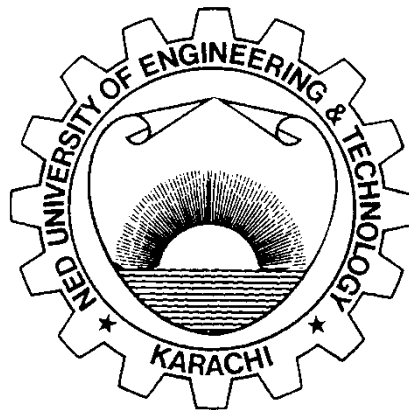


# Practical Workbook

## Object Oriented Programming



Name : \_\_\_\_\_  
Year : \_\_\_\_\_  
Batch : \_\_\_\_\_  
Roll No : \_\_\_\_\_  
Department: \_\_\_\_\_

**Department of Computer & Information Systems Engineering**  
**NED University of Engineering & Technology**  
**Karachi – 75270, Pakistan**



# INTRODUCTION

Object-oriented programming is a programming paradigm that uses abstraction to create models based on the real world. Object-oriented programming is intended to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. By virtue of its strong emphasis on modularity, object oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than less modular programming methods

This laboratory workbook is developed to strengthen topics covered in theory classes. First lab session covers a brief overview of C++ programming and an introduction to Microsoft Visual Studio 2008. Second lab presents introduction to object-orientation. Third lab deals with the concepts of object pointers and dynamic allocation. Subsequent labs engage students in an in-depth examination of essential object-oriented programming concepts. In this course, all object oriented concepts are implemented in C++ language.

Each lab session include many sample programs and programming problems to promote learning and aid students in developing new and useful programming skills.



# CONTENTS

| Lab Session No. | Object   | Page No. |
|-----------------|--|----------|
| 1               | Getting Familiarized with C++ Programming and working with Visual Studio 2008                | 7        |
| 2               | Study of object and classes in Object Oriented Programming                                   | 19       |
| 3               | Working with arrays of objects, pointers to objects and dynamic allocation of objects in C++ | 25       |
| 4               | Study of Inheritance in Object Oriented Programming  | 30       |
| 5               | Study of Polymorphism in Object Oriented Programming   | 34       |
| 6               | Overloading operators in C++   | 40       |
| 7               | Study of Aggregation in Object Oriented Programming  | 44       |
| 8               | Handling Exceptions in C++   | 47       |
| 9               | Reading and writing files by using C++ IO Stream Library                                     | 52       |
| 10              | Implementing Function Templates and Class Templates in C++                                   | 61       |



## Lab Session 01

### OBJECT

*Getting Familiarized with C++ Programming and working with Visual Studio 2008*

### THEORY

#### Introduction to C++

C++ is a high level language with certain low level features as well. C++ is a superset of C. Most of what we already know about C applies to C++ also. However, there are some differences that will prevent C programs to run under C++ compiler. C++ programs differ from C programs in some important respects. Most of the differences have to do with taking advantage of C++'s object-oriented capabilities. But C++ programs differ from C programs in other ways, including how I/O is performed and what headers are included. Also, most C++ programs share a set of common traits that clearly identify them as C++ programs. Before moving on to C++'s object-oriented constructs, an understanding of the fundamental elements of a C++ program is required

#### The New C++ Headers

When you use a library function in a program, you must include its header. This is done using the `#include` statement. Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ created a new kind of header that is used by the Standard C++ library.

The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared. Since the new-style headers are not filenames, they do not have a `.h` extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

`<iostream>`    `<fstream>`    `<vector>`    `<string>`

#### Namespaces

When you include a new-style header in your program, the contents of that header are contained in the `std` namespace. A namespace is simply a declarative region. The purpose of a namespace is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another. Originally, the names of the C++ library functions, etc., were simply put into the global namespace (as they are in C). However, with the advent of the new-style headers, the contents of these headers were placed in the `std` namespace.

#### I/O Operators

```
cout << "This is output. \n";
```

This statement introduces new C++ features, **cout** and **<<**. The identifier **cout** is a predefined object that represents output stream in C++. The standard output stream represents the screen. The operator **<<** is called the insertion operator. It inserts (or sends) the contents of the variable on its right to the object on its left.

Note that you can still use **printf( )** or any other of C's I/O functions in a C++ program. However, most programmers feel that using **<<** is more in the spirit of C++. Further, while using **printf( )** to output a string is virtually equivalent to using **<<** in this case, the C++ I/O system can be expanded to perform operations on objects that you define (something that you cannot do using **printf( )**).

The statement **cin<<num1;** is an input statement and causes program to wait for the user to type in number. The identifier **cin** is a predefined object in C++ that corresponds to the standard input stream. Here stream represents the keyboard. The operator **>>** is known as extraction or get from operator. In general, you can use **cin >>** to input a variable of any of the basic data types plus strings.

## A Sample C++ Program

```
#include <iostream>
using namespace std;
int main()
{ int i;
  cout << "This is output.\n"; // this is a single line comment
  /* you can still use C style comments */
  // input a number using >>
  cout << "Enter a number: ";
  cin >> i;
  // now, output a number using <<
  cout << i << " squared is " << i*i << "\n";
  return 0;
}
```

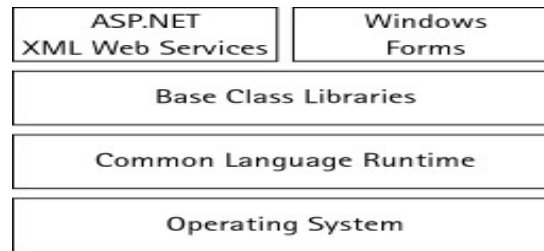
Notice that the parameter list in **main( )** is empty. In C++, this indicates that **main( )** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty.

## Introduction to dot net framework

.NET is a collection of tools, technologies, and languages that all work together in a framework to provide the solutions that are needed to easily build and deploy truly robust enterprise applications. These .NET applications are also able to easily communicate with one another and provide information and application logic, regardless of platforms and languages.

Figure 1 shows an overview of the structure of the .NET Framework.





**Figure 1.1 Structure of .Net Framework.**

The first thing that you should notice when looking at this diagram is that the .NET Framework sits on top of the operating system. Presently, the operating systems that can take the .NET Framework include Windows XP, Windows 2000, and Windows NT. There has also been a lot of talk about .NET being ported over by some third-party companies so that a majority of the .NET Framework could run on other platforms as well.

At the base of the .NET Framework is the Common Language Runtime (CLR). The CLR is the engine that manages the execution of the code.

The next layer up is the .NET Framework Base Classes. This layer contains classes, value types, and interfaces that you will use often in your development process. Most notably within the .NET Framework Base Classes is ADO.NET, which provides access to and management of data.

The third layer of the framework is ASP.NET and Windows Forms. ASP.NET should not be viewed as the next version of Active Server Pages after ASP 3.0, but as a dramatically new shift in Web application development. Using ASP.NET, it's now possible to build robust Web applications that are even more functional than Win32 applications of the past.

The second part of the top layer of the .NET Framework is the Windows Forms section. This is where you can build the traditional executable applications that you built with Visual Basic 6.0 in the past. There are some new features here as well, such as a new drawing class and the capability to program these applications in any of the available .NET languages

## Introduction to visual studio 2008

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft for creating, documenting, and debugging programs written in a variety of .net programming languages.

It is used to develop console and graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

## Working With Visual Studio 2008

Open Visual Studio 2008. You will see the start page.

Select "File Menu-->New-->Project" (as shown in Figure 1.2).

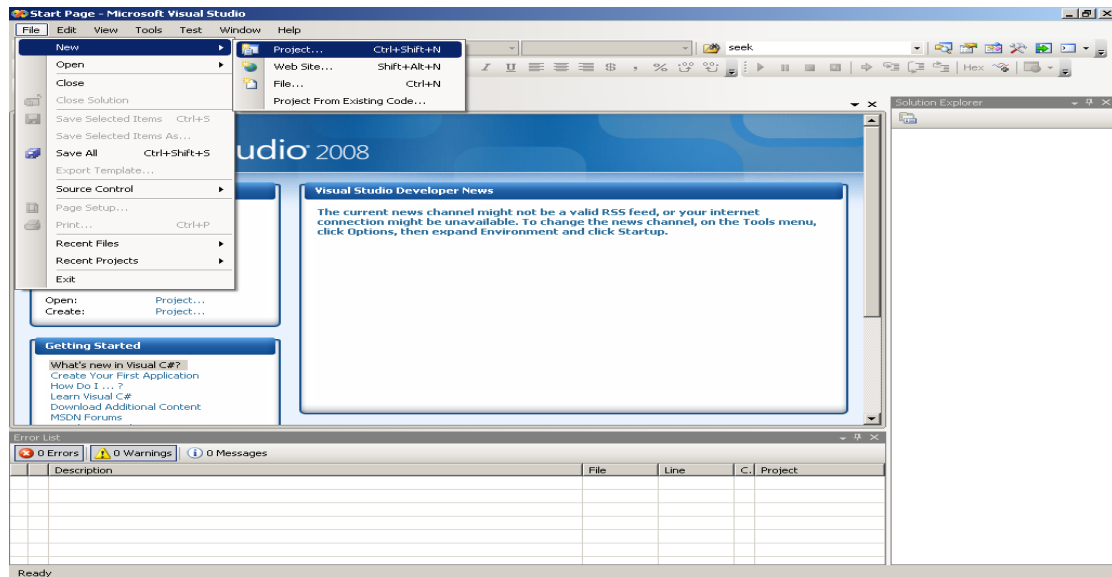


Figure 1.2

In the dialog box that pops up, select Visual C++ in project type and select "Win32 Console Application." Enter the name of project and leave the default path the same (Figure 1.3--You can leave the Create Directory for Solution box checked and you can change the path also). Click ok.

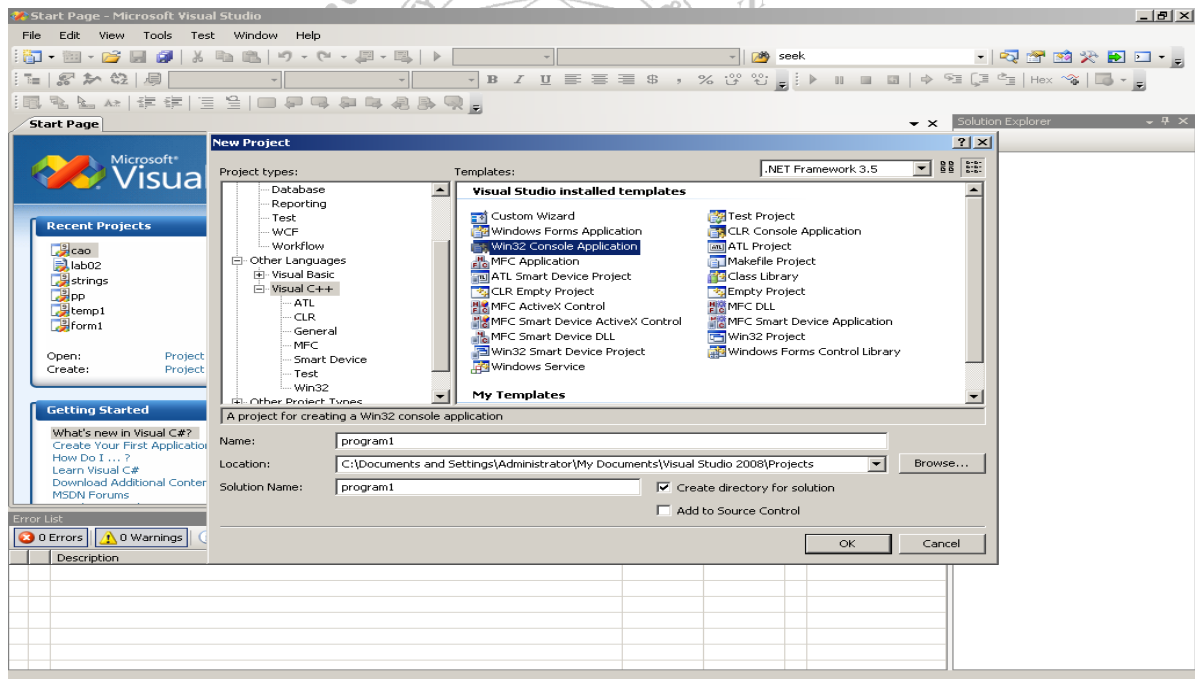


Figure 1.3

The Win32 Application Wizard will pop up (as shown in Figure 1.4).

Make sure that the following are checked (as shown):

1. Under "Application type": Console Application
2. Under "Additional Options": Empty Project

Then Press Finished

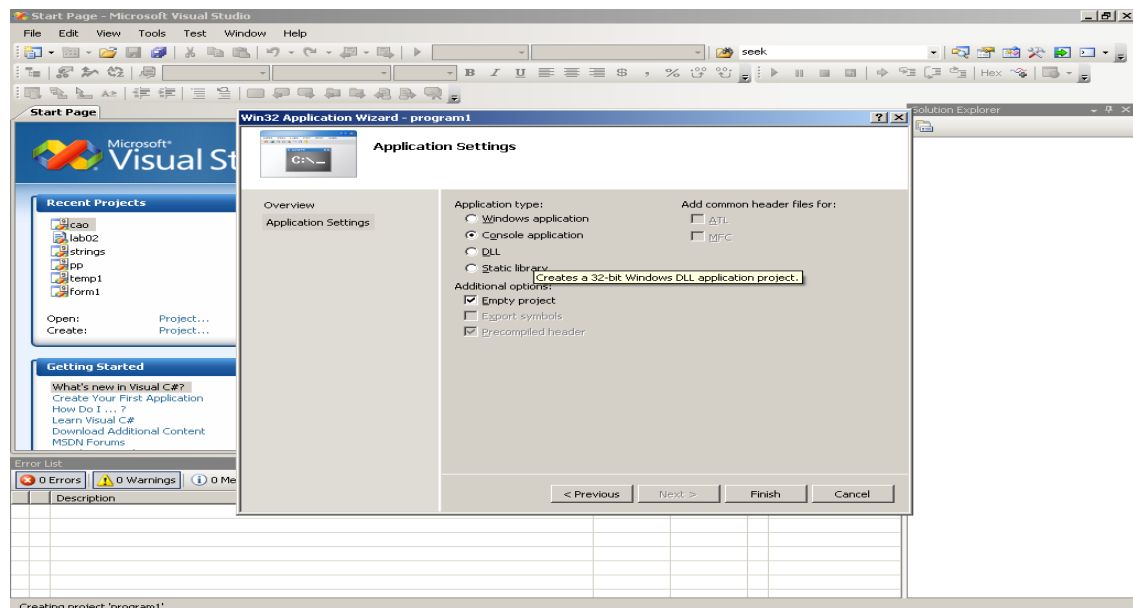


Figure 1.4

After clicking on finish, project will be created. Now you need to add a CPP file to the project. Go to the "Solution Explorer" on the left side and right-click on "Program1" (name of project). Once you have right clicked on Program1 select "Add-->New Item" (Figure 1.5).

