

## C++

Lesson 00:

IGATE is now a part of Capgemini

People matter, results count.



Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

## Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
14-Sep-2009			Liji Shynu	Added topics Exception Handling, Templates, Input Output in C++ and Good Programming Practices

January 28, 2016 | Proprietary and Confidential | -1-



## Course Goals and Non Goals

### ➤ Course Goals

- To understand object oriented concepts and the basic features of C++ like: Classes, Inheritance, Polymorphism, Operator overloading etc.



### ➤ Course Non Goals

- Advanced concepts like STL and RTI are not covered in this course.

## Pre-requisites

- Fair knowledge of C programming language.



Intended Audience

- Developers



January 28, 2016 Proprietary and Confidential + 5 -

 Capgemini  
CONSULTING. IT AND BUSINESS SERVICES

## Day Wise Schedule

- **Day 1**
  - Lesson 1: Object Oriented Features
  - Lesson 2: Features of C++
  - Lesson 3: Functions
  - Lesson 4: Classes
  - Lesson 5: Constructors and Destructors
- **Day 2**
  - Lesson 6: Friend Function and Class
  - Lesson 7: Operator Overloading
- **Day 3**
  - Lesson 7 (Contd.): Operator Overloading
  - Lesson 8: Inheritance

## Day Wise Schedule

### ➤ Day 4

- Lesson 9: Polymorphism
- Lesson 10: String Class
- Lesson 11: Exception Handling

### ➤ Day 5

- Lesson 12: Templates
- Lesson 13: I/O in C++
- Lesson 14 : Best Coding Practices



## Table of Contents

- **Lesson 1: Object Oriented Features**
  - 1.1. Introduction
  - 1.2. Limitations of Structured Programming
  - 1.3. Object Oriented Approach
  - 1.4. Basic Terminologies
  - 1.5. Object-oriented Programming Process
- **Lesson 2: Features of C++**
  - 2.1. Features Inherited from C
  - 2.2. Features of C++
  - 2.3. C++ I/O Library
  - 2.4. Memory Allocation
  - 2.5. Memory Management Operator
  - 2.6. Manipulators
  - 2.7. Enumerated Data Types



## Table of Contents (contd..)

### ➤ Lesson 3: Functions

- 3.1. Inline functions
- 3.2. Default Function Arguments
- 3.3. Function Overloading
- 3.4. Reference Variables
- 3.5. Restrictions on Reference Variables

### ➤ Lesson 4: Classes

- 4.1. Classes
- 4.2. Categories of Class Member Functions
- 4.3. this Pointer
- 4.4. Function Chaining
- 4.5. Static Data Members
- 4.6. Static Member Function
- 4.7. Type Qualifiers



## Table of Contents (contd..)

### ➤ Lesson 5: Constructors and Destructors

- 5.1. Constructors
- 5.2. Constructor Overloading
- 5.3. Copy Constructor
- 5.4. Constructor Initialization Section
- 5.5. Destructor

### ➤ Lesson 6: Friend Function and Class

- 6.1. Friend Function
- 6.2. Friend Class
- 6.3. Namespace
- 6.4. Using keyword
- 6.5. std namespace



## Table of Contents (contd..)

- **Lesson 7: Operator Overloading**
  - 7.1. Operator Overloading
  - 7.2. Operator Functions
  - 7.3. Unary Operator Overloading
  - 7.4. Binary Operator Overloading
  - 7.5. Restrictions on Operator Overloading
- **Lesson 8: Inheritance**
  - 8.1. Inheritance Concepts
  - 8.2. Inheritance Types
  - 8.3. Access Specifiers
  - 8.4. Multilevel Inheritance
  - 8.5. Multiple Inheritance
  - 8.6. Constructors in Derived Class



## Table of Contents (contd..)

- 8.7. Hybrid Inheritance
- 8.8. Virtual Base Class
- **Lesson 9: Polymorphism**
  - 9.1. Polymorphism Types
  - 9.2. Pointers to objects
  - 9.3. Pointers to derived Class
  - 9.4. Virtual Functions
  - 9.5. Virtual Destructor Functions
  - 9.6. Pure virtual Functions
  - 9.7. Abstract Base Class
- **Lesson 10: String Class**
  - 10.1. String Class



