. They are also called as **read only objects**. To declare constant object just write const keyword before object declaration.

STATIC VARIABLE IN FUNCTIONS

A variable declared static in a function retains its state between calls to that function.

STATIC CLASS MEMBERS

- There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A static data member is used for these and otherreasons.
- A static member variable cannot be initialized inside the class declaration. That's because the declaration is a description of how memory is to be allocated, but it doesn't allocatememory.
- Static members exist as members of the class rather than as an instance in each object of the class. So, this keyword is not available in a static memberfunction.
- A non-static member function can be called only after instantiating the class as an object. This is not the case with static member functions. A static member function can be called, even when a class is notinstantiated.
- Static functions may access only static datamembers.

THIS HOLDS THE ADDRESS OF CURRENT OBJECT

```
#include <iostream>
using namespace std;
class example
{
private:
int x; public:

/* If function argument and data member is same then use this pointer to identify the object's field */
void set(int x)
{
(*this).x = x;
}

int get()
{
return x;
}

void printAddressAndValue()
{
cout<<"The adrress is "<<this<<" and the value is "<<(*this).x<<endl;
}
};
```

```
The adrress is 0x23fe40 and the value is 5
The adrress is 0x23fe30 and the value is 6
```

CHAINED FUNCTION CALLS

```
#include<iostream>
using namespace std;
class Test
{
        private: int x; int y;
        public:
        Test(int x = 0, int y = 0)
        {
                this->x = x;
                this->y = y;
        }
        Test& setX(int a)
        {
                x = a;
```

```
                return *this;
        }
        Test& setY(int b)
        {
                y = b;
                return *this;
        }
        void print()
        {
                cout<< "x = " << x << " y = " << y <<endl;
        }
};

int main()
{
    Test obj1(5, 5);
// Chained function calls.  All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20).print();
 }
```

```
x = 10 y = 20

--------------------------------
Process exited after 0.01273 seconds with return value 0
Press any key to continue . . .
```

CONSTANT DATA MEMBERS IN CLASSES

```
#include <iostream>
using namespace std;
class Students
{
private:
string name;
const int rollno;                          // need to be initialized once member is created
float cgpa;
public:
Students(int rno):rollno(rno){}            // using memberinitializerlist
void set(string sname, float cg)
{
name = sname; cgpa = cg;
}
void print()
{
cout<<"Name: "<<name<<", Roll # "<<rollno<<", CGPA : "<<cgpa<<endl;
}
};
```

```
int main ()
{
Students s(12);
s.set("Ahmad",3.67);
s.print();
}
```

CONSTANT MEMBERFUNCTIONS

```
#include<iostream>
using namespace std;
class test
{
	private:
	int a;
	public:
		int nonconstFucntion(int a)
		{
			cout<<"Non Constant Function is called"<<endl; a=a+10;
			return a;
		}
		int constFucntion( int a) const
		{
			cout<<"Constant Function is called"<<endl;
		//	this->a=a+10;
			return a;
	}
};
main()
{
	test t;
	cout<<t.nonconstFucntion(10)<<endl;
	cout<<t.constFucntion(20);
	return 0;
}
```

CONSTANT OBJECTS

```
#include<iostream>
using namespace std;
class test
{
	public:
		int a;
		test()
		{
			a=8;
		}
		int nonconstFucntion()
		{
			cout<<"Non Constant Function is called"<<endl;//a=a+10;
		return a;
		}
```

```
		int constFucntion(int a) const
		{
			cout<<"Constant Function is called"<<endl;
			// this->a=a+10; error
			return a;
		}
};
int main()
{
const test t;
//t.a=10;//	error, can't modify const objects
//cout<<t.nonconstFucntion(); //error, can't call non const objects
cout<<t.constFucntion(10) ;
return 0;
}
```

STATIC VARIABLES IN FUNCTIONS

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

void showstat( int curr )
{
	static intnStatic;		// Value of nStatic is retained
	// between each function call
	nStatic += curr;
	cout<< "nStatic is " <<nStatic<<endl;
}

int main()
{
	for ( int i = 0; i< 5; i++ )
		showstat( i );
	return 0;
}
```

STATIC VARIABLES IN FUNCTIONS

```
#include <iostream>
using namespace std;
class Car
{
	int year;
	int mileage = 34289;
	// Warning: not-static data members initializers only available with c++11 or gnu++11 */

	static int vin = 12345678;
	// error: non-constant data member
	// only static const integral data members
	// can be initialized within a class

	static const  string model = "Sonata";  // error: not-integral type
	// cannot have in-class initializer
```

```
        static const int engine = 6;        // allowed: static const integral type public:
        static void f(int);
};

int Car::year = 2013;      // error: non-static data members
// cannot be defined out-of-class

void Car::f(int z)
{
        mileage=z;
        }
/* error: f(), a static function,
is trying to access non-static member x. */

intmain()
{
return 0;
}
```

STATIC CLASS VARIABLE

```
#include<stdio.h>
#include<iostream>
using namespace std;
class ATM
{
        public:
        static int balance;

};
intATM::balance = 250;
main()
{
        ATM ob1,ob2,ob3;
        cout<<ob1.balance++<<endl;
        cout<<ob2.balance++<<endl;
        cout<<ob3.balance;

}
```

EXERCISE

Question # 1

Create a class 'Employee' having two data members 'EmployeeName'(char*) and 'EmployeeId' (int).Keep both data members private. Create three initialized objects 'Employee1', 'Employee2' and 'Employee3' of type 'Employee' in such a way that the employee name for each employee can be changed when required but the employee Id for each employee must be initialized only once and should remain same always. Use member initializer list, accessors and mutators for appropriate data members. The result must be displayed by calling the accessors. All of the accessors must not have the ability to modify the data.

Question # 2

"Hotel Mercato" requires a system module that will help the hotel to calculate the rent of the

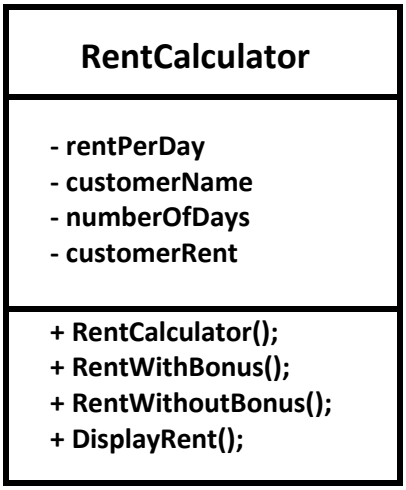
customers. You are required to develop one module of the system according to the following requirements:

1) The hotel wants such a system that should have the feature to change the implementation independently of the interface. This will help when dealing with changingrequirements.
2) The hotel charges each customer 1000.85/- per day. This amount is being decided by the hotel committee and cannot be changed fulfilling certain complexformalities.
3) The module should take the customer's name and number of days, the customer has stayed in the hotel as arguments in the constructor. The customer name must be initialized only once when the constructor iscalled. Any further attempts to change the customer's name shouldfail.
4) The module then analyses the number of days. If the customer has stayed for more than a week in the hotel , he gets discount on the rent. Otherwise, he is being chargednormally.
5) The discounted rent is being calculated after subtracting one day from the total number ofdays.
6) In the end, the module displays the followingdetails:
    a. Customer Name
    b. Days
    c. Rent

Note that, the function used for displaying purpose must not have the ability to modify any data member.

INSTRUCTIONS

- The following class structure must befollowed:



- Use appropriate data types, return types and functionarguments.
- Display the results for two initializedinstances.

REQUIRED OUTPUT

```
CustomerName: Dummy1
Days: 7
Rent: 7005.95
CustomerName: Dummy2
Days: 8
Rent: 7005.95

--------------------------------
Process exited after 0.06338 seconds with return value 0
Press any key to continue . . .
```

QUESTION # 3

Define a class to represent a **Bank account**. Include the following members.

Data members: -

1. Name of thedepositor
2. Accountnumber.
3. Type ofaccount.
4. Balance amount in theaccount.
5. Rate ofinterest

Provide a default constructor, a parameterized constructor to this class.

Also provide Member Functions: -

1. To depositamount.
2. To withdraw amount after checking for minimumbalance.
3. To display all the details of an accountholder.
4. Display rate of interest (a staticfunction)

Illustrate all the constructors as well as all the methods by defining objects.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 06

Outline

1. Inheritance
2. Types of Inheritance with Respect to Base Class Access Control
3. Examples
4. Exercise

INHERITANCE

Classes can share, obtain or "inherit" properties and methods that belong to existing classes.

Base class: It is the class from which features are to be inherited into another class.

Derived class: It is the class in which the base class features are inherited. A derived class can have additional properties and methods not present in the parent class that distinguishes it and provides additional functionality.

REAL WORLD EXAMPLE

Let's consider the Windows operating system. Windows 98 had certain properties and methods that were used in Windows XP. Windows XP was derived from Windows 98, but it was still different from it. Windows 7 was derived from Windows XP, but they were both different. They both followed a certain basic template and shared properties, however.

SYNTAX (INHERITANCE)

```
class derived-class-name : access base-class-name {
// body of class
};
```

TYPES OF INHERITANCE WITH RESPECT TO BASE CLASS ACCESS CONTROL

- Public
- Private
- Protected

PUBLIC INHERITANCE

With public inheritance, every object of a derived class is also an object of that derived class's base class. However, base class objects are not objects of their derived classes.
For example, if we have Flower as a base class and Rose as a derived class, then all Roses are Flowers, but not all Flowers are Roses—for example, a Flower could also be Jasmine or a Tulip.

IS-A RELATIONSHIP

Inheritance is represented by an is-a relationship which means that an object of a derived class also can be treated as an object of its base class for example, a Rose is a Flower, so any attributes and behaviors of a Flower are also attributes and behaviors of a Rose.

PUBLIC INHERITANCE SYNTAX

```
class Rose : public Flower
```

TABLE SHOWING BASE CLASS ACCESS CONTROL (PUBLIC PRIVATE AND PROTECTED)

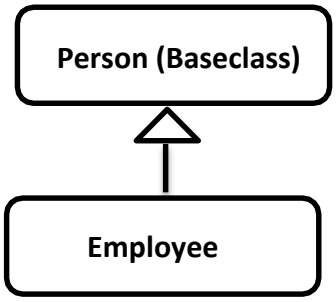
Visibility Of Base Class Members	TYPE OF		
	PUBLIC INHERITANCE	PRIVATE INHERITANCE	PROTECTED INHERITANCE
PUBLIC	PUBLIC in derived class	PRIVATE in derived class	PROTECTED in derived class
PRIVATE	HIDDEN in derived class	HIDDEN in derived class	HIDDEN in derived class
PROTECTED	PROTECTED in derived class	PRIVATE in derived class	PROTECTED in derived class

TYPES OF INHERITANCE WITH RESPECT TO DERIVED CLASSES

- SINGLE INHERITANCE
- HIERARICAL
- MULTILEVEL
- MULTIPLE INHERITANCE
- HYBRID INHERITANCE

1) SINGLE INHERITANCE

It is the type of inheritance in which there is one base class and one derived class.



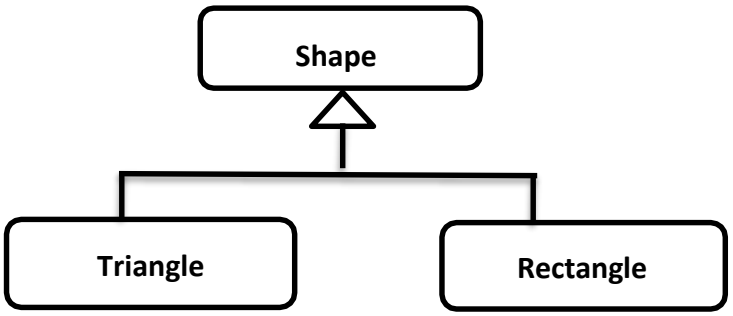
SINGLE INHERITANCE

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

class Person
{
        char name[100],gender[10];
        int age;
        public:
        void getdata()
        {
                cout<<"Name: ";
                cin>>name;
                cout<<"Age: ";
                cin>>age;
                cout<<"Gender: ";
                cin>>gender;
        }

        void display()
        {
                cout<<"Name: "<<name<<endl;
                cout<<"Age: "<<age<<endl;
                cout<<"Gender: "<<gender<<endl;
        }
};
```

```
class Employee: public Person
{
        char company[100];
        float salary;
        public:
        void getdata()
        {
                Person::getdata();
                cout<<"Name of Company: ";
                cin>>company;
                cout<<" Salary: Rs.";
                cin>>salary;
        }
void display()
        {
                Person::display();
        cout<<"Name of Company:"<<company<<endl;
                cout<<"Salary: Rs."<<salary<<endl;
        }
};

int main()
{
        Employee emp;
        cout<<"Enter data"<<endl;
        emp.getdata();
        cout<<endl<<"Displaying data"<<endl;
        emp.display();
        return 0;
}
```

2) HIERARICAL INHERITANCE

This is the type of inheritance in which there are multiple classes derived from one base class. This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.