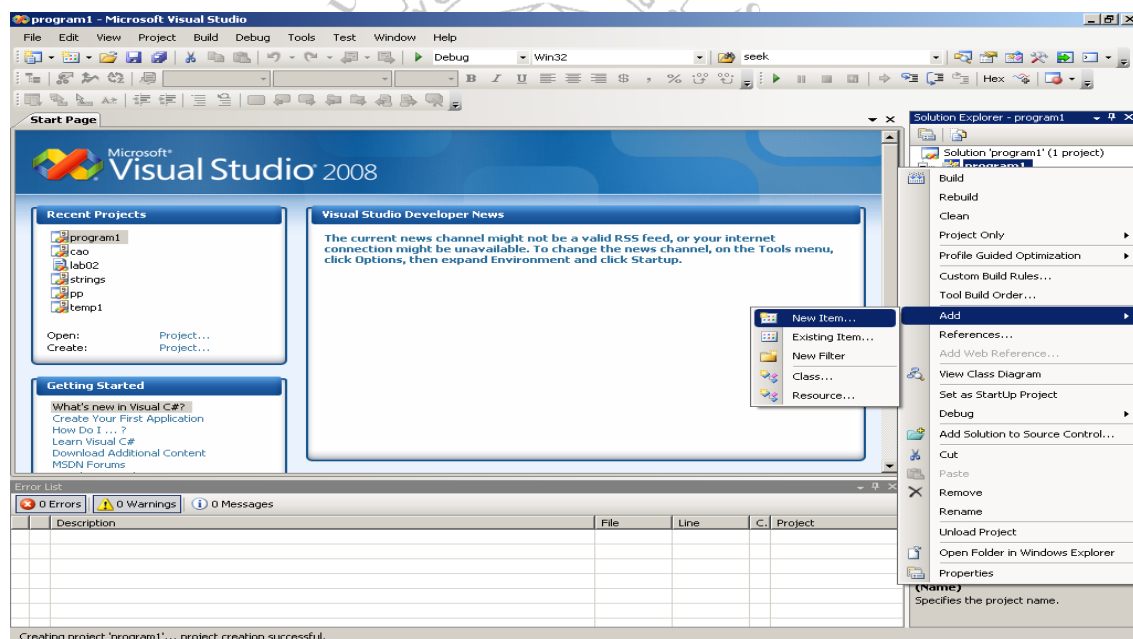


Figure 1.5

In the Dialog Box that appears, select "C++ file (.cpp)" and change the name to program1 (as shown in figure 1.6). Press "OK".

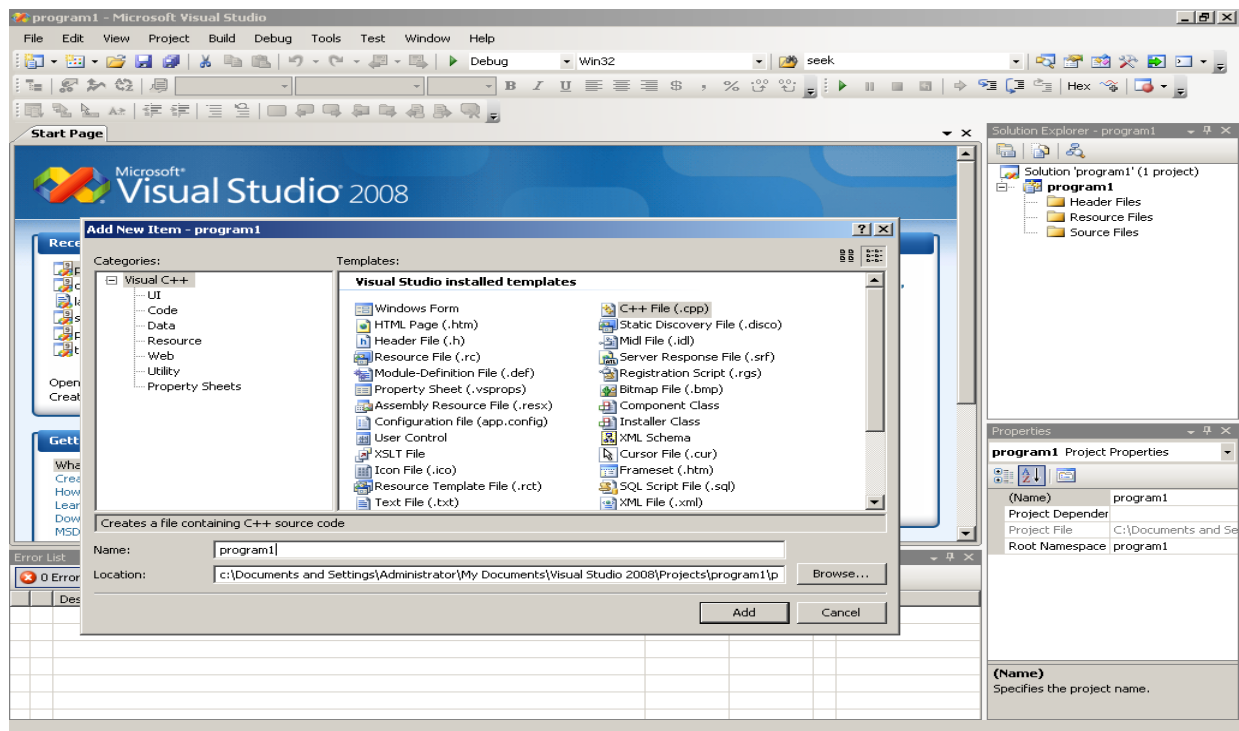


Figure 1.6

A blank screen will appear showing the contents of "program1.cpp". Type the following code into this blank white area

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

// declaring function prototypes
float addition (float a, float b);
//main function
int main ()
{
    float x;           //
    float y;           //declares variables
    float n;           //
    int b;
    b = 1;              //sets value of b to 1
    cout << "Simple Addition Calc- First Program";    //displays info about
program
    while (b==1) //creates loop so the program runs as long as the person
wants to add numbers
    {
        //following code prompts the user for 2 numbers to add and calls
function addition to display results
        cout << "\n" << "Type a number to add (can also use negative,
decimals, etc.): ";
        cin >> x;
        cout << " Second number: ";
```

```

    cin >> y;
    n = addition (x,y);
    cout << "Ans: " << x << " + " << y << " = " << n << "\n";

//following code sets b to the value the user inputs to determine if the
loop is broken to end the program
    cout << "Solve another operation? (1=yes, 2=no): ";
    cin >> b;
    cout << "\n";
    if (b==2)
        cout << "Terminating application.";
    } //ends the main function of the code
    return 0;    }