## Table of Contents (contd..)

- 10.2. String Class – const Member functions
- 10.3. Operators Defined for String
- 10.4. Member Functions

### ➤ **Lesson 11: Exception Handling**

- 11.1. Exception handling
- 11.2. How to handle an Exception
- 11.3. Uncaught Exceptions
- 11.4. Exception Specification

### ➤ **Lesson 12: Templates**

- 12.1. Template
- 12.2. Template Types
- 12.3. Class Templates



## Table of Contents (contd..)

### ➤ Lesson 13: Input Output in C++

- 13.1. Input Output in C++
- 13.2. File Handling
- 13.3. Some Useful Functions

### ➤ Lesson 14: Good Programming Practices

- 14.1. Variable Declaration
- 14.2. Class Definition
- 14.3. General



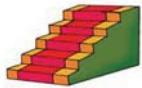
## References

- Programming with C++ by John R. Hubbard
- Let us C++ by Yashwant Kanetkar



## Next Step Courses (if applicable)

➤ VC++



## Other Parallel Technology Areas

- Java
- PERL

# C++

## Lesson 1: Object-Oriented Features

January 26, 2016 | Proprietary and Confidential | - 1 -



## Lesson Objectives

➤ In this lesson, you will learn about:

- What is Object Oriented Programming?
- Need for Object Oriented Programming
- Object Oriented Programming
  - Approach
  - Terminologies
  - Process



January 26, 2016 | Proprietary and Confidential | + 3 -

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Lesson Objectives

This lesson is an introduction towards the Object-oriented concepts and approaches.

The lesson contents are:

#### Lesson 1: Object Oriented Features

- 1.1. Introduction
- 1.2. Limitations of Structured Programming
- 1.3. Object Oriented Approach
- 1.4. Basic Terminologies
- 1.5. Object-oriented Programming Process

L1: Object Oriented Programming: Features

## Introduction

- Bjarne Stroustrup of AT&T Bell Laboratories developed C++ in the early 1980's in Murray Hill, New Jersey.
- His inspiration came from the language Simula67, which supported the concept of a class.
- AT&T made many improvements to this initial language before releasing it commercially for the first time in 1985.

January 26, 2016 | Proprietary and Confidential | + 5 +

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Introduction to C++:

Bjarne Stroustrup of AT&T Bell Laboratories developed C++ in the early 1980's in Murray Hill, New Jersey. He created C++ while adding features to C to support efficient event-driven simulation programs. His inspiration came from the language Simula67, which supported the concept of a class. AT&T made many improvements to this initial language before releasing it commercially for the first time in 1985. Since then C++ has continued to evolve with AT&T controlling the releases.

1.2: Need for Object Oriented Programming

## structured programming : Problems

➤ Limitations of structured programming are:

- As programs grow larger and compiles, the structured programming approach fails. Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself.
- In a procedural language, the emphasis is on doing things i.e. emphasis is on action. Data is given a second class treatment.
- In procedural languages, we declare global variables; these variables can be accessed by all functions.
- Programmer not familiar with the program may create a function that accidentally corrupts the data.
- Many functions access the same data, hence the way the data is stored becomes critical.
- Data access should be restricted. This protects the data and simplifies maintenance.
- It cannot model real world hence difficult to design.

January 26, 2016 | Proprietary and Confidential | + 8 +

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Need for Object Oriented Programming:

#### Limitations of structured programming:

As programs grow larger and complex, the structured programming approach fails.

Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

In a procedural language, the emphasis is on doing things – read the keyboard, check for errors and so on. i.e. emphasis is on action. Data is given a second-class treatment.

In procedural languages, we declare global variables; these variables can be accessed by all functions. These functions can perform various operations on the data. You may declare local variables, but they cannot be accessed by all functions.

1.3: Object Oriented Features: Approach

## Object-Oriented Approach

- The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object.
- An object's functions, called member functions (methods), typically provide the only way to access its data.

January 26, 2016 | Proprietary and Confidential | -5-

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Principle behind Object Oriented Programming:

The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object.