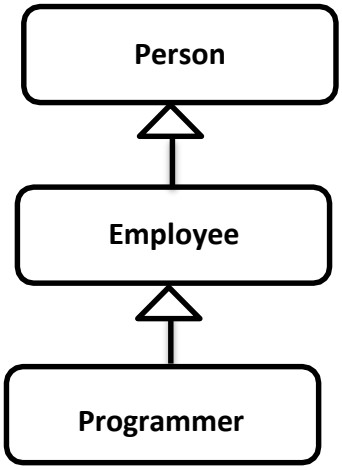


HIERARICAL INHERITANCE

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

class Shape
{
        protected:
        float width, height;
        public:
        void set_data (float a, float b)
        {
                width = a;
                height = b;
        }
};
class Rectangle: public Shape
{
        public:
        float area ()
        {
                return (width * height);
        }
};

class Triangle: public Shape
{
        public:
        float area ()
        {
                return (width * height / 2);
        }
};
int main ()
{
        Rectangle rect;
        Triangle tri;
        rect.set_data (5,3);
        tri.set_data (2,5);
        cout << rect.area() << endl;
        cout << tri.area() << endl;
        return 0;
}
```

3) MULTI LEVEL INHERITANCE

When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.

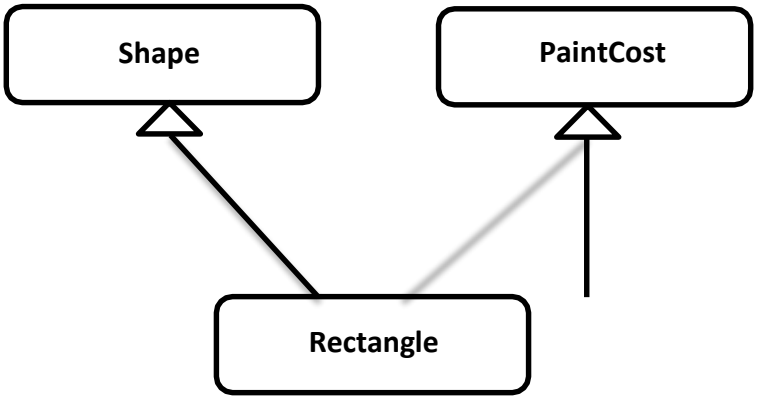


MULTI LEVEL IHERITANCE

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

class Person
{
        char name[100],gender[10];
        int age;
        public:
        void getdata()
        {
                cout<<"Name: ";
                cin>>name;
                cout<<"Age: ";
                cin>>age;
                cout<<"Gender: ";
                cin>>gender;
        }

        void display()
        {
                cout<<"Name: "<<name<<endl;
                cout<<"Age: "<<age<<endl;
                cout<<"Gender: "<<gender<<endl;
        }
};

class Employee: public Person
{
        char company[100];
        float salary;
        public:
        void getdata()
        {
                Person::getdata();
                cout<<"Name of Company: ";
                cin>>company;
                cout<<" Salary: Rs.";
                cin>>salary;
        }
void display()
        {
                Person::display();
                cout<<"Name of Company:"<<company<<endl;
                cout<<"Salary: Rs."<<salary<<endl;
        }
};

 class Programmer: public Employee
{
        int number;
        public:
        void getdata()
        {
                Employee::getdata();
                cout<<"Number of programming language known: ";
                cin>>number;
        }
        void display()
        {
                Employee::display();
               cout<<"Number of programming language known:"<<number;
        }
};

int main()
{
        Programmer p;
        cout<<"Enter data"<<endl;
        p.getdata();
        cout<<endl<<"Displaying data"<<endl;
        p.display();
        return 0;
}
```

4) MULTIPLE INHERITANCE

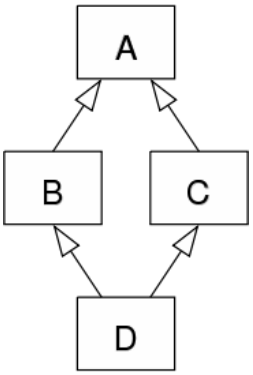
In multiple inheritance, one class inherits the features of multiple classes simultaneously, hence this type of inheritance is called Multiple Inheritance.



MULTIPLE INNHERITANCE	

#include <iostream>
using namespace std;
 	
class Shape // Base class Shape
{
public:
void setWidth(int w)
{
width = w;
}
void setHeight(int h)
{
height = h;
}
protected:
int width;
int height;
};
 	
class PaintCost // Base class PaintCost
{
public:
int getCost(int area)
{
return area * 70;
}
};
	class Rectangle: public Shape, public PaintCost // Derived class
	{
	public:
	int getArea()
	{
	return (width * height);
	}
	};
	
	int main()
	{
	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;
	}

5) HYBRID INHERITANCE

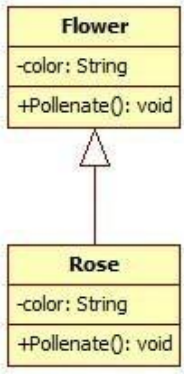


HYBRID INHERITANCE

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


class Student
{
        protected:
        int rno;
        public:
        void get_no(int a)
        {
                rno=a;
        }
        void put_no(void)
        {
                cout<<"Roll no"<<rno<<"\n";

        }
};

class Test:public Student
{
        protected:
        float part1,part2;
        public:
        void get_mark(float x,float y)
        {
                part1=x;
                part2=y;
        }
        void put_marks()
        {
cout<<"Marksobtained:\npart1="<<part1<<"\n"<<"part2="<<part2
<<"\n";

        }
};
```

```
class Sports
{

        protected:
        float score;
        public:
        void getscore(float s)
        {
                score=s;
        }
        void putscore()
        {
                cout<<"sports:"<<score<<"\n";
        }
};



class Result: public Test, public Sports
{
        float total;
        public:
        void display()
        {
                total=part1+part2+score;
                put_no();
                put_marks();
                putscore();
                cout<<"Total Score="<<total<<"\n";
        }
};


int main()
{
        Result stu;
        stu.get_no(123);
        stu.get_mark(27.5,33.0);
        stu.getscore(6.0);
        stu.display();
        return 0;
}
```

UML REPRESENTATION FOR INHERITANCE (GENERALIZATION)




```
class Flower
{
        string color;
        public:
        void Pollenate();
};
class Rose : public Flower
{ };
```

CONSTRUCTOR CALLS

The constructor of a derived class is required to create an object of the derived class type. As the derived class contains all the members of the base class, the base sub-object must also be created and initialized. The base class constructor is called to perform this task. Unless otherwise defined, this will be the default constructor.
The order in which the constructors are called is important. The base class constructor is called first, then the derived class constructor. The object is thus constructed from its core outwards.

BASE CLASS INITIALIZER

C++ requires that a derived-class constructor call its base-class constructor to initialize the base-class data members that are inherited into the derived class. In a derive class, constructor uses a base class initialize list which uses a member initializer to pass arguments to the base-class. When a derived class constructor calls a base-class constructor, the arguments passed to the base-class constructor must be consistent with the number and types of parameters specified in one of the base-class constructors; otherwise, a compilation error occurs.

BASE CLASS INITIALIZER WITH SIINGLE INHERITANCE

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

using namespace std;

class Account
{
        private:
        long accountNumber;             // Account number

        protected:
        string name;                    // Account holder

        public:                         //Public interface:
        const string accountType;       // Account Type

Account(long accNumber, string accHolder, const string& accType)
: accountNumber(accNumber), name(accHolder), accountType(accType)
{
        cout<<"Account's constructor has been called"<<endl<<endl;
}

~ Account()                             //Destructor
{
```

```
        cout<<endl<<"Object Destroyed";
}

const long getAccNumber() const   //accessor for privately defined data menber; accountNumber
{
        return accountNumber;
}

void DisplayDetails()
{
        cout<<"Account Holder: "<<name<<endl;
        cout<<"Account Number: "<<accountNumber<<endl;
        cout<<"Account Type: "<<accountType<<endl;
}
};

class CurrentAccount : public  Account  //Single Inheritance
{
        private:
        double balance;
        public:

        CurrentAccount(long accNumber, const string& accHolder, string accountType, double accBalance)
        : Account(accNumber, accHolder, accountType), balance(accBalance)
        {
                cout<<"CurrentAccount's constructor has been called"<<endl<<endl;
        }


void deposit_currbal()
   {
     float deposit;
     cout<<"Enter amount to Deposit : ";
     cin>>deposit;
     cout<<endl;
     balance = balance + deposit;
   }

void Display()
{
        name = "Dummy"; //can change protected data member of Base class
        DisplayDetails();
        cout<<"Account Balance: "<<balance<<endl<<endl;
}
};

int main()
{
        CurrentAccount currAcc(7654321,"Dummy1", "Current Account", 1000);
        currAcc.deposit_currbal();
        currAcc.Display();

        return 0;
}
```

BASE CLASS INITIALIZER WITH MULTIPLE INHERITANCE

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

class FirstBase
{
        protected:
        int a;
        public:
        FirstBase(int x)
        {
cout<<"Constructor of FirstBase is called: "<<endl;
                a=x;
        }
};

class SecondBase
{
        protected:
        string b;
        public:
        SecondBase(string x)
        {
cout<<"Constructor of SecondBase is called: "<<endl;
                b=x;
        } };
```