//following function adds the numbers
float addition (float a, float b)
{
    float c;
    c = a+b;
    return (c);} //END C++ CODE

```

The blank area with the code in it is shown below (Figure 1.7)

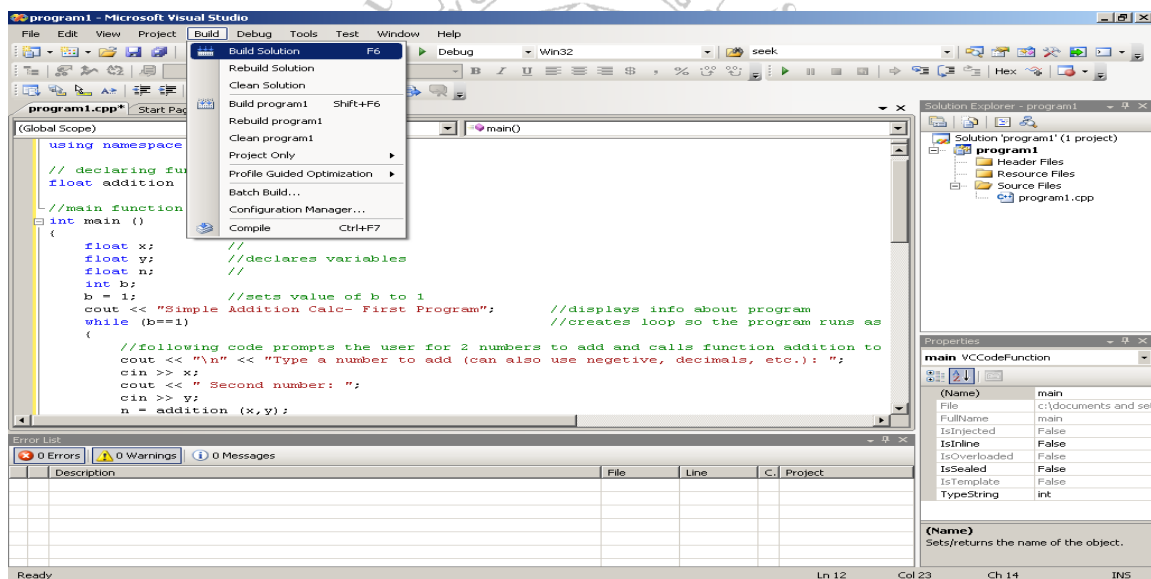


Figure 1.7

Go to the Build Menu and select "Build Solution" (Figure 1.7).

Go to the Debug Menu and select "Start Without Debugging" or simply PRESS F5 on your keyboard, and a small box should pop up that is running your program that will add 2 numbers together (Figure 1.8). This is command to debug your application,

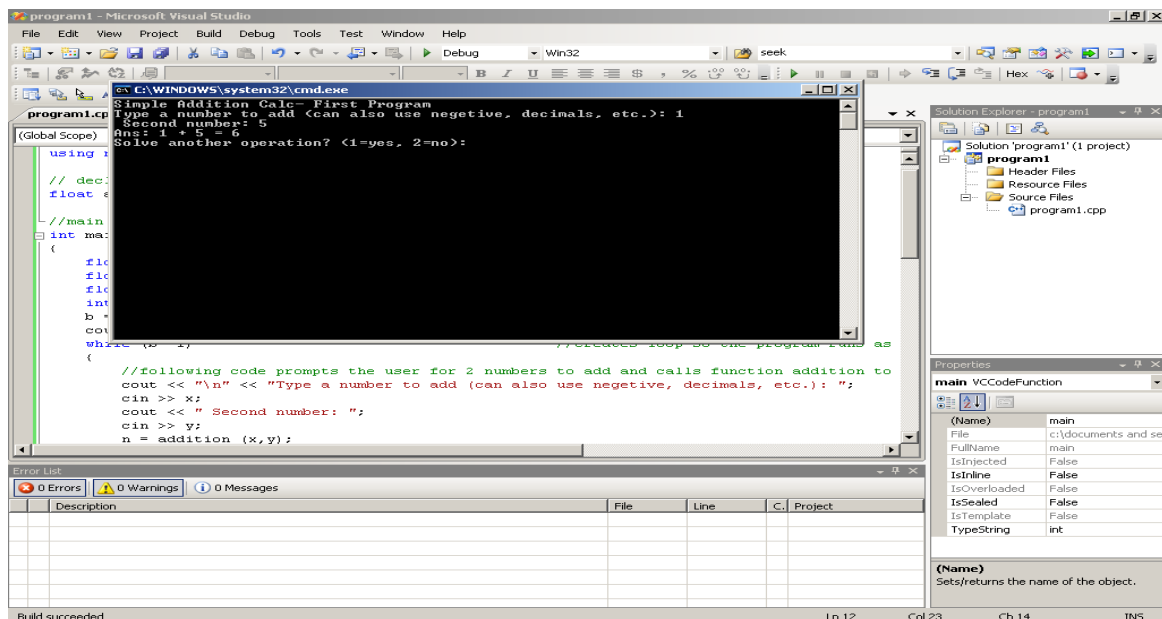


Figure 1.8

## Debugging Your program in Visual Studio 2008

The most basic aspect of any debugging session is the use of breakpoints, which allows you to specify a location in your application (a line of code) where program execution will stop (break) and allow you to inspect the internal state of the application at that exact moment. To add a break point place mouse pointer on the gutter (narrow gray strip on left of code) and click on it in the line you want to debug. A red circle will appear, as shown in figure 1.9

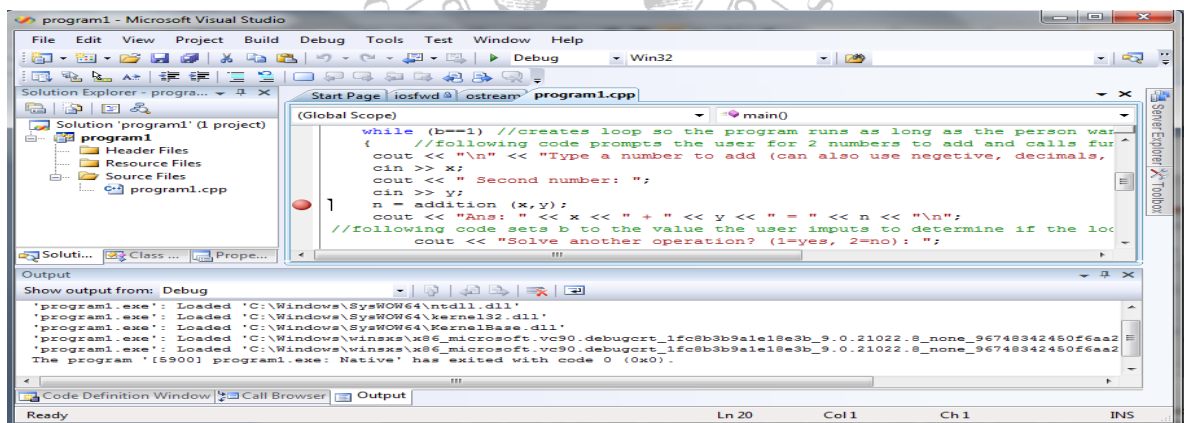


Figure 1.9

To run your program in debugging mode, select 'Start Debugging' from Debug menu. Program execution will stop at the line of break point (see Figure 1.10). Yellow arrow is pointing to the line which will be executed next. At this point, you can view the contents of all variable in different tabs of bottom left window.

**Watch** will allow the user to select the variables and objects in the program and display their values

**Autos** is very similar to the watch and locals windows except along with the value of the object or variable, it also displays the expressions from the current line of code some of it around it. You can see that it picks up more than just the current scope of the instruction pointer; it has picked up the variables from the entire function

**Locals** Similar to watch except that it displays all the local variables and objects that are being referenced in the current scope.

**Threads** Gives an overview of the currently running threads, their ID's category, Name, Location and priorities

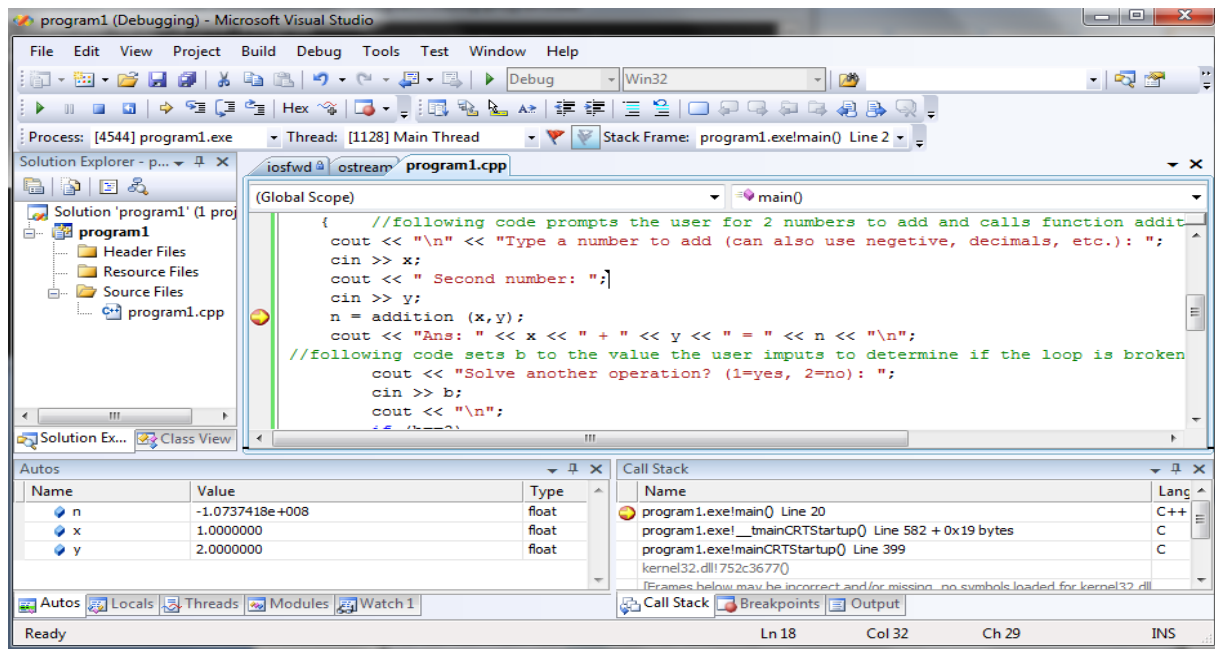


Figure 1.10

Bottom Right window allow you to view the following:

**Call Stack** is very useful when a breakpoint is hit or when the program calls an exception. Assuming the call stack is intact, it gives the user a history of program calls to get to the stop execution point. The user can then select any of the entries to go back in time of the execution and trace backwards things that were happening.

**Breakpoints** window displays the status and the descriptive items associated with program breakpoints

**Output:** The 'Trace' window, it is usually displayed in both code and debug views.

Now you can use following options for single stepping from Debug menu

**Step Into (F11):** Step Into executes only the call itself, then halts at the first line of code inside the function. Use Step Into to go inside your method and stop immediately inside it.

**Step Over:** Moves to the next step in your code also, but doesn't break inside any methods.

**Step Out:** Ignores the rest of the current method and goes to the calling method.

After debugging desired portion of code, you can execute the remaining code at once by selecting Continue from Debug menu.



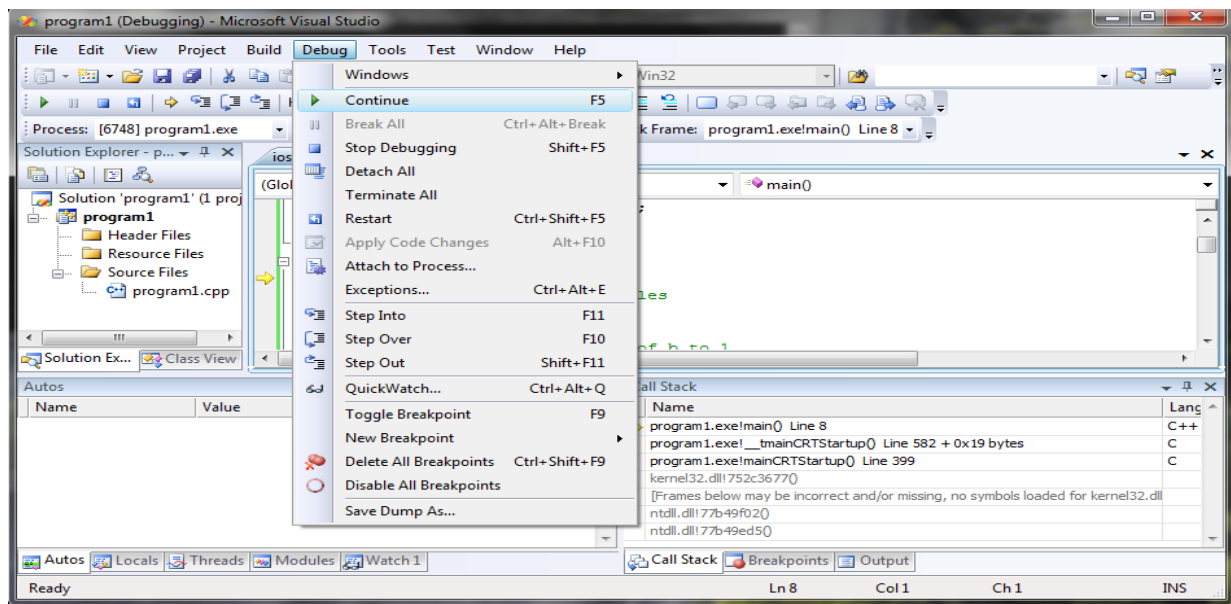


Figure 1.11

## Advanced Breakpoints

In order to set an advanced breakpoint, you must first set a normal breakpoint and then choose a modifier to set from a context menu. To display the context menu from a breakpoint symbol in the gutter, simply right click mouse on the breakpoint symbol or the line containing the breakpoint.(see Figure 1.12)

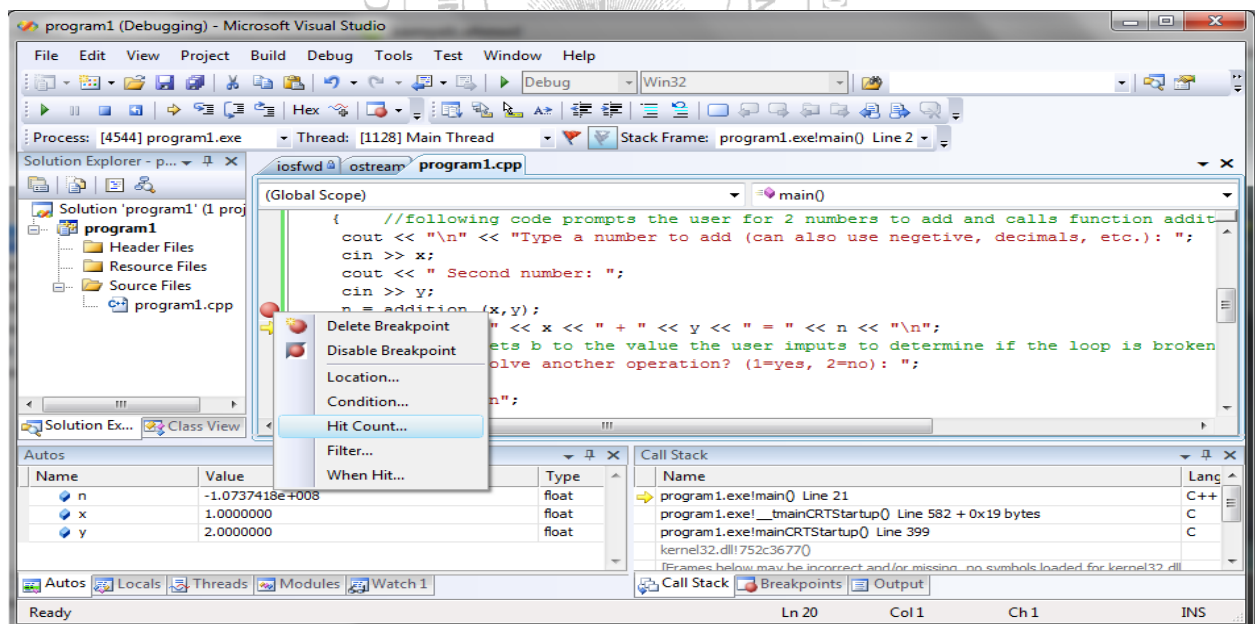


Figure 1.12

**Location** option is used to set the location of breakpoint.

**Condition** option allows you to restrict when the breakpoint will hit based on the evaluation of a Boolean expression. You can choose to hit the breakpoint when the condition is true or when the result of the condition has changed.(Figure 1.13)



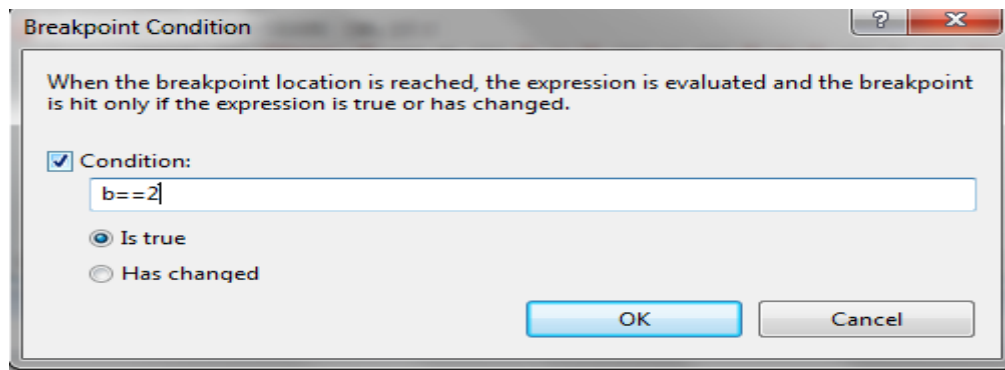


Figure 1.13

**Hit count** is the number of times the breakpoint location has been hit. By setting the hit count modifier, the breakpoint will only hit when the condition is satisfied.(Figure 1.14)

- **break always:** Behaves just like a normal break point.
- **break when the hit count is equal to:** Stop only when the hit count equals the initial value.
- **break when the hit count is a multiple of:** Stop every  $x$  times.
- **break when the hit count is greater than or equal to:** Continues to stop after equaling the initial value.

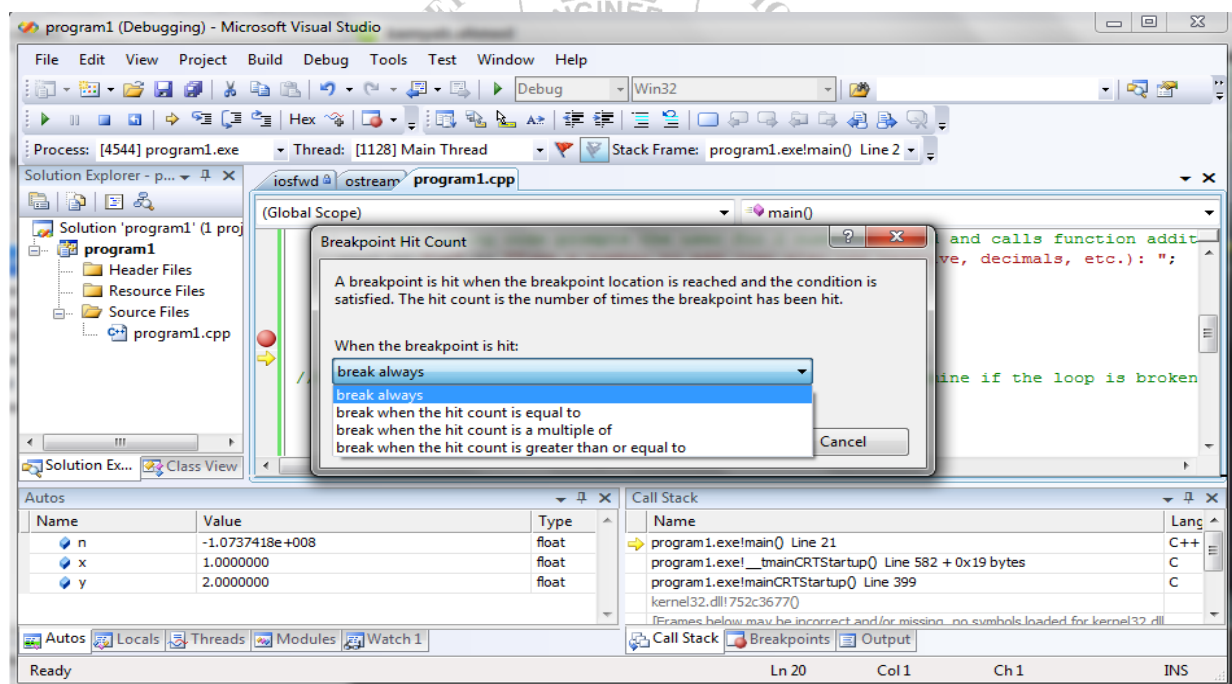


Figure 1.14

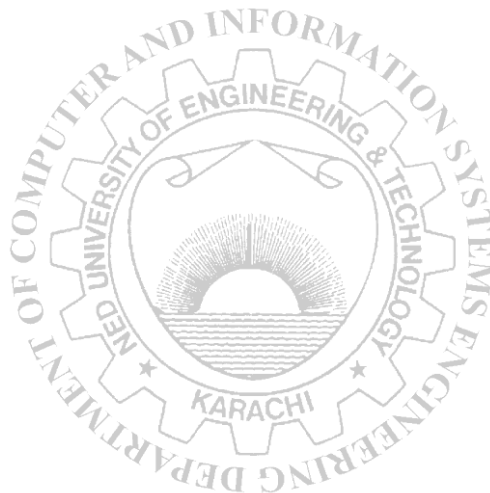
## Applications of advanced breakpoint techniques

Advanced breakpoints are very helpful in debugging in following cases:

- Loop counting
- Recursion debugging
- Counting number of times a variable changes

## Exercise

Q1. Write a program to implement four function calculator that can Add, Subtract, Multiply and Divide two numbers, taken from user as input. Also allow user to select the operation to be performed.



## Lab Session 02

### OBJECT

*Study of object and classes in Object Oriented Programming*

### THEORY

Object-Oriented Programming (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. Object-oriented programming is a paradigm in which a software system is decomposed into subsystems based on objects. Computation is done by objects exchanging messages among themselves. The paradigm enhances software maintainability, extensibility and reusability.

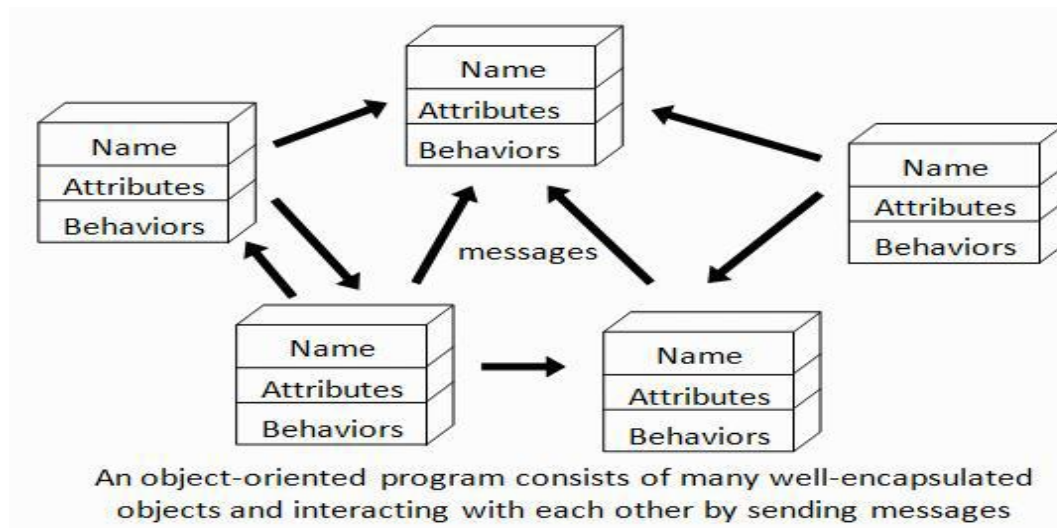


Figure 2.1

### CLASSES AND OBJECTS

Object is the basic unit of object oriented programming. Object represents a Physical/real entity. All real world objects have three characteristics:

- State/Attributes: The states of an object represent all the information held within it
- Behavior: Behavior of an object is the set of action that it can perform to change the state of the object.
- Identity: Object is identified by its unique name

Classes are data types based on which objects are created. Objects with similar attributes and methods are grouped together to form a class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.

Classes are created using the keyword `class`. A class declaration is similar syntactically to a structure. The attributes of object are represented by variables or data structures (arrays, list etc).

The behavior is specified by defining methods, also known as member functions.