An object's functions, called member functions (methods), typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will read the item and return the value to you. You cannot access the data directly. The data is hidden, so it is safe from accidental alterations. Data and its functions are said to be encapsulated into a single entity.

If you want to modify the data in an object, you know exactly what functions interact with it; the member functions in the object. No other functions can access the data. This simplifies writing, debugging and maintaining the program.

All object-oriented programming languages have three traits in common: encapsulation, polymorphism and inheritance.

1.3: Object Oriented Features: Approach

## What is Object Oriented Programming?



- An object is like a black box. The internal details are hidden.
- Identifying objects and assigning responsibilities to these objects.
- Objects communicate to other objects by sending messages.
- Messages are received by the methods of an object

January 26, 2016 | Proprietary and Confidential | +6+

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### What is Object Oriented Programming?

In object-oriented programming we can identify objects as needed.

Object-oriented programming groups operations and data into modular units called objects and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.

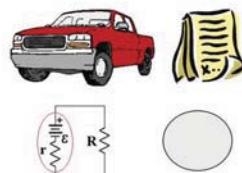
Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects.

1.3: Object Oriented Features: Approach

## What is an Object?

➤ An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well-defined role in the problem domain.

- Tangible things( as a car, printer, ...)
- Roles (as employee, boss, ...)
- Incidents(as flight, overflow, ...)
- Interactions(as contract, sale, ...)
- Specifications(as colour, shape, ...)



January 26, 2016 | Proprietary and Confidential | - 2 -

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### What is an Object?

An object is something tangible, that can be seen or touched, or something that can be eluded to and thought about.

Object-oriented programs consist of hundreds or thousands of objects. These objects communicate with each other by sending messages from one object to another. In a true object oriented program, you have – a coherent group of communicating objects. New objects are created when needed, old ones are deleted, and existing objects carry out operations by sending messages.

Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects. Mechanical device, such as a pocket watch or a piano, embodies both state and behavior

1.3: Object Oriented Features: Approach  
**Why do we Care about Objects?**

- **Modularity** - large software projects can be split up in smaller pieces.
- **Reusability** - Programs can be assembled from pre-written software components.
- **Extensibility** - New software components can be written or developed from existing ones.

1.3: Object Oriented Features: Approach

## Example

➤ Code Snippet

```
#include<iostream.h>
class Person
{
    char name[20];
    int yearOfBirth;
public:
    void displayDetails()
    {
        cout << name << " born in " << yearOfBirth << endl;
    }
    //...
};
```

January 26, 2016 | Proprietary and Confidential | - 9 -

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

Class is a construct used as a blueprint to create instance. They are called as class instances, class objects, instance objects or just the objects. A class defines members that enable these class instances to have state and behavior. Data field members enable the class object to maintain state. Class methods enable a class object's behavior. Example Person class can create an instance of type Person. A class usually represents a noun, such as a person, place or thing.

The above code snippet represents a Person class consisting of two variables for holding the person details and a displayDetails() function. displayDetails() function is used to display the name of a person with year of Birth.

1.3: Object Oriented Features: Approach  
**Parts of an Object**

- An object has two parts:
  - Object = Data + Methods
  - or to say the same thing differently:
- An object has the responsibility to know and the responsibility to do.



January 26, 2016 | Proprietary and Confidential | + 10 +

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

1.4: Object Oriented Features: Basic Terminology

## Abstraction and Encapsulation

- **Encapsulation:**
  - It is the practice of including in an object everything it needs hidden from other objects.
- **Abstraction:**
  - It is the representation of the essential features of an object. They are 'encapsulated' into an abstract data type.
- **Inheritance:**
  - It means that one class inherits the characteristics of another class. This is also called a "is a" relationship:

January 26, 2016 | Proprietary and Confidential | +11+

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

**Encapsulation:**

Encapsulation is a mechanism that binds together code and data, and that keeps both safe from outside interference or misuse. Further, it allows the creation of an object. An object is a logical entity that encapsulates both data and the code that manipulates that data. When you define an object, you are implicitly creating a new data type.

**Abstraction:**

This is the ability to represent a complex problem in simple terms. In the object oriented approach it is seen in the ability to create a high-level class definition that has little or no detail included in it. We are able to "abstract" the problem to a simple class definition and only add detail later in the process.