```
class Derived : public FirstBase, public SecondBase
{
        public:
        Derived(int a,string b):
FirstBase(a),SecondBase(b)
        {
        cout<<"Child Constructor is called: "<<endl;
        }

        void display()
        {
                cout<<a<<" "<<b<<endl;
        }
};

int main()
{
        Derived obj(24,"Multiple Inheritance");
        obj.display();
}
```

DESTROYING OBJECTS

When an object is destroyed, the destructor of the derived class is first called, followed by the destructor of the base class. The reverse order of the constructor calls applies. You need to define a destructor for a derived class if actions performed by the constructor need to be reversed. The base class destructor need not be called explicitly as it is executed implicitly.

ADVANTAGES OF INHERITANCE

Reusability – Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

Saves Time and Effort - The above concept of reusability achieved by inheritance saves the programmer's time and effort. Since the main code written can be reused in various situations as needed. **Maintainability** - It is easy to debug a program when divided in parts.

LAB 06 EXERCISES

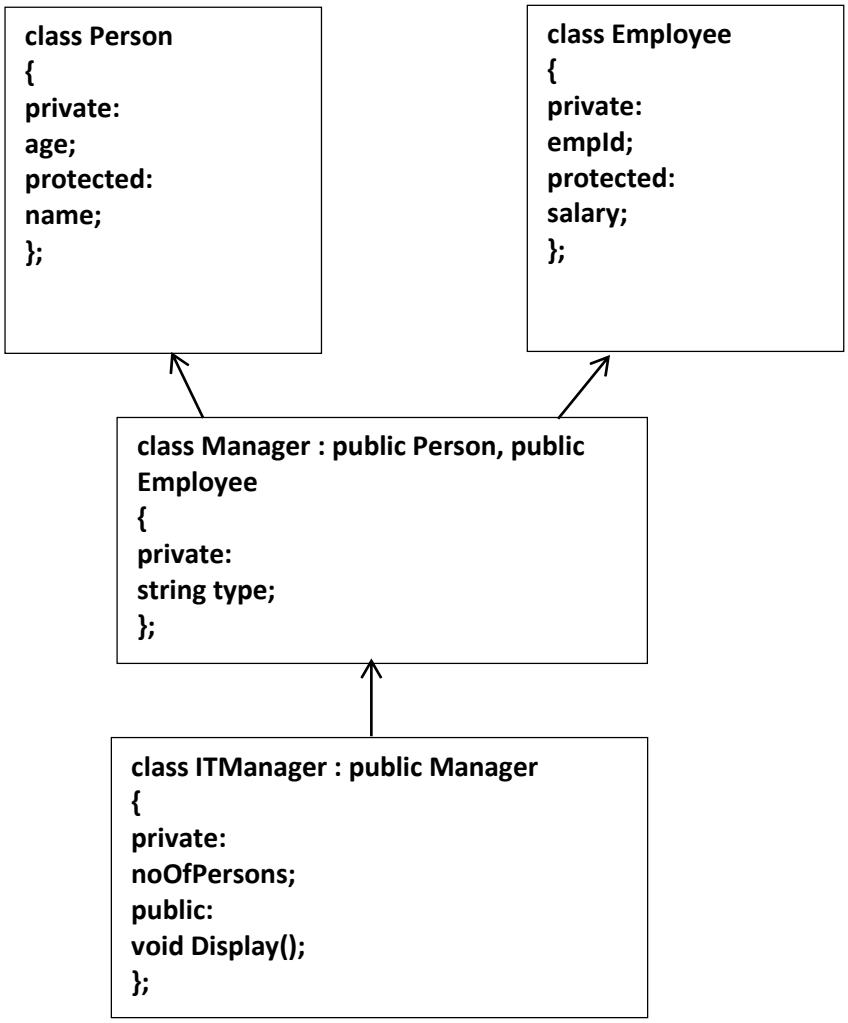
INSTRUCTIONS:

NOTE: Violation of any of the following instructions may lead to the cancellation of your submission.

1) Create a folder and name it by your student id (k16-1234).
2) Paste the .cpp file for each question with the names such as Q1.cpp, Q2.cpp and so on into that folder.
3) Submit the zipped folder on slate.

QUESTION#1

Implement the following scenario in C++:



1) The Display() function in "ITManager" should be capable of displaying the values of all the data members declared in the scenario (age,name,empId,salary,type,noOfPersons) without being able to alter the values.

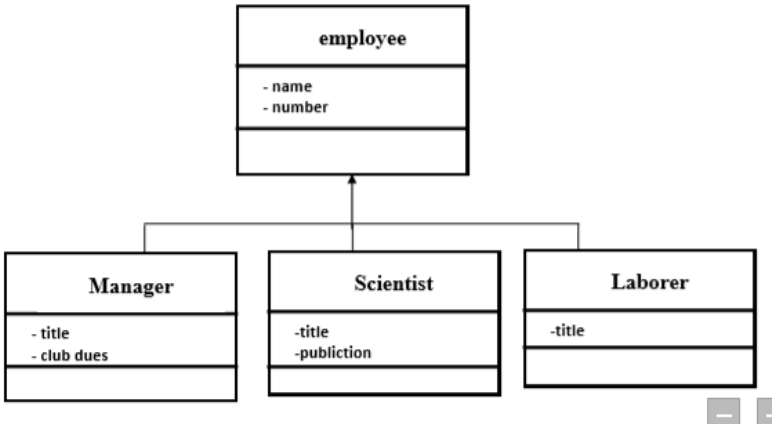
2) The "int main()" function should contain only three program statements which are as follows:

a) In the first statement, create object of "ITManager" and pass the values for all the data members:
ITManager obj(age,name,empId,salary,type,noOfPersons);
b) In the second statement, call the Display() function.
c) In the third statement, return 0.

Question#2

In the database only three kinds of employees are represented. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses.

The database stores a name and an employee identification number for all employees, no matter what their category is. However, for managers, it also stores their titles and golf club dues. For scientists, it stores the number of scholarly articles they have published and their title. Laborers store the title only.
You must start with a base class employee. This class handles the employee's last name and employee number. From this class three other classes are derived: manager, scientist, and laborer. All three classes contain additional information about these categories of employee, and member functions to handle this information as shown in figure.



Question#3

A supermarket chain has asked you to develop an automatic checkout system. All products are identifiable by means of a barcode and the product name. Groceries are either sold in packages or by weight. Packed goods have fixed prices. The price of groceries sold by weight is calculated by multiplying the weight by the current price per kilo.

Develop the classes needed to represent the products first and organize them hierarchically. The Product class, which contains generic information on all products (barcode, name, etc.), can be used as a base class.

The Product class contains two data members for storing barcodes and the product name. Define a constructor with parameters for both data members. Add default values for the parameters to provide a default constructor for the class. In addition to the access methods setCode() and getCode(), also define the methods scanner() and printer(). For test purposes, these methods will simply output product data on screen or read the data of a product from the keyboard.

The next step involves developing special cases of the Product class. Define two classes derived from Product, PrepackedFood and Fresh- Food. In addition to the product data, the PrepackedFood class should contain the unit price and the FreshFood class should contain a weight and a price per kilo as data members. In both classes define a constructor with parameters providing default-values for all data members. Use both the base and member initializer. Define the access methods needed for the new data members.Also redefine the methods scanner() and printer() to take the new data members into consideration.

Test the various classes in a main function that creates three objects each of the types Product, PrepackedFood and FreshFood. One object of each type is fully initialized in the object definition. Use the default constructor to create the other object and test the scanner() method. For the third object, test the get and set methods. Display the products on screen for all objects.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

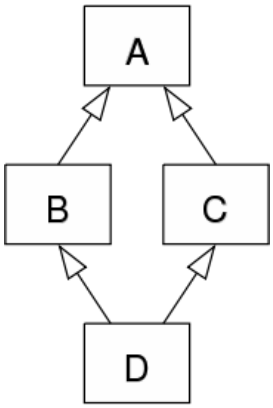
Lab # 07

Outline

1. Diamond problem in Hybrid Inheritance
2. Polymorphism
3. Polymorphism Using Function overloading and Function Overriding
4. Examples
5. Exercise

Diamond Problem

In case of hybrid inheritance, a Diamond problem may arise. The "dreaded diamond" refers to a class structure in which a particular class appears more than once in a class's inheritance hierarchy.



For Example:

```
class Animal
{
public:
voidwalk()
{
cout<<"animal walks";
}

};
```

```
class Donkey : public Animal
{

};
```

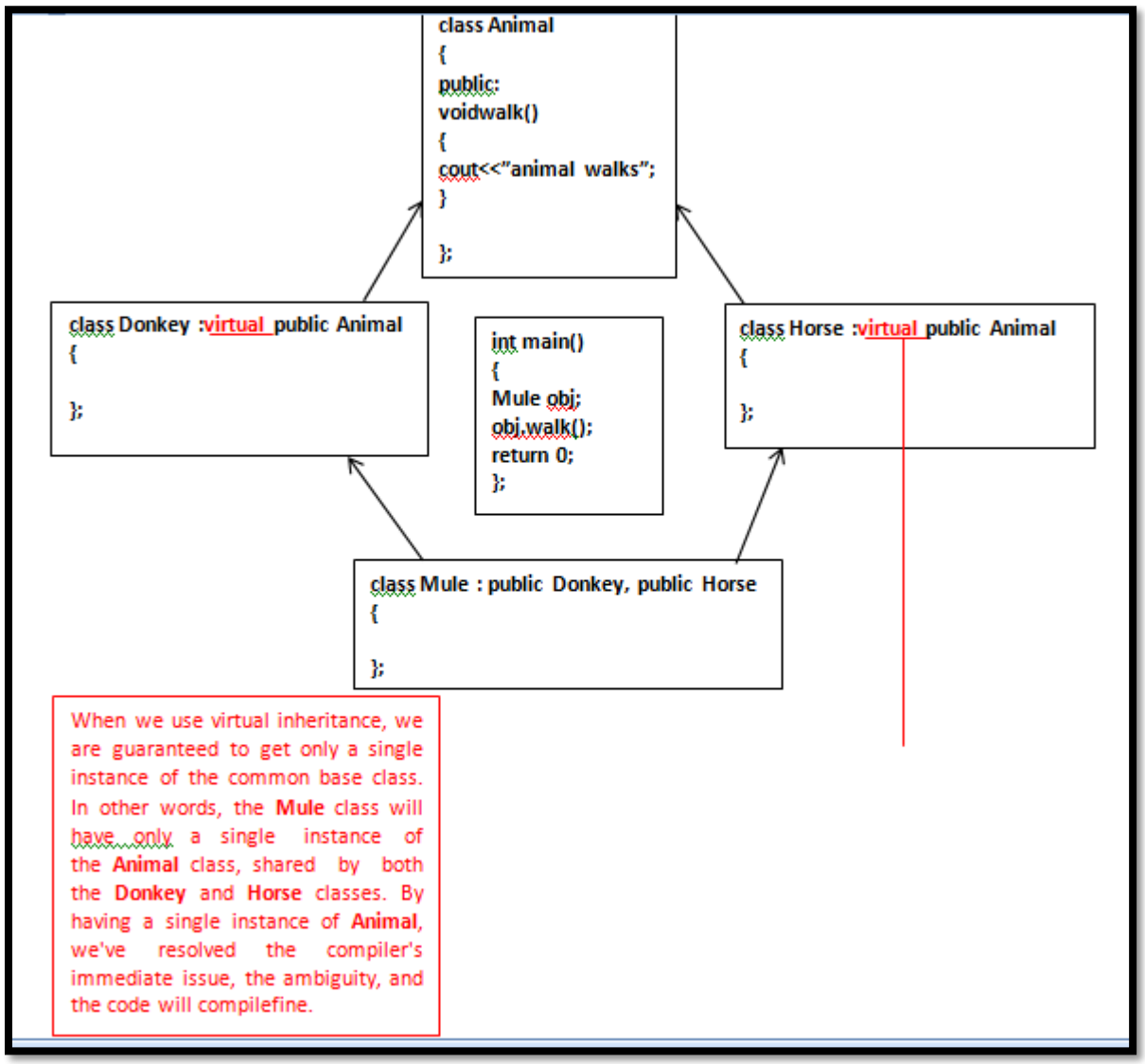
```
int main()
{
Mule obj;
obj.walk();
return 0;
};
```

```
class Horse : public Animal
{

};
```

```
class Mule : public Donkey, public Horse
{

};
```

Call is ambiguous; the compiler can't know which implementation of **walk()** to use, the one from the **Donkey** class or the one from the **Horse** class.

How to solve this Problem??? Virtual Base Class Inheritance



Example#01:

Solution of Diamond Problem Using Virtual Inheritance

```
#include <iostream>

class LivingThing {

public:

   void breathe() {

      std::cout << "I'm breathing as a living thing." <<
std::endl;

   }

};

class Animal : virtual public  LivingThing {

public:

   void breathe() {

      std::cout << "I'm breathing as an animal." << std::endl;

   }

};

class Reptile :virtual public  LivingThing {

public:

void crawl() {

      std::cout << "I'm crawling as a reptile." << std::endl;

   }

};

class Snake :public Animal,public Reptile {

};
```

```
int main() {

Snake snake;

snake.breathe();

   snake.crawl();

 return 0;

 }
```

Example#02:

Parametrized Constructor Calling

```
#include<iostream>

using namespace std;

class Person {

public:

   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }

   Person()    { cout << "Person::Person() called" << endl;   }

};

class Faculty : virtual public Person {

public:

   Faculty(int x):Person(x)   {

      cout<<"Faculty::Faculty(int ) called"<< endl;

   }};

class Student : virtual public Person {

public:

   Student(int x):Person(x) {

      cout<<"Student::Student(int ) called"<< endl;

   }

};

class TA : public Faculty, public Student  {

public:

   TA(int x):Student(x), Faculty(x), Person(x)   {

      cout<<"TA::TA(int ) called"<< endl;

   }

};

int main()  {

   TA t(30);

 }
```

Polymorphism

Polymorphism refers to the ability of a method to be used in different ways, that is, it can take different forms at different times (poly + morphos).

TYPES OF POLYMORPHISM:

There are two types of polymorphism:

- Compile timepolymorphism
- Run timepolymorphism.

COMPILE TIME POLYMORPHISM:

In C++ you can achieve compile time polymorphism by,

- Constructor Overloading (have discussed in the previous labs)
- Function/Method Overloading
- Operator Overloading

Function Overloading

Function Overloading Example

```
#include<iostream>
using namespace std;
class subtraction
{
public:
void difference(int a,int b)
  {
 cout<<a-b<<endl;
}
void difference(int a,int b,int c)
{
cout<<a-b-c<<endl;
}
void difference(double a,double b)
  {
cout<<a-b<<endl;
}

 void difference(int a,double b)
  {
cout<<a-b<<endl;
}

 void difference (double a,int b)
   {
 cout<<a-b<<endl;
}

};
```

```
int main()
{ subtraction obj;
obj.difference(67,34);
obj.difference(4.5,2.3);
obj.difference(12,2,5);
obj.difference(9.4,4);
obj.difference(3,1.4);
return 0;
}
```

Function OverRiding

If we inherit a class into a Derived class and provide definition of base Class function again inside a derived class,then that function said to be overridden and this mechanism is called function overriding.

Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

Requirements For Function Overriding:

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Function Overriding Simple Example

```
#include<iostream>

using namespace std;

class Person {

int id;

public:

   Person(int x)  { id=x;}

   Person()    { cout << "Person::Person() called" << endl;   }

void show()

{cout<<"person's show calling"<<endl;}


};

class Faculty : virtual public Person {int empid;

public:

   Faculty(int x,int y):Person(x)   {

     empid=y;    }};

class Student : virtual public Person {int std_id;

public:

   Student(int x,int y):Person(x) {std_id=y;

         }

void show()

{cout<<"student's show calling"<<endl;}


};
```

```
class Admin : public Faculty, public Student  {

public:

   Admin(int x,int y,int z):Student(x,y), Faculty(x,z), Person(x)   {

         }

void show()

{/*call the Overridden function from overriding function*/

Student::show();

cout<<"admin's show calling"<<endl;}

};

int main()  {

   Admin t(30,40,50);

t.show();

/* call overridden function from the child class*/

t.Person::show();

 }
```

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.

Function call Using Objects

```
#include<iostream>
using namespace std;
class Base
{
 public:
 void display()
 {
  cout<<"Base class"<<endl;
 }
};
class Derived:public Base
{
 public:
 void display()
 {
  cout<<"Derived Class"<<endl;
 }
};
```

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

Function Call Binding With base Class Pointer

Function call Using Objects

```
#include<iostream>
using namespace std;
class Base
{
 public:
 void display()
 {
  cout<<"Base class"<<endl;
 }
};
class Derived:public Base
{
 public:
 void display()
 {
  cout<<"Derived Class"<<endl;}};
```

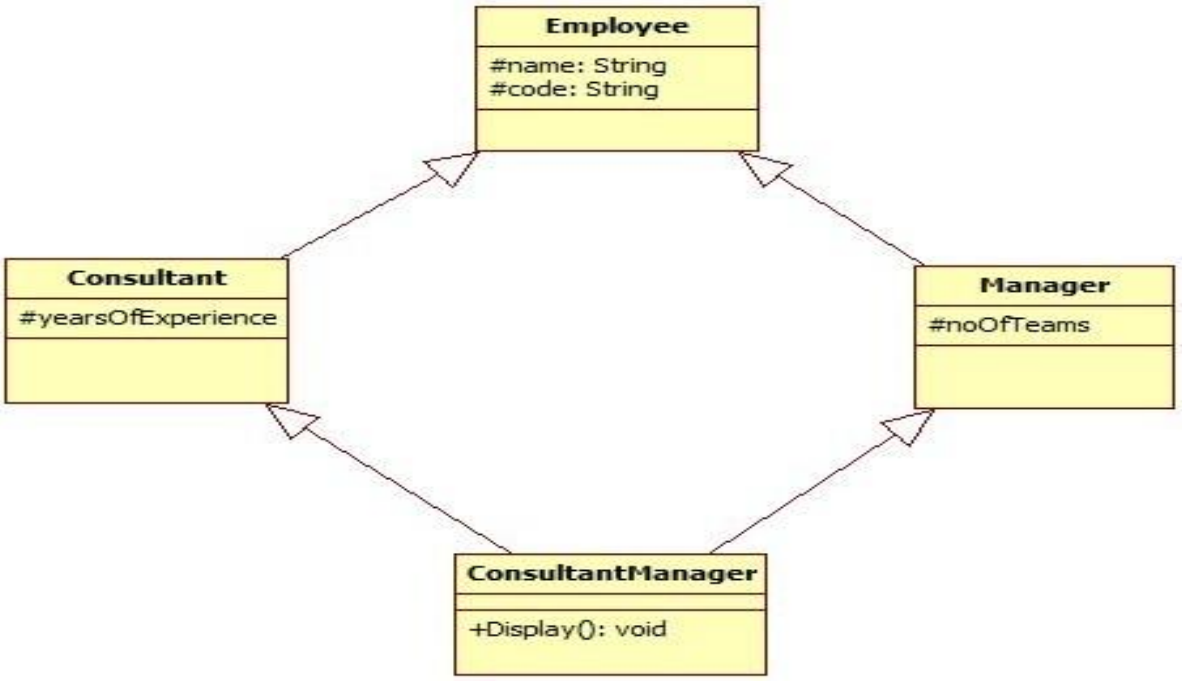
```
   int main()
  {
   Base *b;      //Base class pointer
   Derived d;  //Derived class object
  b=&d;
  b->display();  //early Binding Occurs


}
```

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 **display()** function, without knowing the actual object type.

EXERCISES

QUESTION#1

Implement the following scenario in C++:



1. No accessors and mutators are allowed to be used.
2. The Display() function in "ConsultantManager" should be capable of displaying the values of all the data members declared in the scenario (name,code,yearsOfExperience,noOfTeams) without being able to alter thevalues.

3. The "int main()" function should contain only three program statements which are asfollows:

   a) In the first statement, create object of "ConsultantManager" and pass the values for all the data members:

   ConsultantManagerobj("Ali","S-123",17,5);

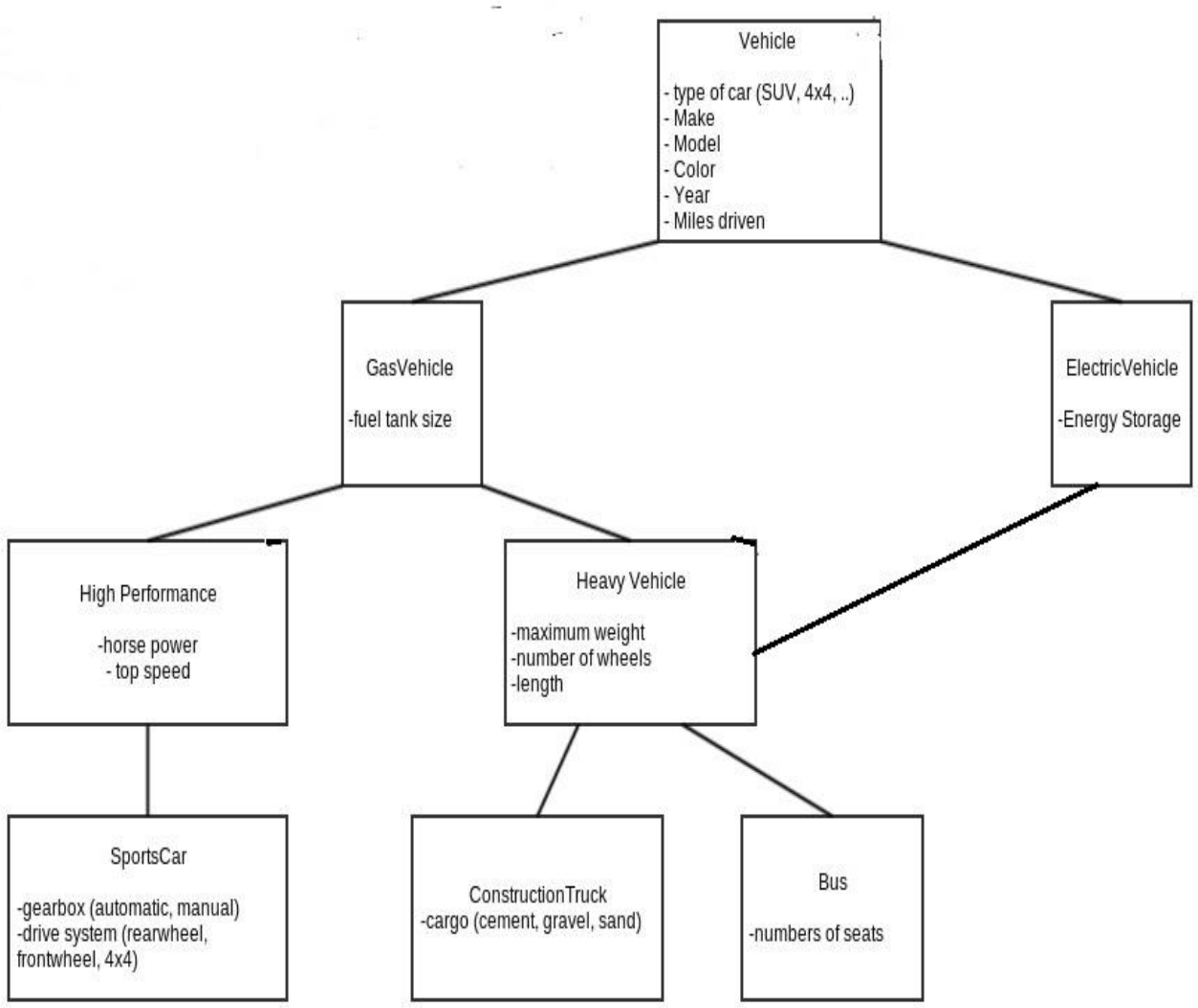
   b) In the second statement, call the Display()function.

   c) In the third statement, return0.

All the values are required to be set through constructors parameter.

QUESTION#2

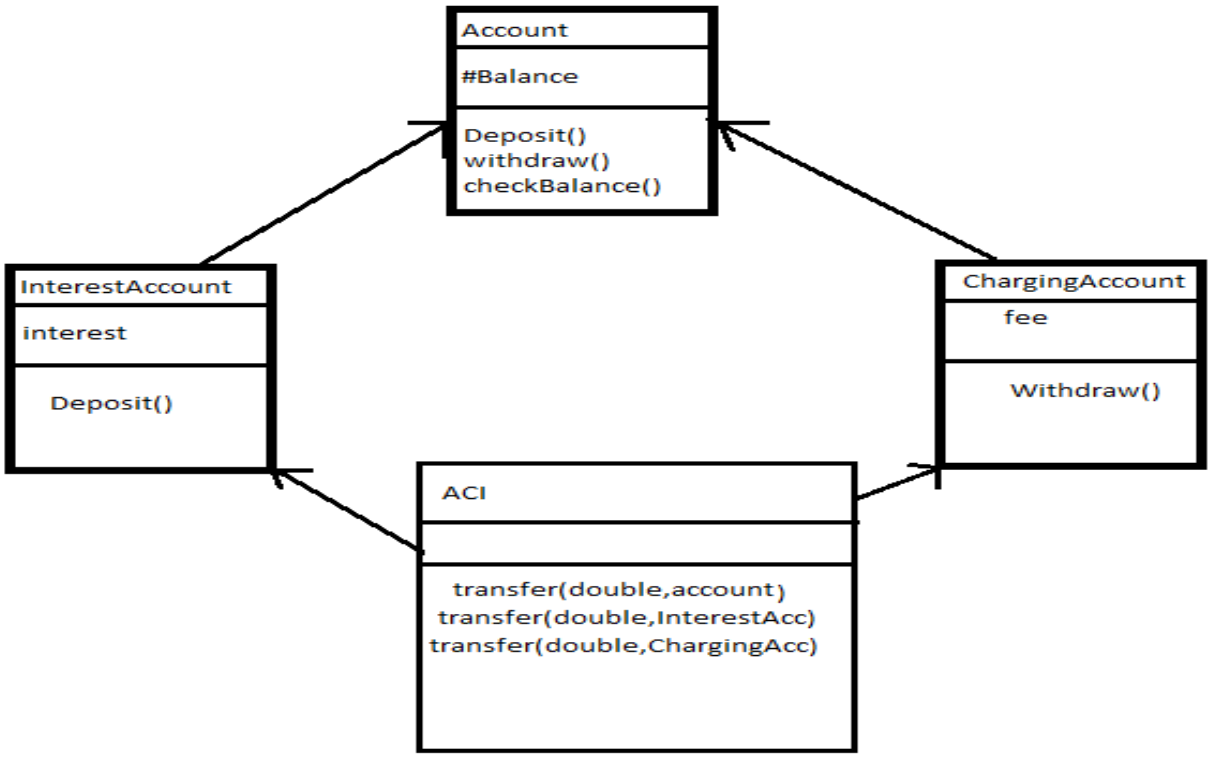
Implement the following scenario in C++:



1. All the values are required to be set through constructors parameter.
2. Provide necessary accessor functions where required.
3. Create an object of class bus by initializing it through parametrized constructor in the main function and display all data members by calling display function of class bus.

QUESTION#3

Implement the following scenario in C++:



1. The interestaccount class adds interest for every deposit,assume a default of 30%.
2. The charging account class charges a default fee of $3 for every withdrawl.
3. Transfer method of aci class takes two parameters,amount to be transfer and object of class in which we have to transfer that amount.
4. Make parametrized constructor, and default constructor to take user input for all data members.
5. Make a driver program to test all functionalities.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 08

Outline

1. Polymorphism
2. Operator Overloading
3. Operator overloading as member functions
4. Operator overloading as non-functions
5. Examples
6. Exercise

POLYMORPHISM

Polymorphism refers to the ability of a method to be used in different ways, that is, it can take different forms at different times (poly + morphos).

TYPES OF POLYMORPHISM

There are two types of polymorphism:

- Compile timepolymorphism
- Run timepolymorphism.

COMPILE TIME POLYMORPHISM

Compile time (static) polymorphism occurs when a method is overloaded.
EXAMPLE: An example of this would be suggesting different names for being the President of a country, which would get you different results each time – but they would still be called thePresident.

TYPES OF OVERLOADING:

- Constructor Overloading – Already discussed in previouslabs.
- Function Overloading - Already discussed in previouslabs.
- Operator Overloading - Is discussedbelow.

OPERATOR OVERLOADING

An operator is said to be overloaded if it is defined for multiple types. In other words, overloading an operator means making the operator significant for a newtype.

BUILT IN OVERLOADS

Most operators are already overloaded for fundamental types.

Example:

1) In the case of theexpression:

   a / b

   the operand type determines the machine code created by the compiler for the division operator. If both operands are integral types, an integral division is performed; in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

2) <<, which is used both as the stream insertion operator and as the bitwise left-shiftoperator.

OVERLOADS FOR USER DEFIND TYPES

Operators can be used with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

Example:

The effect of + operator can be stipulated for the objects of a particular class.

OPERATOR FUNCTION SYNTAX

To overload an operator, an appropriate *operator function is required.*