```
class classname
{
    access specifier:
    data member;
    member functions;
:
    access specifier:
    data member;
    member functions;
};
```

Here, access-specifier is one of the three C++ keywords:

- `public`
- `private`
- `protected`

By default, functions and data declared within a class are private to that class and can be accessed only by other members of the class. The public access specifier allows functions or data to be accessible to other parts of your program. The protected access specifier is needed only when inheritance is involved. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

## Inline and Out of line methods

Member functions defined within the class definition are called inline functions. Member functions which are only declared in the class definition, and defined outside it, are called out-of-line. The implementation code of inline methods is duplicated for every function call.

The **inline** functions allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. However, when a function is expanded in line, none of the calling and return mechanism occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

## Sample Program

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

class BankAccount{
```

```

    int Account_no;
    double Balance;
public:
    void setBalance(double b);
    void setAccno(int no);
    double getBalance();
    int getAccno();
    void display();
    void withdraw(double amount);
    void deposit(double amount);
};

void BankAccount::setBalance(double b)
{ Balance=b;}
void BankAccount::setAccno(int no)
{ Account_no=no;}
double BankAccount::getBalance()
{ return Balance;}
int BankAccount::getAccno()
{ return Account_no;}
void BankAccount::display()
{ cout<< "Account no is"<< Account_no << "\n Balance is"<< Balance;}

void BankAccount::withdraw(double amount)
{ if(amount>=Balance)
    Balance-=amount;}
void BankAccount::deposit(double amount)
{ Balance+=amount;}

int main()
{
    BankAccount b1;
    BankAccount b2;
    b1.setAccno(1);
    b1.setBalance(5000.0);
    b2.setAccno(2);
    b2.setBalance(4500.0);
    b1.display();
    b2.display();
    b1.withdraw(500.0);
    b2.deposit(1000.0);
    b1.display();
    b2.display();
    return 0;
}

```

## CONSTRUCTORS

It is very common for some part of an object to require initialization before it can be used. C++ allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor function. A constructor is a special function that is a member of a class and has the same name as that class.

It is also possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object. For example, we can add a parameterized constructor for **BankAccount** class.

```

#include <iostream>
using namespace std;

```

```

Class BankAccount{
    int Account_no;
    double Balance;
public:
    void setBalance(double b);
    void setAccno(int no);
    double getBalance();
    int getAccno();
    void display();
    void withdraw(double amount);
    void deposit(double amount);

//constructor declaration
    BankAccount(int Accno, double Bal);
};
// constructor definition
BankAccount::BankAccount(int Accno, double Bal)
{
    Account_no=Accno;
    Balance=Bal;
}
// get, set method definitions
int main()
{
    BankAccount b1(1,2000.0);
    BankAccount b2(2, 1000.0);
    b1.display();
    b2.display();
    b1.withdraw(500.0);
    b2.deposit(1000.0);
    b1.display();
    b2.display();
    return 0;
}

```

Even in the presence of parameterized constructor, you may still need set methods to edit the attributes of an object.

### Constructor with Default Arguments

It's possible to defined constructor with default arguments. For e.g. BankAccount( ) can be declared as

```
BankAccount(int Accno, double Balance=500.0 );
```

The default value of the argument **Balance** is **zero**. Then, the statement BankAccount B(1); assigns the value 1 to the **Accno** variable and 500.0 to **Balance** (by default).

### Copy Constructor

A copy constructor is used to declare and initialize an object from another object. For e.g. the following statement would define the object b2 and at the same time initialize it to b1

```
BankAccount b2(b1); // b1 is an object of BankAccount type
```

The process of initializing through a copy constructor is known as *copy initialization*. A copy constructor takes a reference to an object of the same class as itself as an argument. Here is the code for copy constructor of BankAccount class

```

BankAccount::BankAccount (BankAccount& b)
{
    Account_no=b.Account_no;
    Balance=b.Balance
}

```

```
}
```

## DESTRUCTOR

A destructor as name implies, is used to destroy the objects that have been created by a constructor. Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by tilde. For Example, the destructor for the class `BankAccount` can be defined as shown below

```
~BankAccount() { }
```

A destructor never takes any argument nor does it returns any value. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. If you want to perform some specific task when an object is destroyed, you can define the destructor in the class. If you don't define, the default one is called.

## FRIEND FUNCTIONS

It is possible to grant a nonmember function access to the private members of a class by using a friend. A friend function has access to all private and protected members of the class for which it is a friend. To declare a friend function, include its prototype within the class, preceding it with the keyword `friend`. Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
    a = i;
    b = j;
}
// Note: sum() is not a member function of any class.
int sum(myclass x)
{
    /* Because sum() is a friend of myclass, it can directly access a and b. */
    return x.a + x.b;
}
int main()
{
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
    return 0;
}
```

In this example, the `sum( )` function is not a member of `myclass`. However, it still has full access to its private members. Also, notice that `sum( )` is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

## Exercise

Q1. What is the difference between object and a class?

---

---

---

Q2. What is the criterion for defining a functions **inline** or **out of line**?

---

---

---

Q3. Define a class **Complex\_No** that has two member variables; **Real** and **Imaginary**. Also include following in the class

- A parameterized constructor that takes Real and Imaginary values as argument.
- A default constructor that assign zero to Real and Imaginary.
- A copy constructor
- A method **Display** that shows the value of complex number in appropriate format.
- A method **Magnitude** that calculates the magnitude of complex number
- A method **Add** that adds two complex numbers and return result; take one complex number as argument.

Write a driver program to test your class

Q4. Define a class **Counter** having an attribute **value**. Provide a constructor that initializes value to zero. Also provide following methods:

- Increment () : that increment the value by one.
- Decrement () : that decrement the value by one.

Q5. Define a function **Reset** that takes a **Counter** type object as input and resets its **value** to zero. Make this function a friend of **Counter** class

Q6. Define a class **Student** that has following attributes:

**Name:** allocated dynamically by a character pointer.

**Rollno:** an integer.

**Marks:** a double type array of 5 elements.

**Percentage:** a float

Include a constructor that takes values of Name, Rollno and Marks from user as input. Also include following methods:

**CalculatePercentage:** that adds all 5 elements of array Marks and calculate percentage according to formula  $\text{Percentage} = (\text{Total marks} / 500) * 100$  and stores result in member variable Percentage.

**Grade:** that calls CalculatePercentage method and displays the grade accordingly

Write a driver program to test your class.



## Lab Session 03

### OBJECT

*Working with arrays of objects, pointers to objects and dynamic allocation of objects in C++*

### THEORY

#### Arrays Of Objects

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. The general format is:

*classname ob[3]; // declaring array of objects of **classname***

*ob[i].functionname(); // calling member function using object of array*

#### *Initializing Arrays of objects*

If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax

```
classname ob[3] = {1, 2, 3}; // initializing one-variable class; short form
classname ob[3] = { cl(1), cl(2), cl(3) }; // initializing one-variable class; long form
```

As each element in the array is created, a value from the list is passed to the constructor's parameter. The short form is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument. Thus, the short form can only be used to initialize object arrays whose constructors require one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example:

```
classname ob[3] = {cl(1,2), cl(3,4), cl(5,6)}; //initializing two-variable class
```

#### Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class, given a pointer to an object, use the arrow (→) operator instead of the dot operator. The following program illustrates how to access an object given a pointer to it:

```
#include <iostream>
using namespace std;
class cl {
    int i;
public:
    cl(int j) { i=j; }
    int get_i() { return i; }
```

```
};

int main()
{
    cl obj(88), *p;
    p = &obj; // get address of obj
    cout << p->get_i(); // use -> to call get_i()

    // initializing array of objects
    cl objarray[3] = {1, 2, 3};
    int i;
    p = objarray; // get start of array
    for(i=0; i<3; i++) {
        cout << p->get_i() << "\n";
        p++; // point to next object
    }
    return 0;
}
```

when a pointer is incremented, it points to the next element of its type. The same is true of pointers to objects. The above program uses a pointer to access all three elements of array **objarray** after being assigned **objarray**'s starting address.

### Initializing Pointer Variables

C++ does not automatically initialize variables. Pointer variables must be initialized if you do not want them to point to anything. Pointer variables are initialized using the constant value 0, called the **null pointer**. The statement `p = 0;` stores the null pointer in `p`, that is, `p` points to nothing. Some programmers use the named constant `NULL` to initialize pointer variables. The following two statements are equivalent:

```
p = NULL;
p = 0;
```

The number 0 is the only number that can be directly assigned to a pointer variable.

### Dynamic Allocation

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:

```
p_var = new type;
delete p_var;
```

Here, `p_var` is a pointer variable that receives a pointer to memory that is large enough to hold an item of type `type`. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

Here is a short program that creates a class called **balance** that links a person's name with his or her account balance. Inside **main()**, an object of type **balance** is created dynamically.

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

```

class balance {
    double cur_bal;
    char name[80];
public:
    balance(double n, char *s) {
        cur_bal = n;
        strcpy(name, s);
    }
    ~balance() {
        cout << "Destructing ";
        cout << name << "\n";
    }
    void get_bal(double &n, char *s) {
        n = cur_bal;
        strcpy(s, name);
    }
};

int main()
{
    balance *p;
    char s[80];
    double n;
    p = new balance(12387.87, "Ralph Wilson");
    p->get_bal(n, s);
    cout << s << "'s balance is: " << n;
    cout << "\n";
    delete p;
    return 0;
}

```

You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructors, one will be parameter less. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program. To allocate an array of object of class **balance**, modify the definition of class **balance** as follows.