**Inheritance:**

Inheritance is a process by which one object can acquire the properties of another object. This feature allows you to add refinements of your own to an existing object. For e.g. : A car is a vehicle, An Employee is a Person.



January 28, 2016

Proprietary and Confidential

+ 12 +

**Specialization:**

Seen in class structures, specialization is the ability to make lower level subclasses more detailed than their parent classes. Technically, specialization is defined as the ability of an object to inherit operations and attributes from a super class (parent class) with possible restrictions and additions.

These above two concepts work in conjunction with one and other. Abstraction allows us to define high-level classes that lack minute details of implementation while specialization supports the introduction of those details later in the class design.

**Reuse**

In object-oriented programming we have new tools that allow for an even better level of reusability. By taking advantage of things like polymorphism and inheritance we can create class definitions that are truly universal in their applicability.

1.4: Object Oriented Features: Basic Terminology

## Polymorphism

➤ **Polymorphism:**

- It means “having many forms”.
- It allows different objects to respond to the same message in different ways, the response is specific to the type of the object.

➤ **Example:**

The message `displayDetails()` of the `Person` class should give different results when sent to a `Student` object (For example, the enrolment number).

January 26, 2016 | Proprietary and Confidential | +13+

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Polymorphism:

In literal sense, polymorphism means the quality of having more than one form. In the context of OOP, polymorphism refers to the fact that a single operation can have different behavior in different objects. In other words, different objects react differently to the same message.

Object-Oriented programming languages support polymorphism, which is characterized by the phase “one interface, multiple methods”. Polymorphism is the attribute that allows one interface to be used with a general class of actions.

Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of actions. It is the compiler's job to select the specific action (method) as it applies to each situation.

1.4: Object Oriented Features: Basic Terminology

## Behaviours and Messages

➤ The most important aspect of an object is its behavior (the things it can do). A behavior is initiated by sending a message to the object (usually by calling a method).



January 26, 2016 | Proprietary and Confidential | + 14 +

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Behaviour of an object:

The behaviour of an object is defined by the set of methods that can be applied on it. A running program is a pool of objects where objects are created, destroyed and interacting. This interacting is based on messages, which are sent from one object to another asking the recipient to apply a method on itself.

Sending a message asking an object to apply a method is similar to a procedure call in “traditional” programming languages. However, in object-orientation, there is a view of autonomous objects that communicate with each other by exchanging messages. Objects react when they receive messages by applying methods on themselves. They also may deny the execution of a method, for example if the calling object is not allowed to execute the requested method.

1.5: Object Oriented Programming Process

## Object-Oriented Programming Steps

➤ Following are the two steps in Object Oriented Programming:

- Step 1: Making Classes: Creating, extending or reusing abstract data types.
- Step 2: Making Objects interact: Creating objects from abstract data types and defining their relationships.

January 26, 2016 | Proprietary and Confidential | +15+

 Capgemini  
CONSULTING TECHNOLOGIES OUTSOURCING

### Steps in Object Oriented Programming:

#### Step 1: Making Classes:

A class is the implementation of an abstract data type. It defines attributes and methods, which implement the data structure and operations of the abstract data type, respectively. Instances of classes are called objects. Consequently, classes define properties and behavior of sets of objects.

#### Step 2: Making Objects interact:

Objects need to associate with other objects to enable messages to be sent. Sending a message to an object needs to know the recipient of the message.

## Summary

➤ **In this lesson we learnt:**

- Problems with Structured programming
  - It is complex.
  - Data access cannot be restricted.
  - It cannot model real world problems into programming.
- Object oriented approach helps combine data and the functions into a single unit.(Object = Data + Methods)
- Important features are – abstraction, encapsulation, inheritance and polymorphism



## Review Question

- **Question 1:** Procedural programs are far more easier to design than Object-oriented programs.
  - True / False?
- **Question 2:** Which of the following features of OOPS prevents data from being directly accessed?
  - Option 1: Data Isolation
  - Option 2: Data Hiding
  - Option 3: Data Encapsulation