```
returntype operator op (arg_list)
{
   function body // task defined
}
```

- **returntype**is the type of value returned by the specifiedoperation.
- **op** is the operator being overloaded. (+,-etc)
- **op** is preceded by the keyword**operator**.

LIST OF OPERATORS THAT CAN BE OVERLOADED IN C++

```
new     delete      new[]       delete[]
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=     <<=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
()      []
```

LIST OF OPERATORS THAT CAN'T BE OVERLOADED

- **?:**(conditional)
- **.** (memberselection)
- **.***(member selection withpointer-to-member)
- **::**(scoperesolution)
- **sizeof** (object sizeinformation)
- **typeid** (object type information)

OPERATOR OVERLOADING AS MEMBER FUNCTIONS

If the operator function of a binary operator is defined as a method inside the class, the left operand must always be an object of the class. The operator function is called for this object. The second, right operand is passed as an argument to the method. The method thus has a single parameter.

EXAMPLE

```
//Rupee.h
#include<sstream>       // The class
stringstream#include<iomanip>
#include <iostream>
using namespace std;

class Rupee
{
        private:
                long data;
        public:
                Rupee( int rupee = 0)
                        {
                                data = rupee;
                        }
```

```
        Rupeeoperator-()const                   // Negation (unaryminus)
                {
                        Rupee temp;
                        temp.data = -data;
                        return temp;
                }

        Rupee operator+( const Rupee&obj)const      // addition.
                {
                        Rupee temp;
                        temp.data = data + obj.data;
                        return temp;
                }

        Rupee operator-( const Rupee&obj)const      // Subtraction.
                {
                        Rupee temp;
                        temp.data = data - obj.data;
                        return temp;
                }

        Rupee& operator+=( constRupee&obj)          // AddRupees.
                {
                        data += obj.data;
                        return *this;
                }

        Rupee& operator-=( constRupee&obj)          // SubtractRupees.
                {
                        data -= obj.data;
                        return *this;
                }

        friend ostream&operator<<( ostream&os, const Rupee &e );
};

        ostream& operator<<(ostream&os, const Rupee& e) //Overloading << operator
                {
                        os<<e.data;
                        return os;
                }
```

```
//TestRupee.cpp
#include "Rupee.h"
#include <iostream>
using namespace std;

int main()
{
        Rupee wholesale(20), retail;
        retail=wholesale;               // Standard assignment

        cout<< "Wholesale price:"<<wholesale;
```

```
        cout<< "\nRetail price: "<<retail;

        Rupee discount(2);
        retail -= discount;
        cout<< "\nRetail price including discount: "<<retail;

        wholesale = 34.10;
        cout<< "\nNew wholes

        retail = wholesale + 10;
        cout<< "\nNew retail p rice: "<<retail;

        Rupee profit( retail - w olesale);
        cout<< "\nThe profit: " <<profit;

        profit = -profit;
        cout<< "\nThe profit after unary minus: "<<profit;

        return 0;
}
```

This Statement Means:

retail = wholesale.operator+(Rupee(10)); The binary operator is always called with reference to the left hand argument and here, it must be a class object because the operator function has been defined as a class method. So, the operator function doesn't handle the following situation:

retail = 10 + wholesale;

However, if we want to convert both operands, we will need global definitions for the operator functions.

```
Wholesale price: 20
Retail price: 20
Retail price including discount: 18
New wholesale price: 34
New retail price: 44
The profit: 10
The profit after unary minus: -10
--------------------------------
Process exited after 0.03745 seconds with return value 0
Press any key to continue . . .
```

OPERATOR OVERLOADING AS NON-FUNCTIONS

GENERAL SYNTAX:

```
TYPE1 operator OP(TYPE2lhs, TYPE3rhs)
{

}
```

EXAMPLE:

```
//Rupee.h
//Globally Defined
Rupee operator+( const Rupee& e1, const Rupee& e2) // addition.
                        {
                                Rupee temp(e1);
                                temp += e2;
                                return temp;
                        }
```

ISSUE:

A global function cannot access the private members of the class i.e. data. The function operator+() shown above therefore uses the += operator, whose operator function is defined as a public method.
A global operator function can be declared as a "**friend**" of the class to allow it access to the private members of thatclass.

EXAMPLE:

```
//Inside class Rupee
friend Rupee operator+( const Rupee& e1, const Rupee& e2);


//Globally Defined
Rupee operator+( const Rupee& e1, const Rupee& e2) // addition.
                        {
                                Rupee temp; temp.data = e1.data + e2.data;
                                return temp;
                        }
```

LAB 08 EXERCISES

QUESTION#1

You are required to develop a location based System as per the following requirements:

1) Create a class (Location), that takes two inputs latitude (integer) and longitude (integer). The values for both inputs are set through the constructor.
2) The "Location" class is capable of displaying both the values without being able to alter them through Display().
3) Create another class (Details) which extends the functionality of Location class by displaying another location related attribute i.e. address through the function Display().The function being unable to alter the values. Set the value of address through constructor.
4) The "int main()" function should be coded as follows:

    a) Create one initialized instance (details) of "Details" class and three initialized instances (obj1, obj2 and obj3) of Location class having the following values (10,20),(5,30) and (90,90) respectively.
    b) Display the contents of each instance.
    c) Perform the pre increment operation on obj1 and display the result.
    d) Perform the post increment operation on obj1 , assign it to obj2 and display the result forobj2.
    e) Add "10" to obj1 (keeping "10" the operand on right hand side), assign the result to obj2 and display the result forobj2.
    f) Add "10" to obj1 (keeping "10" the operand on left hand side), assign the result to obj2 and display the result forobj2.
    g) Assign the contents of obj3 to obj1 and obj2 in a single statement and display the results for all three instances.
    h) Reference the instance of "Details" class by a pointer to the "Location" class. Use this pointer to display the address.
    i) Return0.

QUESTION#2

Define a class named PrimeNumber that stores a prime number. The default constructor should set the prime number to 1. Add another constructor that allows the caller to set the prime number. Also, add a function to get the prime number. Finally, overload the prefix and postfix ++ and -- operators so they return a PrimeNumber object that is the next largest prime number (for ++) and the next smallest prime number (for --). For example, if the object's prime number is set to 13, then invoking ++ should return a PrimeNumber object whose prime number is set to 17. Create an appropriate test program for the class.

QUESTION#3

Write a class Time which represents time. the class should have three fields for hours, minutes and seconds. It should have constructor to initialize the hours, minutes and seconds. A function print Time() to print the current time. Overload the following operators:

- Plus operator (+) to add two time objects based on 24-hour clock.
- Operator< to compare two time objects.

QUESTION#4

Complete the following tasks:

a. Design a PhoneCall class that holds a phone number to which a call is placed, the length of the call in minutes, and the rate charged pe rminute.

b. Overload the == operator to compare two PhoneCalls. Consider one PhoneCall to be equal to another if both calls are placed to the same number.

c. Create a main() function that allows you to enter 10 PhoneCalls into an array. If a PhoneCall has already

been placed to a number, do not allow a second PhoneCall to the same number. Save the file as PhoneCall.cpp.

QUESTION#5

Design a class called NumDays. The class's purpose is to store a value that represents a number of work hours and convert it to a number of days. For example, 8 hours would be converted to 1 day, 12 hours would be converted to 1.5 days, and 18 hours would be converted to 2.25 days. The class should have a constructor that accepts a number of hours, as well as member functions for storing and retrieving the hours and days. The class should also have the following overloaded operators:

- The addition operator +. The number of hours in the sum of two objects is the sum of the number of hours in the individual objects.
- The subtraction operator -. The number of hours in the difference of two objects X and Y is the number of hours in X minus the number of hours in Y.
- Prefix and postfix Increment operators ++. The number of hours in an object is incremented by 1.

Prefix and postfix decrement operators --. The number of hours in an object is decremented by1.

QUESTION#6

Complete the following tasks:

a. Design a Meal class with two fields—one that holds the name of the entrée, the other that holds a calorie count integer. Include a constructor that sets a Meal's fields with parameters, or uses default values when no parameters are provided.

b. Include an overloaded insertion operator function that displays a Meal's values.

c. Include an overloaded extraction operator that prompts a user for an entrée name and calorie count for a meal.

d. Include an overloaded operator+()function that allows you to add two or more Meal objects. Adding two Meal objects means adding their calorie values and creating a summary Meal object in which you store "Daily Total" in the entrée field.

e. Write a main()function that declares four Meal objects named breakfast, lunch, dinner, and total. Provide values for the breakfast, lunch, and dinner objects. Include the statement total = breakfast + lunch + dinner; in your program, then display values for the four Meal objects. Save the file asMeal.cpp.

f. Write a main()function that declares an array of 21 Meal objects. Allow a user to enter values for 21 Meals for the week. Total these meals and display the calorie total for the end of the week. (Hint: You might find it useful to create a constructor for the Meal class.) Save the file asMeal2.cpp.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 09

Outline

1. Friend Function
2. Casting
3. Runtime Polymorphism
4. Pure Virtual Functions
5. Abstract Class
6. Examples
7. Exercise

Friend Function

Friend Functions:

The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.

Why they are needed?

They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example, suppose we wrote the code to compute a complex mathematical formula in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

Are friend functions against the concept of Object Oriented Programming?

It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

Consider the following class: class X