```

#include <iostream>
#include <cstring>
using namespace std;
class balance {
    double cur_bal;
    char name[80];
public:
    balance(double n, char *s) {
        cur_bal = n;
        strcpy(name, s);
    }
    balance() {} // parameterless constructor
    ~balance() {
        cout << "Destructing ";
        cout << name << "\n";
    }
    void set(double n, char *s) {
        cur_bal = n;
        strcpy(name, s);
    }
    void get_bal(double &n, char *s) {
        n = cur_bal;
        strcpy(s, name);
    }
}

```

```

    }

};

int main()
{
    balance *p;
    char s[80];
    double n;
    int i;
    p = new balance [3]; // allocate entire array
    // note use of dot, not arrow operators
    p[0].set(12387.87, "Ralph Wilson");
    p[1].set(144.00, "A. C. Conners");
    p[2].set(-11.23, "I. M. Overdrawn");
    for(i=0; i<3; i++) {
        p[i].get_bal(n, s);
        cout << s << "'s balance is: " << n;
        cout << "\n";
    }
    delete [] p;
    return 0;
}

```

## Exercise

Q1. Write a program that takes the record of 10 students from user in an array and display all the records.[use Student class, defined in lab session 02]

Q2. Define a pointer to student class to access the contents of array defined in Q1. Allow user to search a record in array by means of Rollno

Q3. Develop a class to represent an integer array. The member variables include an integer to represent the size of array and an integer pointer to represent the address of the first element of the array. The user is allowed to create an array at runtime using this class. Include appropriate constructors (parameterized and copy). Also include a method that adds the contents of array.

Q4. Consider following code:

```

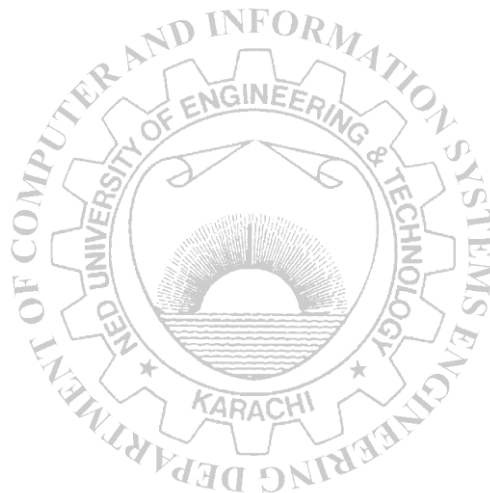
class myclass
{int data[2];
public:
    int* p;
public:
    myclass()
    {p=data;}
};

int main()
{ myclass* cp;
  cp=new myclass[3];
  return 0;
}

```

How would you access the contents of *data* of each element of *myclass* array? Add code in the above program to do the following:

- a. Assign values to array *data* of each element of *myclass* array.
- b. Display contents of *data* of each element of *myclass* array



## Lab Session 04

### OBJECT

#### *Study of Inheritance in Object Oriented Programming*

### THEORY

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class. In keeping with standard C++ terminology, a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class. Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

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

The access status of the base-class members inside the derived class is determined by access.

| Access Type | Description   |
|-------------|---|
| Private     | All public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. |
| Protected   | All public and protected members of the base class become protected members of the derived class.   |
| Private     | All public and protected members of the base class become private members of the derived class.   |

In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

### Sample Program

```
#include <iostream>
using namespace std;
class base {
protected:
    int i, j;
public:
    void set(int a, int b) { i=a; j=b; }
    void show() { cout <<"i="<< i << " j=" << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base {
    int k;
```

```

public:
    void setk() { k = i*j; } // legal
    void show() { // overriding base class method show()
        base::show(); // calling base class method
        cout << "k=" << k << "\n";
    }
};
// i and j inherited indirectly through derived1.

class derived2 : public derived1 {
    int m;
public:
    void setm() { m = i-j; } // legal
    void show() {
        derived1::show(); // calling derived1 method
        cout << "m=" << m << "\n";
    }
};

int main()
{
    base ob;
    derived1 ob1;
    derived2 ob2;
    ob.set(1,2);
    ob.show(); // calling base class method
    ob1.set(2, 3);
    ob1.setk();
    ob1.show(); // calling derived1 method
    ob2.set(3, 4);
    ob2.setk();
    ob2.setm();
    ob2.show(); // calling derived2 method
    return 0;
}

```

In the above example, if base were inherited as private, then all members of base would become private members of derived1, which means that they would not be accessible by derived2.

## Function Overriding

A derive class can redefine a method, already defined in its base class. This is called function overriding, as shown in above program. When a method is called by reference of a derived class object, compiler first search it in derived class, if found, compiler executes it and if not found, the compiler will execute the base class method. In the above program, derived1 and derived2 classes override the method show(), already defined in their base classes.

## Order of Execution of Constructors and Destructors

When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor i.e. constructors are executed in their order of derivation. Destructors are executed in reverse order of derivation. Following Program demonstrates this.

```
#include <iostream>
```

```
using namespace std;
class base {
public:
    base() { cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }

};
class derived1 : public base {
public:
    derived1() { cout << "Constructing derived1\n"; }
    ~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
    derived2() { cout << "Constructing derived2\n"; }
    ~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
    derived2 ob;
    // construct and destruct ob
    return 0;
}
```

## OUTPUT

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

## Exercise

Q1. If A is derived from B and B is derived from C and an object of class A is created. What will be the sequence of constructor execution? What will be the sequence of destructor execution?

---

---

---

Q2. What would you do if you want the public methods of base class to be accessible by the derived class, but not by objects of the derived class? Mention syntax.

---

---

---

---

Q3. Extend the **Student** class( defined in lab session 02) to represent science and arts student. Science student has additional practical course of 150 marks while arts student has an optional course of 100 marks. In each class redefine **CalculatePercentage** method.



Science student:  $\text{Percentage} = \text{Total marks} / 650 * 100$

Art student:  $\text{Percentage} = \text{Total marks} / 600 * 100$

Include constructor and display method in each class. Write a driver program to test your classes.

Q4. Develop a class **Post** that has following attributes

**Name:** a string

**To :** a string that holds the receiver's address

**StampCost:** a float that holds the value of postal stamp required

The class should include following:

- A constructor that initializes StampCost to 1\$ and To to empty address
- **Read():** a method that reads data member's values from user
- **Print():** a method that displays data member's values on screen
- **TotalCost():** a method that returns stampCost value

Develop another class **RegisteredPost** that inherits from **Post** class and has following additional attributes:

**Weight:** a float that holds the weight of post

**RegistrationCost:** a float that holds registration charges

Also include following in the class.

- A constructor that initializes weight to 20 gms and RegistrationCost to 10\$
- **Read():** a method that reads data member's values from user
- **Print():** a method that displays data member's values on screen
- **TotalCost():** a method that returns stampCost+RegistrationCost.

Q5. Develop a class **InsuredRegisteredPost** that inherits from **RegisteredPost** class and has additional attribute **AmtInsured** to hold the insured value of post. The class should include following:

- A constructor that initializes AmtInsured to 20\$
- **Read():** a method that reads data member's values from user
- **Print():** a method that displays data member's values on screen
- **TotalCost():** a method that returns StampCost+RegistrationCost+AmtInsured.

Q6. Use following driver program to test your classes

```
#include <iostream>
using namespace std;
int main()
{ InsuredRegisteredPost envelope;
  envelope.Read();
  cout<<"Post Details..";
  envelope.Print();
  return 0;
}
```

Q7. Define classes to represent following products available in a shopping mall.

- **Utility Items** having attributes **name**, **product id** and **discount**
- **Food Items** having attributes **name**, **product id** and **date of expiry**

Also define a method to show data in each class.

Write efficient code for handling constructors and methods using Inheritance.

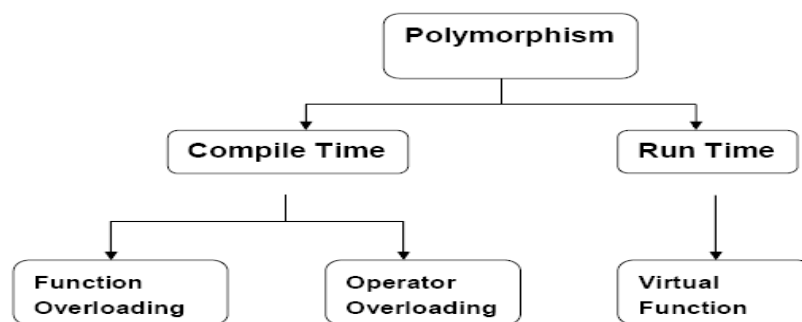
## Lab Session 05

### OBJECT

#### *Studying Polymorphism in Object Oriented Programming*

### THEORY

In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning to a particular function or "operator" in different contexts. Polymorphism is supported by C++ both at compile time and at run time. Compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions.



**Figure 5.1 Types of Polymorphism**

### Static Binding and Dynamic Binding

Binding means the actual time when the code for a given function is attached or bound to the function call

| Static Binding   | Dynamic Binding                              |
|--|--|
| Actual code will be attached during runtime/execution time | Actual code is attached during compile time. |
| Implementing function overloading                          | Implementing virtual functions               |

### Function Overloading.

The term overloading means 'Same thing for different purposes'. Overloading of functions involves defining distinct functions which share the same name, each of which has a unique signature. Function overloading is appropriate for:

- Defining functions which essentially do the same thing but operate on different data types.
- Providing alternate interfaces to the same function.

Using Function Overloading we can perform variety of different tasks, for instance consider a function, `GetTime`, which returns in its parameter(s) the current time of the day, and suppose that we require two variants of this function: one which returns the time as seconds from midnight, and one which returns the time as hours, minutes, and seconds. Given that these two functions serve the same purpose, there is no reason for them to have different names. C++ allows functions to be overloaded, that is, the same function to have more than one definition:

```
long GetTime (void); // seconds from midnight
void GetTime (int &hours, int &minutes, int &seconds);
```

When `GetTime` is called, the compiler compares the number and type of arguments in the call against the definitions of `GetTime` and chooses the one that matches the call. For example:

```
int h, m, s;
long t = GetTime(); // matches GetTime(void)
GetTime(h, m, s); // matches GetTime(int&, int&, int&);
```

To avoid ambiguity, each definition of an overloaded function must have a unique signature. Member functions of a class may also be overloaded:

```
class Time {
//...
long GetTime (void); // seconds from midnight
void GetTime (int &hours, int &minutes, int &seconds);
};
```

Function overloading is useful for obtaining flavours that are not possible using default arguments alone. Finally before we go to implement, we have to remember some points:

- If the compiler doesn't find exact match of actual and formal parameters then compiler implicitly converts the actual parameters value types to any of overloading function parameter types, if the conversion is possible to have multiple matches, then the compiler will generate an error message.
- We cannot overload functions with same signature but have different return type

## Virtual Functions

A *virtual function* is a member function that is defined within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this

determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. Following example illustrate this:

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void vfunc() {
        cout << "This is base's vfunc().\n";
    }
};
class derived1 : public base {
public:
    void vfunc() {
        cout << "This is derived1's vfunc().\n";
    }
};
class derived2 : public base {
public:
    void vfunc() {
        cout << "This is derived2's vfunc().\n";
    }
};
int main()
{
    base *p, b;
    derived1 d1;
    derived2 d2;
    // point to base
    p = &b;
    p->vfunc(); // access base's vfunc()
    // point to derived1
    p = &d1;
    p->vfunc(); // access derived1's vfunc()
    // point to derived2
    p = &d2;
    p->vfunc(); // access derived2's vfunc()
    return 0;
}
```

In the above example, base class pointer **p** is assigned the address of **b**, and **vfunc()** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc()** is executed. Next, **p** is set to the address of **d1**, and again **vfunc()** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc()** to be executed. Finally, **p** is assigned the address of **d2**, and **p->vfunc()** causes the version of **vfunc()** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc()** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

## Pure Virtual Functions

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some

situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

A *pure virtual function* is a virtual function that has no definition within the base class and all derived classes must provide their own definition for this function. If the derived class fails to override the pure virtual function, a compile-time error will result.

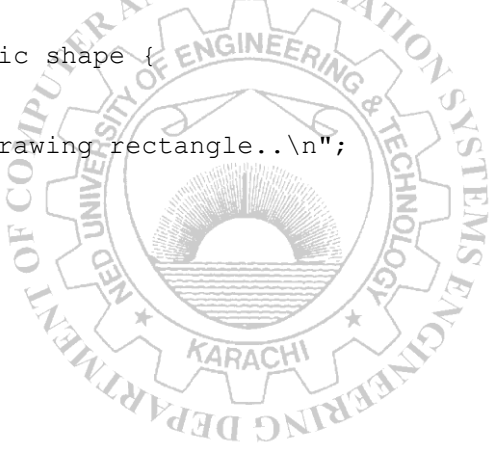
To declare a pure virtual function, use this general form:

virtual *type func-name*(*parameter-list*) = 0;

The following program contains a simple example of a pure virtual function

```
#include <iostream>
using namespace std;
class shape {
public:
    virtual void draw() = 0; // draw() is a pure virtual function
};
class circle : public shape {
public:
    void draw() {
        cout << "drawing circle...\n";
    }
};
class rectangle : public shape {
public:
    void draw() {
        cout << "drawing rectangle...\n";
    }
};

int main()
{
    circle c;
    rectangle r;
    c.draw();
    r.draw();
    return 0;
}
```



Although this example is quite simple, it illustrates how a base class may not be able to meaningfully define a virtual function. In this case, **shape** simply provides the common interface for the derived types to use. There is no reason to define **draw()** inside **shape**. Of course, you can always create a placeholder definition of a virtual function. However, making **draw()** pure also ensures that all derived classes will indeed redefine it to meet their own needs.

## Abstract Classes

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

## Exercises

Q1

```
class abc{
public:
    virtual void func()=0;
};
class xyz:public abc
{ public:
void func()
{ cout << "this is function of xyz";}
};
int main()
{
    abc obj1;
    xyz obj2;
    obj1.func();
    obj2.func();
    return 0;
}
```

Is there any error in the above code? Explain.

Q2. Show the output of the sample programs.

Q3. Define a class **Shape** having an attribute **Area** and a pure virtual function **Calculate\_Area**.

Also include following in this class.

- A constructor that initializes Area to zero.
- A method **Display()** that display value of member variable.

Now drive two classes from **Shape**; **Circle** having attribute radius and **Rectangle** having attributes Length and Breadth. Include following in each class.