# C++

## Lesson 2 : Features of C++

January 28, 2016 | Proprietary and Confidential | 8+



## Lesson Objectives

➤ In this lesson, you will learn about:

- C features used in C++
- Features of C++
  - I/O library usage in C++ program
  - Memory management – static, heap, stack
  - Manipulators
  - Enumerator datatype



January 28, 2016

Proprietary and Confidential

» 2 »



### Lesson Objectives

This lesson covers the features of C++ Programming language. The lesson contents are:

#### Lesson 2: Features of C++

- 2.1. Features Inherited from C
- 2.2. Features of C++
  - 2.2.1. C++ I/O Library
  - 2.2.2. Memory Allocation
  - 2.2.3. Memory Management Operator
  - 2.2.4. Manipulators
  - 2.2.5. Enumerated Data Types

2.11 'C' Features used in C++  
**Features Inherited from C**

➤ **Following features are inherited from C:**

- Built-in types (float, int, etc.), operators, expressions.
- Syntax for pointers, arrays, structures, const, etc.
- Syntax for functions (C prototypes from C++).
- Syntax for decisions and control.
- File organization: header (.h), source (.c, .cc, .C, .cpp, .cxx, etc.).
- Preprocessor (but used less).
- Standard conversions (including narrowing).

**2.2: Features of C++**

## Features of C++: Overview

➤ **Features of C++ are:**

- Strictly enforces prototypes:

```
main()
{
}
```

// will give compilation error  
// as main() is not prototyped

- Data declarations can be done anywhere in the program.

```
for (int i=0;i<10;i++)
```

January 28, 2016 | Proprietary and Confidential | +4+



### Features of C++: Strictly enforces prototypes

```
main()
{
}
```

// will give compilation error  
// as main() is not prototyped

Data declarations:  
Variable can be declared anywhere in the program

```
for(int i=0 ;i<10 ;i++){ }
```

Structures can have functions within them.  
Keyword struct need not be used while declaring structure variables  
It is treated as a User-Defined Data Type:

```
struct rec {
    int a;
};
rec emprec;
```

Contd..

**(contd.):**

The same applies to enums.

Block variable visibility.

You cannot access variables defined within a block outside the block; it is only visible within the braces.

```
void main( )  
  
{  
    for(int n=1 ; n < 10 ; n++)  
    {  
        cout << n;  
        int cube = n * n * n;  
        cout << cube;  
    }  
}
```

Here variable cube is accessible only within the for loop whereas variable n is accessible from the point of declaration onwards.

2.2: Features of C++

## C++ I/O Library

➤ The C++ I/O library:

- #include <iostream>  
    // No ".h" in front of standard headers
- using namespace std;  
    // To prevent "std::" before everything
- contains definitions for several classes
  - istream
  - ostream
  - fstream
  - ios
- >> Extraction Operator extracts data from input stream
- << Insertion Operator inserts data into output stream

January 28, 2016 | Proprietary and Confidential | + 6 +



### C++ I/O Library:

The identifiers `cin` and `cout` are actually objects. They are predefined to correspond to the standard input and output streams respectively. A stream is an abstraction that refers to the flow of data.

The operator `<<` is called the insertion or put to operator. It directs the contents of the variable on its right to the object on its left. It is the overloaded left-shift operator. (Chapter 9 explains the concept of operator overloading.)

The operator `>>` is called the extraction or get from operator. It directs the contents of the object on its left to the variable on its right. It is the overloaded right-shift operator.

The statements beginning with `#` are called preprocessor directives. The header file `iostream.h` contains declarations that are needed by the `cout` identifier and the `<<` operator. Without these declarations, the compiler will not recognize `cout`.

The `<<` and `>>` operators can be cascaded to display or accept multiple values in the same statement.

Example:

```
cin >> a >> b ;
cout << "Value of a is :" << a << "Value of b is :" << b ;
```

Accepting a sentence.

```
cin.get(string,length)
```

Example:

```
char str[10];
cin.get(str,10);
```

Example:

```
#include<iostream>
using namespace std;
void main(void){
    int i=10 ;
    char s[10];
    cout << "Integer:" << i;
    cin >> s;
    cout << "String:" << s;
}
```

2.2: Features of C++  
**Scope Resolution Operator**

➤ **Code Snippet**

```
int m = 10;
main()
{
    int m = 5;
    cout << ::m << m;
}
```

Global variable  
can be accessed  
anywhere  
using :: operator

January 28, 2016 | Proprietary and Confidential | +8+

 Capgemini  
CONSULTING TECHNOLOGY PARTNERING

In the example, `::m`, represents the global variable whereas `m` is the local variable.