```
{
private:
int a, b;
public:
void MemberFunction ();
  …
}
```

Suppose we have a global function DoSomething that need to access the private members of class X, when we will try to access them compiler will generate error as outside world can not access private members of a class except its member functions.

```
void DoSomething(X obj)
{
        obj.a = 3; //Error
        obj.b = 4; //Error
}
```

In order to access the member variables of the class, we must make function friend of that class, **class X**

```
{
 private:
        int a, b;
public:
…
friend void DoSomething(X obj);
 }
```

Now the function DoSomething can access data members of **class X**

```
void DoSomething(X obj)
{
        obj.a = 3;
        obj.b = 4;
}
```

Prototypes of friend functions appear in the class definition. But friend functions are NOT member functions.

Friend functions can be placed anywhere in the class without any effect Access specifiers don't affect friend functions or Classes.

```
class X{
 ...
private:
friend void DoSomething(X);
public:
friend void DoAnything(X);
...
};
```

While the definition of the friend function is:

```
void DoSomething(X obj)
{       obj.a = 3; // No Error obj.b =
        4; // No Error
         …
 }
```

friend keyword is not given in definition.

Friend Classes:

Similarly, one class can also be made friend of another class: class X
{
friend class Y;
…
};

Member functions of class Y can access private data members of class X **class X**

```
 {
          friend class Y;
private:
          int x_var1, x_var2;
...
}; class Y
{
private:
          int y_var1, y_var2; X objX;
public: void setX()
{         objX.x_var1 = 1;
 }
};

 int main()
{
    Y objY; objY.setX();
return 0;
}
```

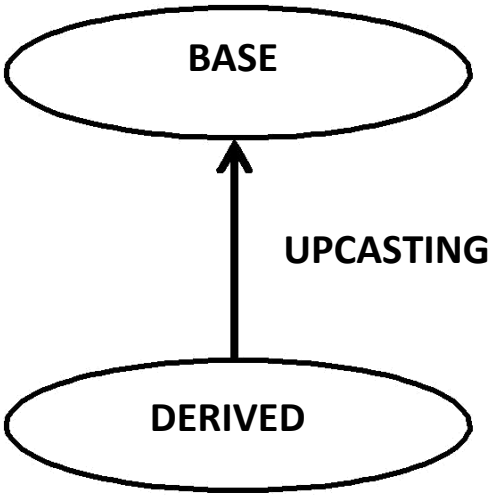
CASTING

A cast is a special operator that forces one type to be converted into another.

UPCASTING

Upcasting is a process of treating a pointer or a reference of derived class object as a base class pointer.

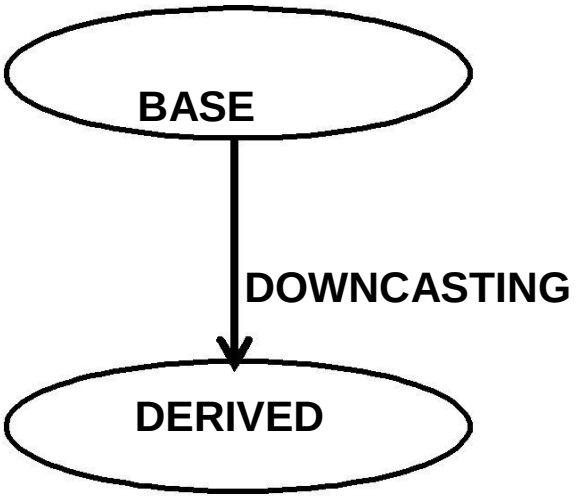
- A base class pointer can only access the public interface of the base class.
- The additional members defined in the derived class are therefore inaccessible.
- Upcasting is not needed manually. We just need to assign derived class pointer (or reference) to base class pointer.



DOWNCASTING

The opposite of Upcasting is Downcasting, It converts base class pointer to derived class pointer.

- Type conversions that involve walking down the tree, or *downcasts*, can only be performed explicitly by means of a cast construction. The cast operator **(type)**or the **static_cast<>**operator are available for this task, and are equivalent in thiscase.
- After downcasting a pointer or a reference, the entire public interface of the derived class isaccessible.



STATIC_CAST

Syntax: static_cast<type>(expression)

The operator static_cast<> converts the expression to the target type type.

```
#include    <iostream>
using  namespace  std;
class Employee
{
public:
        Employee(string fName, string lName, double sal)
        {
                FirstName = fName;
                LastName = lName;
                salary = sal;
        }
        string FirstName;
        string LastName;
        double salary;
        void show()
        {
                 cout<< "First Name: " <<FirstName<< " Last Name: " <<LastName<< " Salary: " << salary<<endl;
        }
        void addBonus(double bonus)
        {
                salary += bonus;
        }
};

class Manager :public Employee
{
public:
        Manager(string fName, string lName, double sal, double comm) :Employee(fName, lName, sal)
        {
                Commision = comm;
        }
        double Commision;
        double getComm()
        {
                return Commision;
        }
};
```

FOR UPCASTING

```
        int main()
{
        Employee* emp;          //pointer to base class object

        Manager m1("Ali", "Khan", 5000, 0.2); //object of derived class

        emp = &m1; //implicit upcasting

        emp->show(); //okay because show() is a base class function
        return 0;}
```

FOR DOWNCASTING USING (type)

```
int main()
{
	Employee e1("Ali", "Khan", 5000); //object of base class

	//try to cast an employee to Manager
	Manager* m3 = (Manager*)(&e1); //explicit downcasting

	cout<< m3->getComm() <<endl; return 0;
}
```

FOR DOWNCASTING USING (static_cast)

```
int main()
{
	Employee e1("Ali", "Khan", 5000); //object of base class

	//try to cast an employee to Manager
	Manager* m3 = static_cast<Manager*>(&e1); //explicit downcasting

	cout<< m3->getComm() <<endl; return 0;
}
```

DOWNCASTING IS UNSAFE

-Since, e1 object is not an object of Manager class so, it does not contain any information about commission.

-That's why such an operation can produce unexpected results.

-Downcasting is only safe when the object referenced by the base class pointer really is a derived class type.

-To allow safe downcasting C++ introduces the concept of *dynamic casting.*

VIRTUAL FUNCTION

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform Late Binding on this function.

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

class Shape
{
        public:
         virtual void Draw()
        {
                cout<<"Shape drawn!"<<endl;
        }
};

class Square : public Shape
{
        public: void
        Draw()
        {
                cout<<"Square drawn!"<<endl;
        }
};

int main()
{
        Squareobj;    //Derived ClassObject

        Shape* shape = &obj; /* derived class object being referenced by a
                                pointer to the base class */

        shape->Draw(); /* outputs "Square drawn!" if Draw() is virtual in base
                          class else outputs "Shape drawn!"*/
        return 0;
}
```

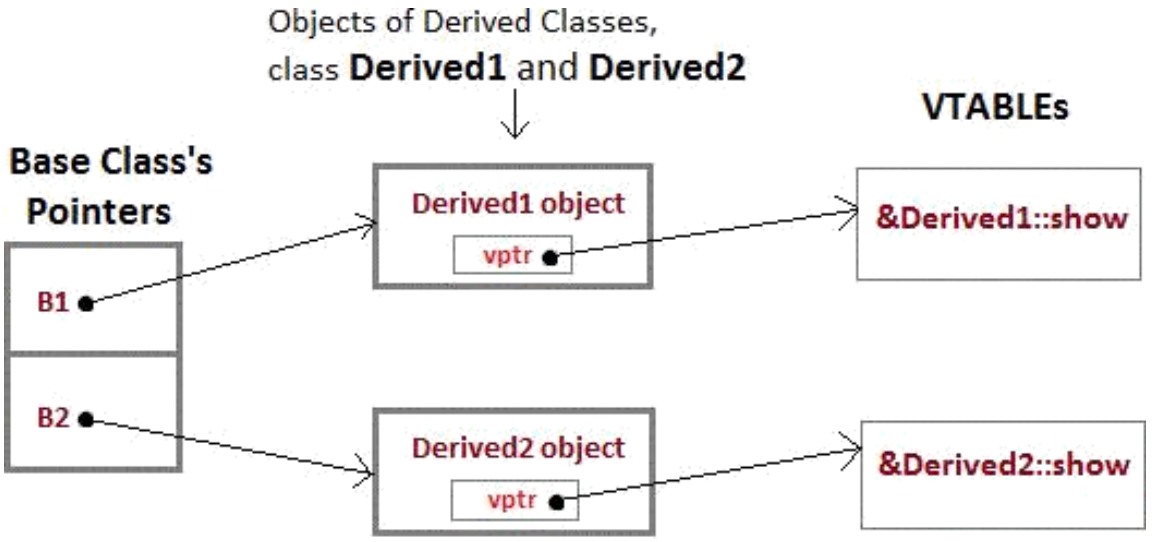
INTERESTING FACTS

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

MECHANISM OF LATE BINDING

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.

- Only the Base class Method's declaration needs the Virtual Keyword, not thedefinition.
- If a function is declared as virtual in the base class, it will be virtual in all its derivedclasses.
- The address of the virtual Function is placed in the VTABLE and the compiler uses VPTR(vpointer)to point to the VirtualFunction.