- A constructor that takes values of member variables as argument.
- A method **Display()** that override Display() method of Shape class.
- A method **Calculate\_Area()** that calculates area as follows:

Area of Circle=  $\text{PI} \times \text{Radius}^2$

Area of Rectangle=Length\*Breadth

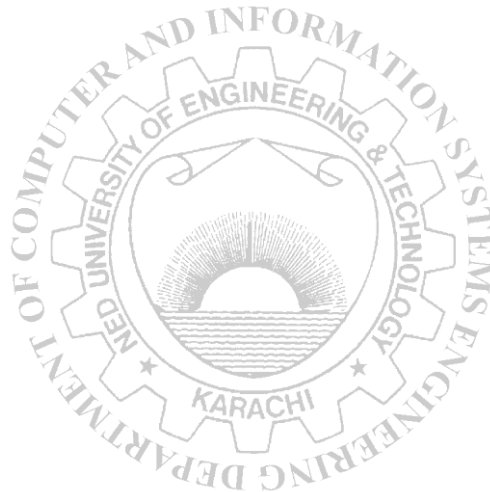
Q4. Use following driver program to test above classes.

```
int main()
{
    Shape *p;
    Circle C1(5);
    Rectangle R1(4,6);
    p=&C1;
    p->Calculate_Area();
    p->Display();
    p=&R1;
    p->Calculate_Area();
    p->Display();
    return 0;
}
```

Q5 Rewrite the program of Q3 Lab session 04 to use a pointer to student class to access the objects of science and art students.

Q6. Define a class **DateTime** that has member variables to hold day, month, year, hours, minutes and seconds value. The class should include following versions of SetValue() method to set date and time values.

- SetValue(day,month,year)
- SetValue(day, month,year,hours)
- SetValue(day,month,year,hour,minutes)
- SetValue(day, month,year,hour,minutes,seconds)



## Lab Session 06

### OBJECT

#### *Overloading operators in C++*

### THEORY

C++ has ability to provide the operators with a special meaning for a data. The mechanism of giving such special meanings to an operator is known as *operator overloading*. We can overload the entire C++ operator except following.

1. Class member access operator (.)
2. Scope resolution operator (::)
3. Size operator (**sizeof**)
4. Conditional operator (? :)

#### **Why do we need operator overloading?**

Before we proceed further, you should know the reason that why we do operator overloading. Do we really need it? For instance, compare the following syntaxes to perform addition of two objects a and b of a user defined class fraction (assume class fraction has a member function called add() which adds two fraction objects):

**C = a.add(b);**

**C = a+b;**

Which one is easier to use?

Obviously the second one is easy to use and is more meaningful. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types.

If you use the operator without providing the mechanism of how these operators are going to perform with the objects of our class, you will get error message

### Defining operator overloading

Operator overloading is done with the help of a special function, called *operator function*, which defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions.

The general format of member operator function is:

```
return type class name :: operator op(arglist)  
{  
function body // task defined  
}
```

Here is a simple example of operator overloading. This program creates a class called **point**, which stores x and y coordinate values. The program overloads the +, -, and ++ operators relative to this class.



```

#include <iostream>
using namespace std;
class point {
    int x_coordinate, y_coordinate;
public:
    point() {}
    point(int x, int y) {
        x_coordinate = x;
        y_coordinate = y;
    }
    void show() {
        cout << x_coordinate << " , ";
        cout << y_coordinate << "\n";
    }
    point operator+(point op2);
    point operator++();
};
// Overload + for point.
point point::operator+(point op2)
{
    point temp;
    temp.x_coordinate = op2.x_coordinate + x_coordinate;
    temp.y_coordinate = op2.y_coordinate + y_coordinate;
    return temp;
}
// Overload prefix ++ for point.
point point::operator++()
{
    x_coordinate++;
    y_coordinate++;
    return *this;
}
int main()
{
    point ob1(10, 20), ob2(5, 30);
    ob1.show(); // displays 10 20
    ob2.show(); // displays 5 30
    ob1 = ob1 + ob2;
    ob1.show(); // displays 15 50
    ++ob1;
    ob1.show();
    return 0;
}

```

## Creating Prefix and Postfix Forms of the Increment and decrement Operators

Standard C++ allows you to explicitly create separate prefix and postfix versions of the increment or decrement operators. To accomplish this, you must define two versions of the **operator++()** function. One is defined as shown in the foregoing program. The other is declared like this:

```
point operator++(int x);
```

If the ++ precedes its operand, the **operator++()** function is called. If the ++ follows its operand, the **operator++(int x)** is called and **x** has the value zero.

## Operator Overloading Using Friend Function

You can also overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

## Overloading Stream Insertion(<<) and Extraction (>>) operators

To overload insertion and extraction operators, Operator function must be a *nonmember* function of the class. Here is the general syntax.

### Function Prototype ( to be include in definition of class)

```
friend ostream &operator<<( ostream &, const classname& );  
friend istream &operator>>( istream &, const classname& );
```

### Function Definition

```
ostream &operator<<( ostream& oobject, const classname& object )  
{  
    // local declaration if any  
    //output member of object  
    //oobject << ...  
    return oobject;  
}  
istream &operator>>( istream& isobject, const classname& object )  
{  
    // local declaration if any  
    // read data into object  
    //isobject >> ...  
    return isobject;  
}
```

## Limitation on operator overloading

Although C++ allows us to overload operators, it also imposes restrictions to ensure that operator overloading serves its purpose to enhance the programming language itself, without compromising the existing entity. The followings are the restrictions:

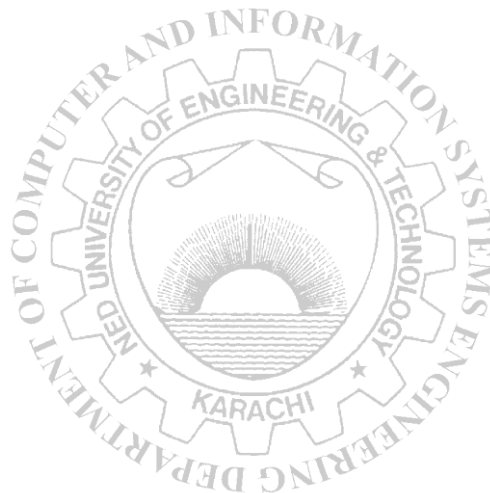
- ✓ Cannot change the original behavior of the operator with built in data types.
- ✓ Cannot create new operator
- ✓ Operators =, [], () and > can only be defined as members of a class and not as global functions
- ✓ The arity or number of operands for an operator may not be changed. For example, addition, +, may not be defined to take other than two arguments regardless of data type.
- ✓ The precedence of the operator is preserved, i.e., does not change, with overloading

## Exercises

Q1. Overload  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $++$ ,  $--$  (both postfix and prefix),  $>$ ,  $<$ ,  $==$  operators for the class **Complex\_No.**(you have defined in lab session 02). Comparison should be done on the basis of magnitude. Write a driver program to test your class.

Q2. Define a class **Matrix** to represent 2x2 matrix. Overload operators for addition and subtraction of two matrices using friend function. Write a driver program to test your class.

Q3. Overload  $[ ]$ , insertion ( $<<$ ) and extraction ( $>>$ ) operators for class **vector** that represents 2D vector. Write a driver program to test your class.



## Lab Session 07

### OBJECT

#### *Studying Aggregation in Object Oriented Programming*

### THEORY

Aggregation is a relationship between two classes that is best described as a "has-a" and "whole/part" relationship. The aggregate class contains a reference to another class and is said to have ownership of that class. Each class referenced is considered to be *part-of* the aggregate class. For example; a computer has components keyboard monitor, mouse and processor box. The relationship between computer and its component is aggregation.

### Containment vs. Containership

Containment is a composition hierarchy in which an object is composed of other object in a fixed relation. The aggregate object cannot exist without its components, which will probably be of a fixed and stable number or at least will vary within a fixed set of possibilities.

Container class contains objects of other classes but the existence of container is independent of whether it actually contains any objects at a particular time, and contained objects will probably be a dynamic or possibly heterogeneous collection

Aggregation can be fixed, variable or recursive:

- **Fixed:** The particular number and type of the component part are predefined. For example, a car has one engine, four wheels, and one steering wheel.
- **Variable:** The number of levels of aggregation is fixed, but the number of parts may vary. For example, rooms in a building.
- **Recursive:** The object contains components of its own type. For example, an object contains a pointer of its own type, allowing it to send messages to other objects of the same class.

### Sample program

```
#include <iostream>
using namespace std;
class CAccessories
{
public:
    int mouse;
    int keyboard;
};
class CCPU {
public:
    int cd;
    int floppy;
```

```
    int proc_speed;
};

class CMonitor {
public:
    char make[80];
    float size;
};

// implementation of the CComputer class.

class CComputer {
public:
    void print(void);
    void get(void);
    CComputer(){};

private:
    CAccessories access;
    CCPU cpu;
    CMonitor monitor;
};

void CComputer::get()
{
    cout << "Enter Information of CPU:\n ";
    cout << "Enter Processor speed: ";
    cin >> this->cpu.proc_speed;
    cout << "Is floppy present (1 for yes, 0 for no) ";
    cin >> this->cpu.floppy; cout << "Is CD present (1 for yes, 0 for no) ";
    cin >> this->cpu.cd;
    cout << endl << "Enter Information for Monitor: \n" ;
    cout << "Enter the \"Make\" of Monitor: ";
    cin >> this->monitor.make; cout << "Enter Monitor size: ";
    cin >> this->monitor.size;
    cout << endl << "Enter Information for the Peripherals: \n" ;
    cout << "Enter Number of buttons of keyboard: ";
    cin >> this->access.keyboard; cout << "Enter Number of buttons of
mouse: ";
    cin >> this->access.mouse;
}

void CComputer::print()
{
    cout << endl << endl << "CPU:" << endl;
    cout << "Processor Speed: " << this->cpu.proc_speed;
    if(this->cpu.floppy==0)
        cout << endl << "Floppy not installed";
    else cout << endl << "Floppy Installed ";
    if(this->cpu.cd==0)
        cout << endl << "CD not installed" ;
    else
        cout << endl << "CD Installed";
    cout << endl << endl << "Monitor: " << endl;
    cout << "\"Make\" of Monitor: " << this->monitor.make << endl;
    cout << "Monitor size: " << this->monitor.size << endl;
    cout << endl << endl << "Peripherals: " << endl;
    cout << "umber of buttons of keyboard: " << this->access.keyboard;
    cout << endl << "Number of buttons of mouse: " << this->access.mouse;
}
```

```
// Defines the entry point for the console application.

int main()
{
    CComputer comp;
    comp.get();
    comp.print();
    cout << endl;
    return 0;
}
```

## Exercises

Q1. Relationship of book with its pages can be modeled as containership. Comment.

---

---

---

Q2. Define a class **Priority\_Queue** that represents a bounded priority queue of nodes. To represent a node you must define a class **Node** having member variables Data (description of node,type string) and a Priority\_No.

In Priority\_Queue class, provide following two methods.

**Enqueue:** that inserts a node in the queue at appropriate position, based on its **Priority\_No**, if queue is not full.

**Dequeue:** that displays and removes a node from front of the queue, if queue is not empty.

Q3. Create a class **Employee**, the member data should include **Employee number** (type integer), **Name of employee** [to be allocated dynamically] and **Salary** (comprising of three float variables; **Gross pay**, **Deductions** and **Net pay**). Also include date of joining of employee. The date should comprise of three integers representing **day**, **month** and **year**. You need to define separate classes for **Date** and **Salary**. Include necessary constructors, destructors and member functions to enter and display the data in each class.

Q4. Define a class **Database** that uses a dynamic array of **Employee**'s objects to stores the record of employees. Also provide following methods in Database class.

**Add:** To add a new employee

**Delete:** To remove an employee's record

**Search:** To search a record by employee number.

Q5. Write a driver program that creates an object of **Database** class and calls its functions according to user requirement.

## Lab Session 08

### OBJECT

#### *Handling Exceptions in C++*

### THEORY

An error result or an unpredictable behavior of your program not caused by the operating system but that occurs in your program is called an exception. The ability to deal with a program's eventual abnormal behavior is called exception handling. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

C++ provides three keywords to handle an exception: try, catch and throw

1. **Trying the normal flow:** To deal with the expected behavior of a program, use the **try** keyword as in the following syntax:

**try** {*Behavior*}

The **try** keyword lets the compiler know that you are anticipating an abnormal behavior and will try to deal with it. The actual behavior that needs to be evaluated is included inside the try block.

2. **Catching Errors:** During the flow of the program as part of the **try** section, if an abnormal behavior occurs, instead of letting the program crash or instead of letting the compiler send the error to the operating system, you can transfer the flow of the program to another section that can deal with it. The syntax used by this section is:

**catch**(*Argument*) {*What To Do*}

This section always follows the **try** section and there must not be any code between the **try**'s closing bracket and the **catch** section. The **catch** behaves a little like a function that defines the behavior of program in case if exception occurs. It uses an argument that is passed by the previous try section. The argument can be a regular variable or a class. If there is no argument to pass, the **catch** must at least take a three-period argument as in **catch(...)**.

The general syntax would be:

```
try {  
    // Try the program flow  
}  
catch(Argument)  
{  
    // Catch the exception  
}
```

3. **Throwing an error:** The general form of the **throw** statement is shown here:

**throw** *exception*;

**throw** generates the exception specified by *exception*. If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly).

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error.

The Standard C++ library provides standard exception classes. Generally it is easier and faster to start with a standard exception class than to try to define your own. If the standard class does not do exactly what you need, you can derive from it.

All standard exception classes derive ultimately from the class **exception**, defined in the header `<exception>`. The two main derived classes are **logic\_error** and **runtime\_error**, which are found in `<stdexcept>` (which itself includes `<exception>`). The class **logic\_error** represents errors in programming logic, such as passing an invalid argument. Runtime errors are those that occur as the result of unforeseen forces such as hardware failure or memory exhaustion. Both **runtime\_error** and **logic\_error** provide a constructor that takes a **std::string** argument so that you can store a message in the exception object and extract it later with **exception::what()**.

**Note:** **std::exception** does not provide a constructor that takes a **std::string** argument.

| Exception classes derived from <b>logic_error</b> |   |
|---|---|
| <b>domain_error</b>                               | Reports violations of a precondition.   |
| <b>invalid_argument</b>                           | Indicates an invalid argument to the function from which it is thrown.  |
| <b>length_error</b>                               | Indicates an attempt to produce an object whose length is greater than or equal to <b>npos</b> (the largest representable value of context size type, usually <b>std::size_t</b> ). |
| <b>out_of_range</b>                               | Reports an out-of-range argument.   |
| <b>bad_cast</b>                                   | Thrown for executing an invalid <b>dynamic_cast</b> expression in runtime type identification.  |
| <b>bad_typeid</b>                                 | Reports a null pointer <b>p</b> in an expression <b>typeid(*p)</b> .  |

Table 8.1

| Exception classes derived from <b>runtime_error</b> |  |
|---|--|
| <b>range_error</b>                                  | Reports violation of a postcondition.  |
| <b>overflow_error</b>                               | Reports an arithmetic overflow.        |
| <b>bad_alloc</b>                                    | Reports a failure to allocate storage. |

Table 8.2

Here is a simple example that shows the way C++ exception handling operates.

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

int main()
```

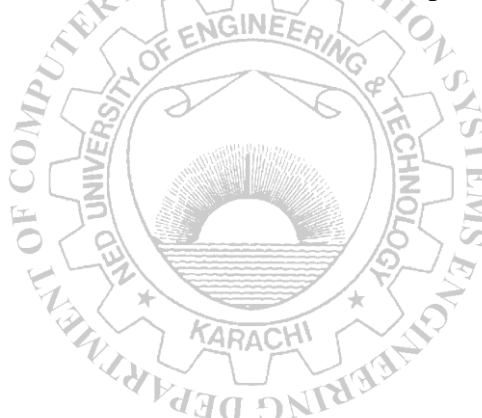


```
{
    int StudentAge;
    cout << "Student Age: ";
    cin >> StudentAge;
    try {
        if(StudentAge < 0)
            throw logic_error("Invalid age");
        cout << "\nStudent Age: " << StudentAge << "\n\n";
    }
    catch(logic_error& e)
    {
        cout << e.what() << endl;
    }
    return 0;
}
```

## Catching Multiple Exceptions

The exceptions as we have seen so far dealt with a single exception in a program. Most of the time, a typical program will **throw** different types of errors. The C++ language allows you to include different **catch** blocks. Each **catch** block can face a specific error. The syntax used is:

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}...
catch (typeN arg) {
// catch block
}
```



The compiler would proceed in a top-down as follows:

1. Following the normal flow control of the program, the compiler enters the **try** block.
2. If no exception occurs in the **try** block, the rest of the **try** block is executed. If an exception occurs in the **try** block, the **try** displays a **throw** that specifies the type of error that happened.
  - a. The compiler gets out of the **try** block and examines the first **catch**
  - b. If the first catch doesn't match the **thrown** error, the compiler proceeds with the next **catch**. This continues until the compiler finds a **catch** that matches the **thrown** error.
  - c. If one of the catches matches the thrown error, its body executes. If no **catch** matches the thrown error, the compiler has two alternatives. If there is no catch that matches the error (which means that you didn't provide a matching catch), the compiler hands the program flow to the operating system (which

calls the **terminate()** function). Another alternative is to include a **catch** whose argument is three periods: **catch(...)**. The **catch(...)** is used if no other **catch**, provided there was another, matches the thrown error. The **catch(...)**, if included as part of a **catch** clause, must always be the last **catch**, unless it is the only **catch** of the clause.

## Creating Your Own Exception Classes

- Programmers can create exception classes to handle exceptions not covered by C++'s exception classes.
- C++ uses the same mechanism to process the exceptions you define as for built-in exceptions
- You must throw your own exceptions using the throw statement
- Any class can be an exception class
- You can also derive your exception class from an standard exception class.

The following example demonstrates the creation and usage of a user defined exception class. The program prompts the user for a positive number. If a negative number is entered, an object of the class **MyException** is created that describes the error. Thus, **MyException** encapsulates information about the error. This information is then used by the exception handler

```
#include <iostream>
#include <cstring>
using namespace std;
class MyException {
public:
    char str_what[80];
    int what;
    MyException() { *str_what = 0; what = 0; }
    MyException(char *s, int e) {
        strcpy(str_what, s);
        what = e;
    }
};

int main()
{
    int i;
    try {
        cout << "Enter a positive number: ";
        cin >> i;
        if(i<0)
            throw MyException("Not Positive", i);
    }
    catch (MyException e) { // catch an error
        cout << e.str_what << ": ";
        cout << e.what << "\n";
    }
    return 0;
}
```

## Exercise

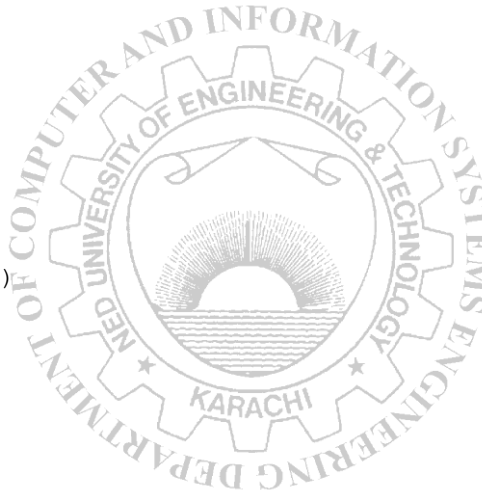
Q1. Define a class **Timer** having attributes **Hour**, **Minute** and **Second**. Include a constructor that initializes all attributes to zero. Also add a method **Tick** that increments **Second** by one, if **Second** reaches 60, increment **Minute** by one, if **Minute** reaches 60, increment **Hour** by one and if **Hour** reaches 24, throws an exception '**Timer overflow**'. Write a driver program that creates an object of class **Timer**, calls **Tick** in a loop and catches the exception.

Q2. Redo Q1 by defining your own exception class to represent 'Timer Overflow Exception'

Q3. Create a class **Stack** to represent stack of integers of fixed size. The member functions Push and Pop must generate exception in case of overflow and underflow respectively. Write a driver program that creates an object of stack and catches the exception.

Q4. Define a class **Date** to represent date in format DD/MM/YYYY. Provide a method **SetDate** that takes date value from user and generates exception if user enters incorrect value. You can define a separate class **DateException** that store messages for possible exceptions. Use following driver program to test your classes.

```
#include <iostream>
using namespace std;
int main()
{
    Date date;
    try {
        date.SetDate();
    }
    catch (DateException e)
    { cout << e.what(); }
    return 0;
}
```



## Lab Session 09

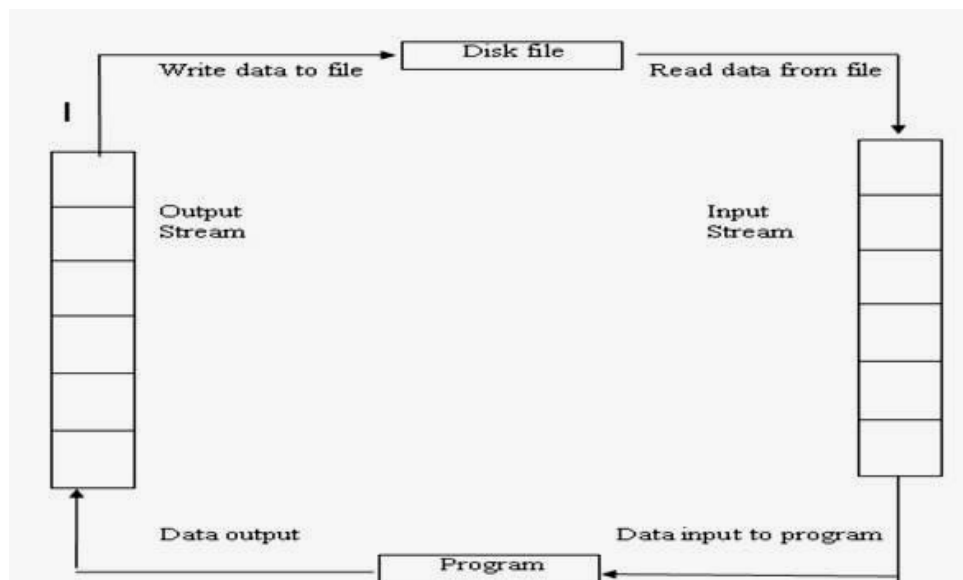
### OBJECT

*Reading and writing files by using C++ IO Stream Library*

### THEORY

A File is a collection of related data stored in a particular area on the disk. Although C++ I/O forms an integrated system, file I/O is sufficiently specialized that it is generally thought of as a special case, subject to its own constraints and quirks.

The file operations make use of streams as an interface between the programs and the files. The stream that supplies data to the program is known as *input stream*. It reads data from a file and hands it over to the program. The stream that receives data from the program is known as the *output stream*. It writes the received data to the file



**Figure 9.1 File I/O**

To perform file I/O, you must first obtain a stream. There are three types of streams: input, output, and input/output. To create an input stream, you must declare the stream to be of class **ifstream**. To create an output stream, you must declare it as class **ofstream**. Streams that would perform both input and output operations must be declared as class **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios**. Another class used by the file system is **filebuf**, which provides low-level facilities to manage a file stream. Usually, you don't use **filebuf** directly, but it is part of the other file classes

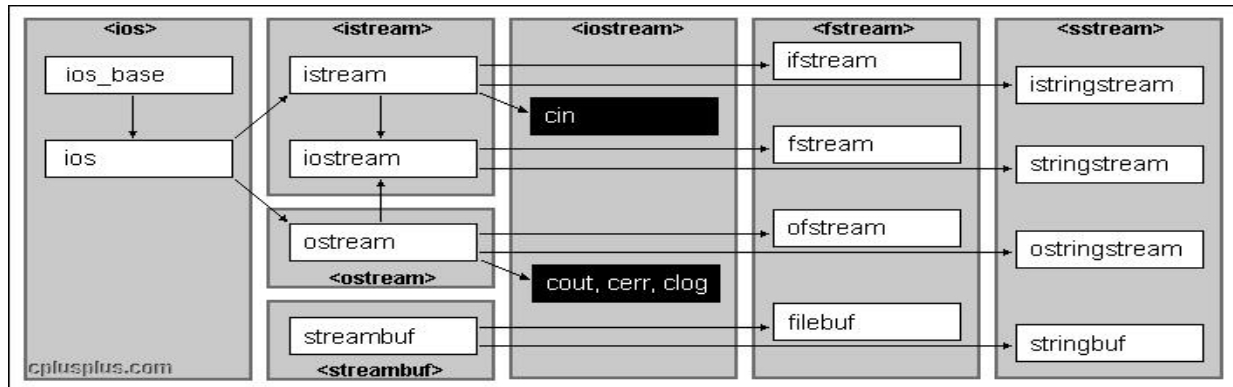


Figure 9.2 Standard I/O stream Library

## Opening and Closing a File

Once a stream has been created, next step is to associate a file with it. Thereafter the file is available (opened) for processing. Opening of a file can be achieved in two ways:

- ❖ Using the **constructor** function of the stream class.
- ❖ Using the function called **open()**.

The first method is preferred when a single file is used with a stream; however, for managing multiple files with the same stream, the second method is preferred.

### Opening a file using constructor

A constructor of a class initializes an object of its class when it (the object) is being created. This involves following steps.

1. Create file stream object to manage the stream using appropriate class, i.e. class **ofstream** is used to create output stream and class **ifstream** to create input stream
2. Initialized the file object with desired file name. For example, following statement opens a file for output, this creates outfile as a ostream object that manages output stream and also opens the file result and attaches it to the output stream outfile.

```
ofstream outfile("result");
```

Similarly, the following statement declares **infile** as an **ifstream** object and attaches it to the file data for reading (input).

```
ifstream infile("data")
```

### Opening a file using open()

Function **open()** can be used to open multiple files that use the same stream object. For example, for processing a set of file sequentially we may create a single stream object and use it to open each file in turn. This function is a member of each of the three stream classes. The prototype for each is shown here:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);
void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

Here, *filename* is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values defined by **openmode**, which is an enumeration defined by **ios**.

| Mode           | Description   |
|----------------|---|
| ios::app       | If FileName is a new file, data is written to it. If FileName already exists and contains data, then it is opened, the compiler goes to the end of the file and adds the new data to it.  |
| ios::ate       | If FileName is a new file, data is written to it and subsequently added to the end of the file. If FileName already exists and contains data, then it is opened and data is written in the current position.  |
| ios::in        | If FileName is a new file, then it gets created fine as an empty file. If FileName already exists, then it is opened and its content is made available for processing   |
| ios::out       | If FileName is a new file, then it gets created fine as an empty file. Once/Since it gets created empty, you can write data to it. If FileName already exists, then it is opened, its content is destroyed, and the file becomes as new. Therefore you can create new data to write to it. Then, if you save the file, which is the main purpose of this mode, the new content is saved it.*This operation is typically used when you want to save a file |
| ios::trunc     | If FileName already exists, its content is destroyed and the file becomes as new  |
| ios::nocreate  | If FileName is a new file, the operation fails because it cannot create a new file. If FileName already exists, then it is opened and its content is made available for processing  |
| ios::noreplace | If FileName is a new file, then it gets created fine. If FileName already exists and you try to open it, this operation would fail because it cannot create a file of the same name in the same location.   |

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

| Class    | Default Mode Parameter |
|----------|------------------------|
| ofstream | ios:: out              |
| ifstream | ios::in                |
| fstream  | ios::in   ios::out     |

Since the *mode* parameter provides default values for each type of stream, the long syntax of open() is seldom used, Therefore, the statement for opening a file will usually look like this:

```
ofstream outfile; // create stream( for output)
outfile.open("DATA1"); //connect stream to file
.....
outfile.close(); //disconnect stream from DATA1

ifstream infile; // create stream( for input)
infile.open("DATA2"); //connect stream to file
```

```
....
infile.close(); //disconnect stream from file
```

## Error handling during file operations

One of the following things may happen when dealing with the file.