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

Overriding Methods – virtual keyword

- Methods in the parent class can be redefined in the child class
- In C++, static binding is the defaults behaviour
- The keyword virtual allows the use of dynamic binding.

```
//Header.h

//Polygon.h
#ifndef __POLYGON_H_INCLUDED__  // if polygon.h hasn't been included yet...
#define __POLYGON_H_INCLUDED__  //  #define this so the compiler knows it has been included
class Polygon {
            protected:
              int width, height;
            public:
              void set_values (int a, int b);
              virtual int area ();


          };
#endif

          //Rectangle.h
          #include "Polygon.h"
          class Rectangle: public
            Polygon { public:
              int area();

          };
//Triangle.h
#include "Polygon.h"
class Triangle: public
  Polygon { public:
    int area ();

};
//Impl
//Polygon.cpp
#include "Polygon.h"
```

```
void Polygon::set_values (int a, int b)
  {
             width=a;
              height=b;
            }
int Polygon::area()
            {
                        return 0;
            }

//Rectangle.cpp
#include "Rectangle.h"

int Rectangle::area()
             {
             return width*height;
           }
//Triangle.cpp
#include "Triangle.h"

int Triangle::area()
           {
           return (width * height / 2);
}
```

//Main.cpp

```
#include <iostream>
#include "Rectangle.h"
#include "Triangle.h"
using namespace std;

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
   Polygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
 cout<< ppoly1->area() << '\n';
 cout<< ppoly2->area() << '\n';
 cout<< ppoly3->area() << '\n';

  return 0;
}
```

Pure Virtual Functions

virtual Type identifier(parameters) = 0;

// Pure Virtual Functions

virtual void price() = 0;

virtual void type() = 0;

Abstract Base Class

- An abstract base class is a class that includes or inherits at least one pure virtual function that has not been defined.

```
#include <string>
#include<iostream>
using namespace std;
//Header files
//Animal.h
class Animal
{
protected:
   string m_strName;

public:
Animal(string strName); string GetName();
   virtual string Speak() = 0; // pure virtual function
};
//Cow.h
class Cow: public Animal
{
public:
Cow(string strName);
          virtual string Speak();

};
//Impl
Cow:: Cow(string strName): Animal(strName)
   {
   }
string Cow::Speak() { return "Meow";
 }
Animal::Animal(string strName): m_strName(strName)
   {
   }
string Animal::GetName()
 {
 return m_strName;
 }
int main()
{
   Cow cCow("Betsy");
   Cout<<cCow.GetName()<<”says”<<cCow.Speak();
   }
```

LAB-09 EXERCISES

QUESTION#1

Design and implement a program that shows the relationship between person, student and professor. Your person class must contain two pure virtual functions named getData() of type void and isOutstanding() of type bool and as well getName() and putName() that will read and print the person name. Class student must consist of function name getData (), which reads the GPA of specific person and isOutstanding() function which returns true if the person GPA is greater than 3.5 else should return false. Class professor should take the respective persons publications in getData() and will return true in Outstanding() if publications are greater than 100 else will return false . Your main function should ask the user either you want to insert the data in professor or student until and unless user so no to add moredata.

QUESTION#2

A company pays its employees weekly. The employees are of four types: **Salariedemployees** are paid a fixed weekly salary regardless of the number of hours worked, **hourlyemployees** are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, **commissionemployees** are paid a percentage of their sales and **base-salaryplus-commissionemployees** receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants you to draw UML diagram of the given scenario and do implementation on c++ that performs its payroll calculations polymorphically.

Your Employee class must have:

- First name, last name and social security number as private data members. Use accesor and mutators to set and get these values.
- Constructor with first name, last name and ssn number as parameter.
- Pure virtual function named earning with return type double.
- Virtual function named print with return type void, which employee first name, last name and ssn number.

Your salaried Employee must have:

- Earning method which return the salary
- Print method that print the employee detail and employee salary.

Your hourly Employee must have:

- You must take the wage and hours as a parameter using base class initializer.
- Earning methods which returns the salary the employed has worked.
- Print method that print the employee detail and employee salary.

Your commission employee must have:

- You must take the commission rate and gross sale rate as a parameter using base class initializer.
- Earning methods which returns the commission of the employee.
- Print method that print the employee detail and employee commission.

Your base-salary plus-commission employees must have:

- You must take the base salary as a parameter using base class initializer.
- Earning methods which return the base commission of the employee.
- Print method that print the employee detail and employee base salary.

You have to perform upcasting and downcasting, and print the details of each employee type.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 10

Outline

1. C++ Filing
2. Examples
3. Exercise

C++ Filing

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax –

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows –

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen.

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output –

```
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

End of File

Returns true if the eofbit *error state flag* is set for the stream.
This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream.
Note that the value returned by this function depends on the last operation performed on the stream (and not on the next).
Operations that attempt to read at the *End-of-File* fail, and thus both the eofbit and the failbit end up set. This function can be used to check whether the failure is due to reaching the *End-of-File* or to some other reason.

Return Value
true if the stream's eofbit error state flag is set (which signals that the End-of-File has been reached by the last input operation).
false otherwise.

```
#include <iostream>     // std::cout
#include <fstream>      // std::ifstream

int main () {

  std::ifstream is("example.txt");   // open file

  char c;
  while (is.get(c))                  // loop getting single characters
    std::cout << c;

  if (is.eof())                     // check for EOF
    std::cout << "[EoF reached]\n";
  else
    std::cout << "[error reading]\n";

  is.close();                       // close file

  return 0;
}
```

LAB 10 EXERCISES

INSTRUCTIONS:

NOTE: Violation of any of the following instructions may lead to the cancellation of your submission.

1) Create a folder and name it by your student id (k17-1234).
2) Paste the .cpp file for each question with the names such as Q1.cpp, Q2.cpp and so on into that folder.
3) Submit the zipped folder on slate.

QUESTION#1

Write a program to implement I/O operations on characters. I/O operations includes inputting a string, calculating length of the string, Storing the String in a file and fetch the stored characters from it.

QUESTION#2

Write a program to copy the contents of one file to another.

QUESTION#3

Take a class Person having two attributes name and age.

- Include a parametrized constructor to give values to all data members.
- In main function
    i. Create an instance of the person class and name it person1.
    ii. Create a binary file person.bin and write person1 object into it.
    iii. Read the person1 object from the file.
    iv. Return 0

QUESTION#4

Take a class Participant having three attributes (ID, name and score) and following member functions

- Input () function takes data of the object and stores it in a file name participant.dat
- Output () function takes id from user and show respective data of that id.
- Max () gives the highest score of the Participant in the file.

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 217 – OBJECT ORIENTED PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Farah Sadia, Mr. Syed Zain ul Hassan, Mr Muhammad Fahim

Email: **basit.jasani@nu.edu.pk, farah.sadia@nu.edu.pk, zain.hassan@nu.edu.pk, m.fahim@nu.edu.pk**

Lab # 11

Outline

1. Generic Programming
2. Templates
3. Function Templates
4. Class Templates
5. Examples
6. Exception Handling (Basics)
7. Exercise

GENERIC PROGRAMMING

A generic program abstraction (function, class) defines a general set of operations that will be applied to various types of data.

TEMPLATES

In C++ generic programming is done using templates. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed.

Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, one function or class can be used with several different types of data without having to explicitly recode specific versions for each data type.

KINDS OF TEMPLATES

- Function Templates
- Class Templates

FUNCTION TEMPLATES

Through a generic function, as ingle general procedure can be applied to a wide range of data. For example, the insertion sort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. By creating a generic function, you can define the nature of the algorithm, independent of any data. Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function. In essence, when you create a generic function you are creating a function that can automatically overload itself.

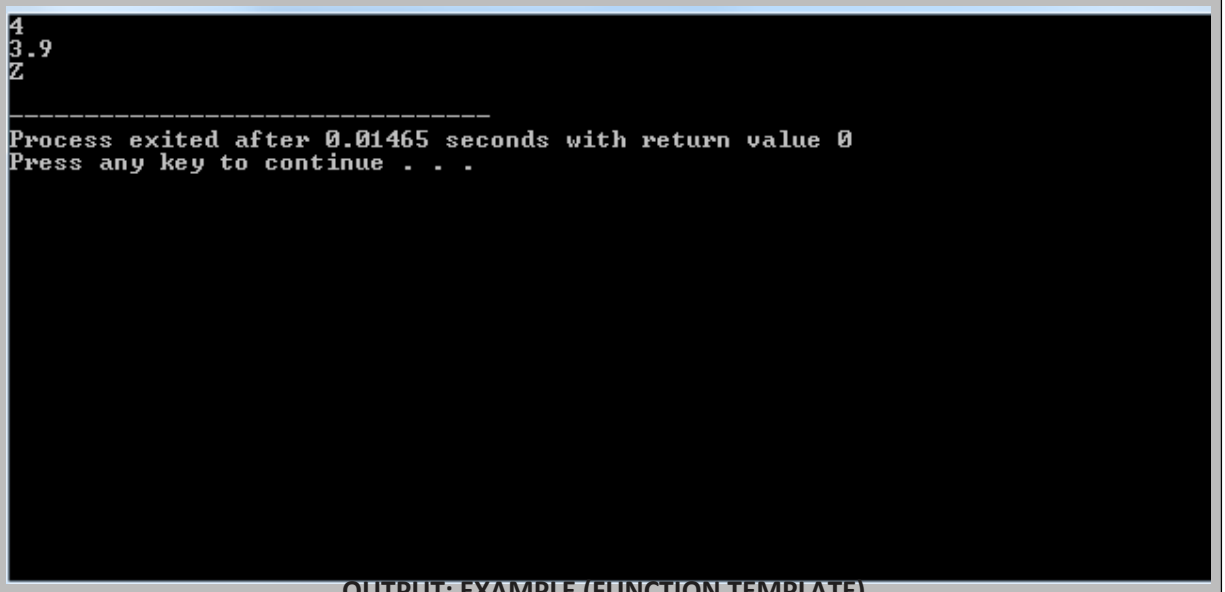
DECLARATION

```
Template < class T>
Void funName (T x);
// OR
Template <typename T >
Void funName (T x);
// OR
Template < class T, class U …>
Void funName (T x, U y …);
```

All function-template definitions begin with the keyword template followed by template parameters enclosed in angle brackets (< and >); each template parameter that represents a type must be preceded by keyword class or type name. Keywords type name and class used to specify function-template parameters mean “any fundamental type or user-defined type.”

EXAMPLE (FUNCTION TEMPLATE)

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

template<class T>
T largest(T a, T b)
{
   return (a>b)?a:b;
}

int main()
{
   int iResult = largest(2, 4);
   cout<<iResult<<endl;

   float fResult = largest(3.5, 3.9);
   cout<<fResult<<endl;

   char cResult = largest('A', 'Z');
   cout<<cResult<<endl;

   return 0;
}
```



OUTPUT: EXAMPLE (FUNCTION TEMPLATE)

A TEMPLATE FUNCTION WITH TWO GENERIC TYPES

More than one generic data type in the **template** statement can be defined by using a comma-separated list. For example, this program creates a template function that has two generic types.

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

template<class type1, class type2>

void myfunc(type1 x, type2 y)
{
        cout<< x << ' ' << y << '\n';
}
int main()
{
        myfunc(10, "I like C++");
        myfunc(98.6, 19);
        return 0;
}
```

The placeholder types "type1 and type2" are replaced by the compiler with the data types int and char *, and double and long, respectively, when the compiler generates the specific instances of myfunc( ) within main( ).