1. A file which we are attempting to open for reading does not exist
2. The file name used for new file already exists
3. Attempting an invalid operation, use of an invalid file name
4. There may not be space on the disk for storing more data, etc.

The class **ios** supports several member functions that can be used to read the status recorded in a file stream

| Function | Return value and meaning  |
|----------|---|
| eof()    | Returns true if a file opened for reading has reached the end   |
| fail()   | Returns true in the same cases as bad(), but also in the case that a format error happens   |
| Good     | It is the most generic state flag: it returns false in the same cases in which calling any of the previous function would return true |
| bad()    | Returns true if a reading or writing operation fails  |

## Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file. For example, this program creates a short inventory file that contains each item's name and its cost:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream out("INVENTORY"); // output, normal file
    if(!out) {
        cout << "Cannot open INVENTORY file.\n";
        return 1;
    }
    out << "Radios " << 39.95 << endl;
    out << "Toasters " << 19.95 << endl;
    out << "Mixers " << 24.80 << endl;
    out.close();
    return 0;
}
```

Figure 9.3 shows contents of INVENTORY file.

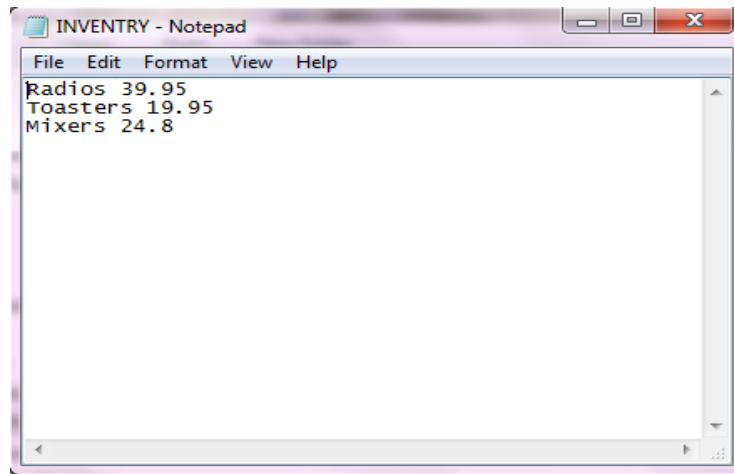


Figure 9.3 Contents of INVENTORY

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream in("INVENTORY"); // input
    if(!in) {
        cout << "Cannot open INVENTORY file.\n";
        return 1;
    }
    char item[20];
    float cost;
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in.close();
    return 0;
}
```

## Object Serialization

Object serialization consists of saving the values that are part of an object, mostly the value gotten from declaring a variable of a class. At the current standard, C++ doesn't inherently support object serialization. To perform this type of operation, you can use a technique known as binary serialization.

### Binary I/O

When performing binary operations on a file, be sure to open it using the **ios::binary** mode specifier. To read and write blocks of binary data is to use C++'s **read ( )** and **write ( )** functions. Their prototypes are:



```
istream &read(char *buf, streamsize num);  
ostream &write(const char *buf, streamsize num);
```

The **read()** function reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*. The **write()** function writes *num* characters to the invoking stream from the buffer pointed to by *buf*. To indicate that you want to save a value as binary, when declaring the **ofstream** variable, specify the **ios** option as **binary**.

Here is an example that stores objects of a class Employee in a file using binary serialization:

```
#include <fstream>  
#include <iostream>  
using namespace std;  
  
class Employee {  
public:  
    char   Name[40];  
    int    Age;  
    double Salary;  
public:  
    Employee (){}  
    Employee(char *n, int age, double sal){  
        strcpy(Name,n);  
        Age =age;  
        Salary=sal;  
    }  
};  
  
int main()  
{  
    Employee e1("Ahmed",40,1000.00);  
    ofstream ofs("employeefile", ios::binary);  
    if(!ofs) {  
        cout << "Cannot open file.\n";  
        return 1;  
    }  
    ofs.write((char *)&e1, sizeof(e1));  
    ofs.close();  
    return 0;  
}
```

Figure 9.4 shows the contents of **employeefile**. The contents are unreadable because the file is written in binary format

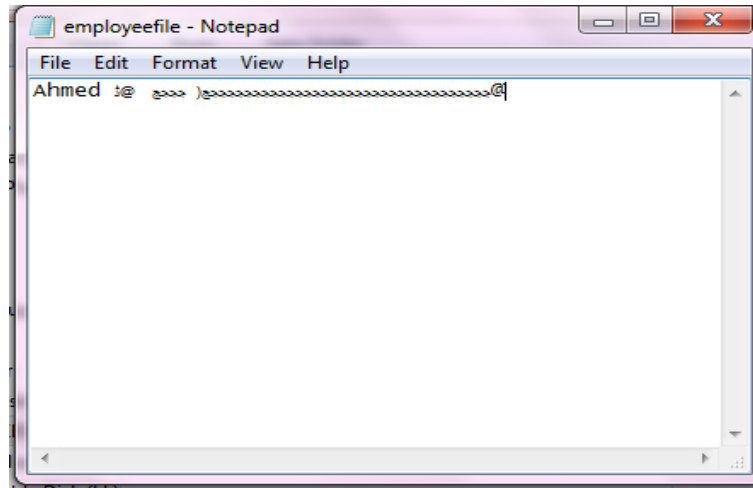


Figure 9.4 Contents of employeefile

### Reading from the Stream

Reading an object saved in binary format is as easy as writing it. To read the value, call the `ifstream::read()` method. Here is an example:

```
#include <iostream>
#include <fstream>
int main()
{
    Employee e2;
    ifstream ifs("employeefile", ios::binary);
    if(!ifs) {
        cout << "Cannot open file.\n";
        return 1;
    }
    ifs.read((char *)&e2, sizeof(e2));
    cout << "Employee Information\n";
    cout << " Name: " << e2.Name << endl;
    cout << "Age   " << e2.Age << endl;
    cout << "Salary " << e2.Salary << endl;
    ifs.close();
    return 0;
}
```

### Random Access

In C++'s I/O system, you perform random access by using the `seekg( )` and `seekp( )` functions. Their most common forms are

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

Here, **off\_type** is an integer type defined by **ios** that is capable of containing the largest valid value that *offset* can have. **seekdir** is an enumeration defined by **ios** that determines how the seek will take place. The C++ I/O system manages two pointers associated with a file. One is the *get*

*pointer*, which specifies where in the file the next input operation will occur. The other is the *put pointer*, which specifies where in the file the next output operation will occur. Each time an input or output operation takes place, the appropriate pointer is automatically sequentially advanced. However, using the `seekg()` and `seekp()` functions allows you to access the file in a nonsequential fashion. The `seekg()` function moves the associated file's current get pointer *offset* number of characters from the specified *origin*, which must be one of these three values:

`ios::beg` Beginning-of-file

`ios::cur` Current location

`ios::end` End-of-file

The `seekp()` function moves the associated file's current put pointer *offset* number of characters from the specified *origin*, which must be one of the values just shown. Generally, random-access I/O should be performed only on those files opened for binary operations. The character translations that may occur on text files could cause a position request to be out of sync with the actual contents of the file.

The following program uses `seekg()`. It displays the contents of a file beginning with the location specified by user. You need to create a text file `Myfile.txt` and save it on D: drive before running this program

```
#include <cstdlib>
using namespace std;
int main()
{
    int i;
    char ch;
    ifstream in("D:\\Myfile.txt", ios::in | ios::binary);
    if(!in) {
        cout << "Cannot open file.";
        return 1;
    }
    cout << "Enter file location: ";
    cin >> i;
    in.seekg(i, ios::beg);
    while(in.get(ch))
        cout << ch;
    return 0;
}
```

## Exercises

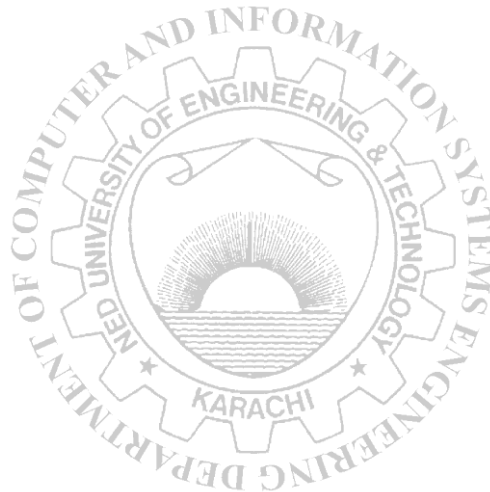
Q1. Write a program that reads strings entered at the keyboard and writes them to disk. The program should stop when the user enters an exclamation point.

Q2. Write a program that allows user to change a specific character in a file. Take filename, number of the character in the file you want to change, and the new character as user input.

Q3. Write a program that creates an object of class `Priority_Queue` (defined in Lab 7). Fill it by taking inputs from user and store it in a file.

Q4. Redefine the class **Database** (you have defined in Lab 7) to include **Save** and **Retrieve**

methods. **Save** method creates a file named 'db.txt' and stores the records of employees in the file. **Retrieve** method opens the same file 'db.txt' and retrieves the data from file. Also write a driver program to test your class.



## Lab Session 10

### OBJECT

#### *Implementing Function Templates and Class Templates in C++*

### THEORY

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one. Templates are very useful when implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type

C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.

There are two kinds of templates: *function templates* and *class templates*

Use function templates to write generic functions that can be used with arbitrary types. For example, one can write searching and sorting routines which can be used with any arbitrary type. The Standard Template Library generic algorithms have been implemented as function templates, and the containers have been implemented as class templates

### Function templates

A *function template* behaves like a function except that the template can have arguments of many different types. In other words; a function template represents a family of functions. The general format of a function template is as follows:

```
template <class T>
return type functionname(argument of type T )
{
// body of function with Type T
}
```

The following example declares a function template `max(x, y)` which returns either `x` or `y`, whichever is larger

```
template <class myType>
myType max (myType a, myType b) {
    return (a>b?a:b);}
```

The above definition of function `max()` can work with different kinds of data types. A function template does not occupy space in memory. The actual definition of a function template is generated when the function is called with a specific data type. The function template does not save memory.

```
#include <iostream>
int main()
{
    // This will call max <int> (by argument deduction)
    std::cout << max(3, 7) << std::endl;
    // This will call max<double> (by argument deduction)
    std::cout << max(3.0, 7.0) << std::endl;
}
```

```
    return 0;
}
```

In the above program, the template argument T is automatically deduced by the compiler to be int and double, respectively. This function template can be instantiated with any copy-constructible type for which the expression (y < x) is valid. For user-defined types, this implies that the less-than operator must be overloaded.

## Class templates

A class template provides a specification for generating classes based on parameters. Class templates are commonly used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments. The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as vector.

### *Implementing a class template*

A class template definition looks like a regular class definition, except it is prefixed by the keyword template. For example, here is the definition of a class template for a Stack.

```
const int SIZE = 10;
template <class StackType>
class stack {
    StackType stck[SIZE]; // holds the stack
    int tos; // index of top-of-stack
public:
    stack() { tos = 0; } // initialize stack
    void push(StackType ob); // push object on stack
    StackType pop(); // pop object from stack
};
```

StackType is a type parameter and it can be any type. For example, Stack<Token>, where Token is a user defined class. StackType does not have to be a class type as implied by the keyword class. For example, Stack<int> is a valid instantiation, even though int is not a "class".

Implementing template member functions is somewhat different compared to the regular class member functions. The declarations and definitions of the class template member functions should all be in the same header file.

While implementing class template member functions, the definitions are prefixed by the keyword template. Here is the complete implementation of class template Stack:

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;
const int SIZE = 10;
// Create a generic stack class
template <class StackType>
```

```

class stack {
    StackType stck[SIZE]; // holds the stack
    int tos; // index of top-of-stack
public:
    stack() { tos = 0; } // initialize stack
    void push(StackType ob); // push object on stack
    StackType pop(); // pop object from stack
};
// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }

    stck[tos] = ob;
    tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
    if(tos==0) {
        cout << "Stack is empty.\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stck[tos];
}

```

### Using a class template

Using a class template is easy. Create the required classes by plugging in the actual type for the type parameters. This process is commonly known as "Instantiating a class". Here is a sample driver class that uses the Stack class template.

```

#include <iostream>
#include "stack.h"
using namespace std ;
int main()
{
    // Demonstrate character stacks.
    stack<char> s1, s2; // create two character stacks
    int i;
    s1.push('a');
    s2.push('x');
    s1.push('b');
    s2.push('y');
    s1.push('c');
    s2.push('z');
    for(i=0; i<3; i++)
        cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++)
        cout << "Pop s2: " << s2.pop() << "\n";
    // demonstrate double stacks
    stack<double> ds1, ds2; // create two double stacks
    ds1.push(1.1);
    ds2.push(2.2);
    ds1.push(3.3);
}

```

```
    ds2.push(4.4);
    ds1.push(5.5);
    ds2.push(6.6);
    for(i=0; i<3; i++)
        cout << "Pop ds1: " << ds1.pop() << "\n";
    for(i=0; i<3; i++)
        cout << "Pop ds2: " << ds2.pop() << "\n";
    return 0;
}
```

In the above example we defined a class template Stack. In the driver program we instantiated a Stack of characters and a Stack of double. Once the template classes are instantiated you can instantiate objects of that type

We can use more than one generic data type in a class template. They are decided as a comma separated list within the template specification as shown below

### Syntax

```
Template<class T1, class T2 ...>
class classname
{
    ..... (Body of the class)
    .....};
```

### Exercises

- Q1. What is the output of the above program?
- Q2. Define a function template to implement bubble sort algorithm.
- Q3. Define a function template that finds maximum value in an array.
- Q4. Define a function template Power(x,y) that calculates  $x^y$
- Q5. Define template class **Matrix** that represent a 2x2 matrix. Include following methods
- **Determinant**: that calculates and displays the determinant of matrix
  - **Display**: that displays matrix in appropriate format.

Also overload + and – operator for matrix class.

Q6. Write a program that define two matrices of type float using template class Matrix, calculate their determinants and perform addition and subtraction on them.

Q7. Write a program that defines two matrices of type **Complex\_No** using template class Matrix and perform addition and subtraction on them.[use Complex\_No class, defined in lab 06]