```
10 I like C++
98.6 19

--------------------------------
Process exited after 0.06432 seconds with return value 0
Press any key to continue . . .
```

OUTPUT: TEMPLATE FUNCTION WITH TWO GENERIC TYPES

EXPLICIT TYPE PARAMETERIZATION

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

template <typename T, typename U>
T GetInput( U u )
{
        cout<<u<<endl;
   return u;
}

int main()
{
double d = 10.5674;
double i = GetInput<int>( d );
cout<<i;
return 0;
}
```

USER DEFINED SPECIALIZATION

- A template may not handle all the type successfully.
- Explicit specializations need to be provided for specific type(s)

EXAMPLE (USER DEFINED SPECIALIZATION)

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

template<typename T >
bool isEqual( T x, T y )
 {
return ( x == y );
}

template<const char* >
bool isEqual(const char* x, const char* y )
{ return ( strcmp(x,y)==0);}

int main()
{
cout<<isEqual( 5, 6 )<<endl;
cout<<isEqual( 7.5, 7.5 )<<endl;
cout<<isEqual( "abc", "xyz" )<<endl;
return 0;
}
```

CLASS TEMPLATES

- Just as we can define function templates, we can also define class templates.
- A single class template provides functionality to operate on different types of data
- Facilitates reuse of classes.
- A class or class template can have member functions that are themselves templates
- For example, A **Vector** class template can store data elements of different types

DECLARATION

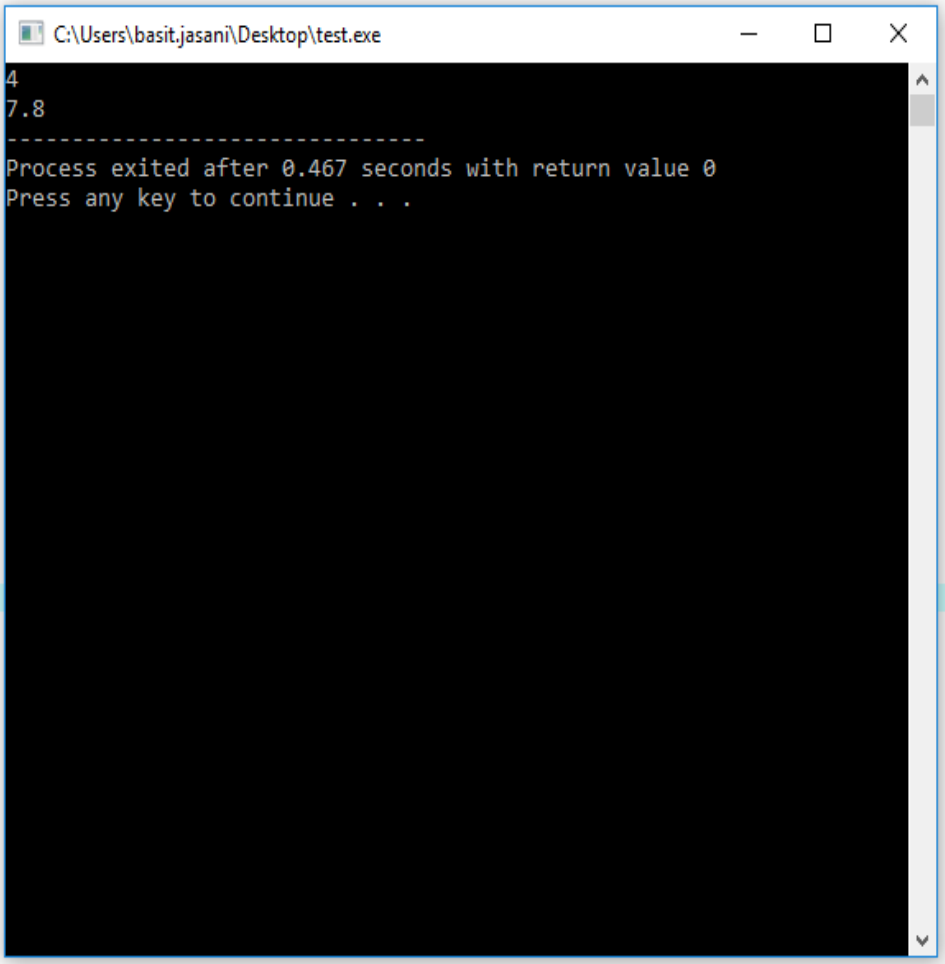
The general form of a generic class declaration is shown here:

```
template<class type>
class class-name
{
}
```

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

EXAMPLE (CLASS TEMPLATE)

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

template<class T>
class myClass
{
public:
T element;

//constructor
myClass(T element)
{
this->element = element;
}

//increment function
T increment()
{
return ++element;
}
};

int main()
{
myClass<int> iOb(3);
int iResult = iOb.increment();
cout << iResult << endl;

myClass<float> fOb(6.8);
float fResult = fOb.increment();
cout<<fResult;

return 0;
}
```

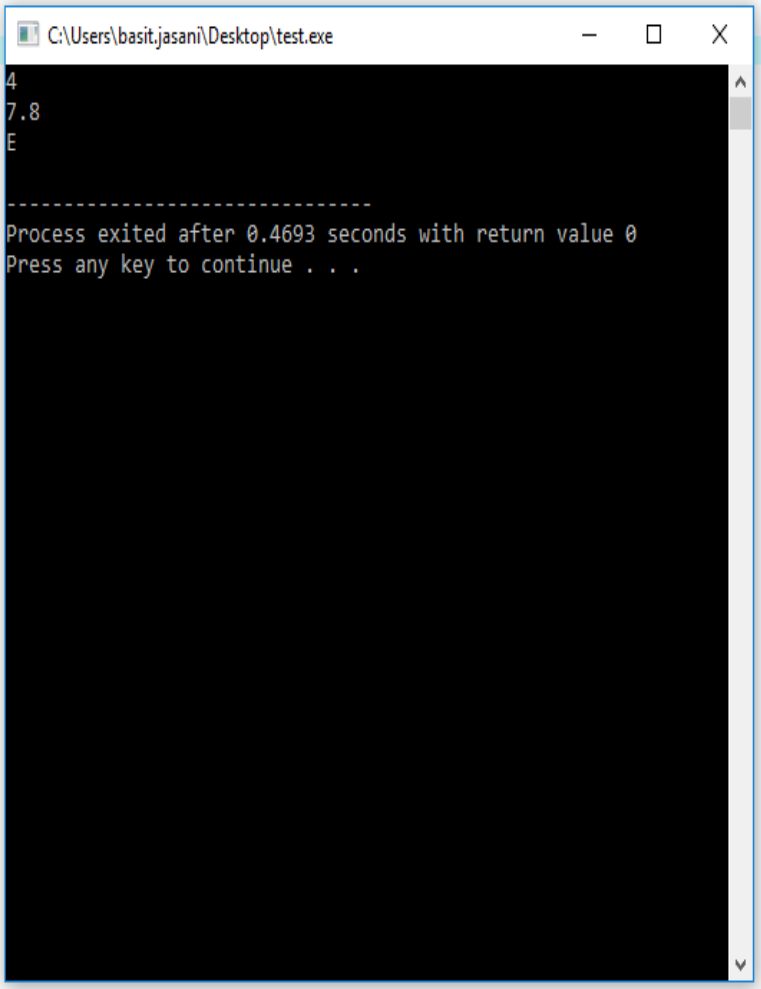


CLASS TEMPLATE SPECIALIZATION

- Like function templates, a class template may not handle all the types successfully
- Explicit specializations are provided to handle such types.
-

EXAMPLE (CLASS TEMPLATE SPECIALIZATION)

```
template<class T>
class myClass
{
public:
T element;
//constructor
myClass(T element)
{
this->element = element;
}
//increment function
T increment()
{
return ++element;
}
};
//specialized class template
template<>
class myClass<char>
{
public:
char element;

myClass(char element)
{
this->element = element;
}
char increment()
{
if(element >= 'a' && element <= 'z')
element += 'A'-'a'; //converting from lowercase touppercase

return element;
}
};
int main()
{
myClass<int> iOb(3);
int iResult = iOb.increment();
cout << iResult << endl;
myClass<float> fOb(6.8);
float fResult = fOb.increment();
cout << fResult << endl;
myClass<char> cOb('e');
char cResult = cOb.increment();
cout << cResult << endl;
return 0;
```



Exception Handling

One of the advantages of C++ over C is Exception Handling. C++ provides following specialized keywords for this purpose.

- **Try:** represents a block of code that can throw an exception.
- **Catch:** represents a block of code that is executed when a particular exception is thrown.
- **Throw:** Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

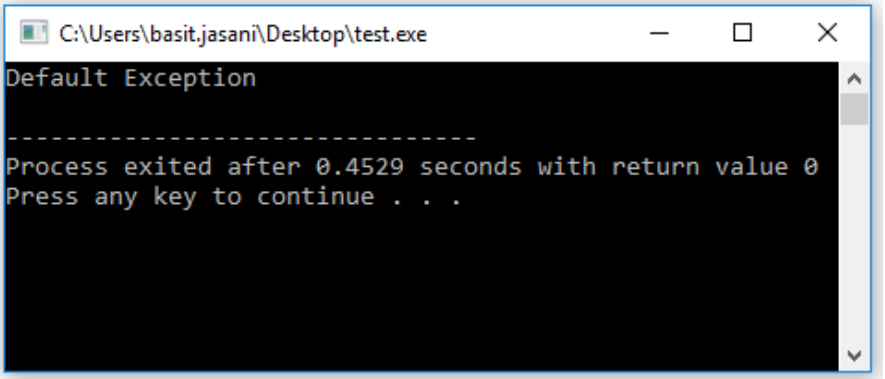
Simple Example:

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

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

```
C:\Users\basit.jasani\Desktop\test....
Inside try
Exception Caught
After catch (Will be executed)

--------------------------------
Process exited after 0.5266 seconds with r
eturn value 0
Press any key to continue . . .
```

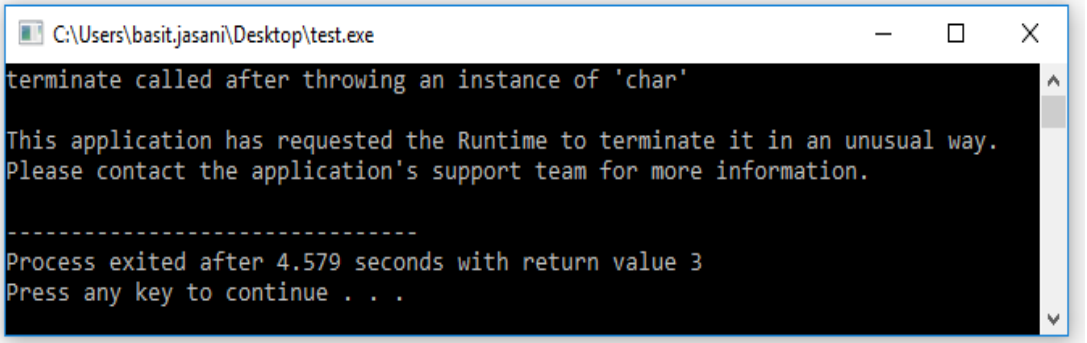
There is a special catch block called ‘catch all’ catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

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

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```



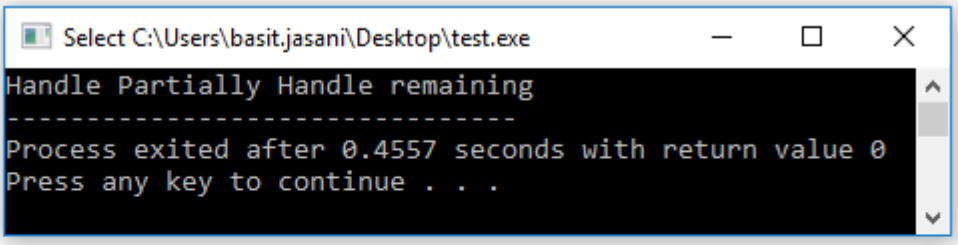
If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```



In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

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

int main()
{
    try {
        try  {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw;   //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```



LAB 11 EXERCISES

INSTRUCTIONS:

NOTE: Violation of any of the following instructions may lead to the cancellation of yoursubmission.

1) Create a folder and name it by your student id(k17-1234).
2) Paste the .cpp file for each question with the names such as Q1.cpp, Q2.cpp and so on into that folder.
3) Submit the zipped folder on slate.

QUESTION#1

Create a class Inventory that contains item code, description, rate per unit and total units in stock.

a) Create 5 objects of inventory.
b) Create a menu based program to handle data file to store these five objects.
c) Also add a function to edit the stock of any given item.
d) Add a function that displays the details of the item whose code is passed as a parameter.
e) Add an overloaded version of the function details () that print all the items with their details.

QUESTION#2

Implement bubble sort algorithm using function template and show the results for the following arrays

1) 7, 5, 4, 3, 9, 8, 6
2) 4.3, 2.5, -0.9, 100.2, 3.0

QUESTION#3

Create a class template with two generic data types to print their addition. Show the results for following types:

- **int** and **double** for example: (10,0.23) would print10.23
- **char*** and **char***
  For example: (“Now”, “Then”) would print NowThen

QUESTION#4

Create a class containing a function template capable of returning the sum of all the elements in an array being passed as parameter. Show the results for two arrays, one integer type and the other double type.

QUESTION#5

Create a function template to multiply two values. Overload this function template for three values and display the results.