## Demo: Accessing Global Variable

➤ Global.cpp



January 28, 2016 | Proprietary and Confidential | + 9 +



### Demo: global.cpp

```
#include<iostream>
using namespace std;
int m = 10;
main()
{
    int m = 5;
    cout << ::m << m;
}
```

2.2: Features of C++

## Memory Allocation

➤ There are three methods of allocating memory in C++:

- Static Allocation
  - Stores Global variables and variables declared static.
- Stack Allocation
  - Stores Function parameters and non-static local variables.
- Heap Allocation
  - The heap is the memory allocated to a program that isn't being used to store the stack, the static variables, or executable code.
  - Dynamically allocated and deallocated using new and delete operators.

January 28, 2016 | Proprietary and Confidential | + 10 c



### Memory Allocations:

#### Static Allocation

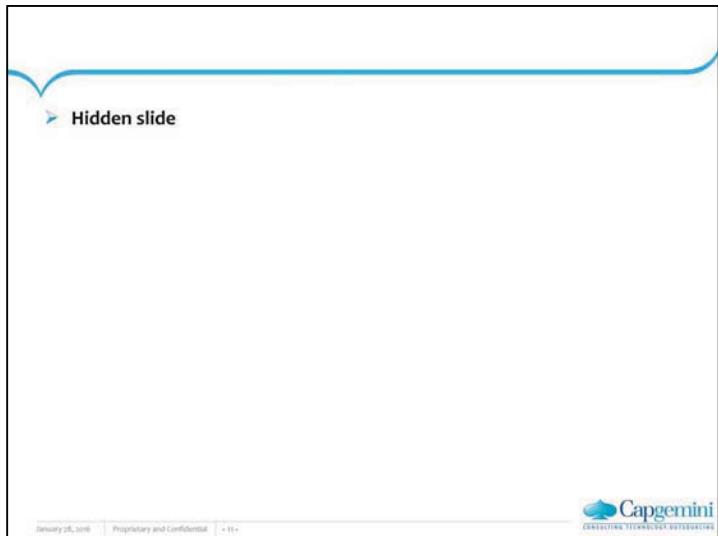
The Static Allocation is used for storing global variables and variables declared static (both local static variables and static class members). These variables have to persist for the entire run of a program—there is no point where they "go out of scope". Static space is permanently allocated and thus not available for anything else.

#### Stack Allocation

The Stack Allocation is used for storing function parameters and non-static local variables. Depending on the processor or OS architecture, either the hardware or the application program maintains a stack, which is used for storing these things. When a function is invoked, it reserves enough space on the stack for its parameters and local variables. This space is reclaimed when the function terminates.

The OS or the application has to set aside a fixed amount of space for the stack. Even though the stack can grow or shrink, potentially allowing the space to be used for some other purpose when the stack doesn't need it, this isn't done, because then you'd have to be careful to move things out of the way when the stack expanded, which is an unreasonable cost. But if the stack exceeds the amount of memory set aside for it (stack overflow), you're just stuck and the program terminates.

Contd..



➤ Hidden slide

January 28, 2016 | Proprietary and Confidential | + 11 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

- **Heap Allocation**

Finally there's **Heap Allocation**. The **heap** is all of the memory allocated to a program that isn't being used to store the stack, the static variables, or executable code.

Typically, this is most of the program's address space.

As a rule, C++ doesn't automatically allocate anything on the heap. To put something on the heap, you ask the memory manager, a service provider of the OS, for a block of memory of a given size.

The memory manager will then satisfy this request with memory from the heap. You ask for a block of memory with `malloc()` in C or with the `new` operator in C++. In other words, the heap is the pool of memory that's available to satisfy calls to `malloc()`.

You use the heap (the `new` operator) to store objects that you want to persist longer than a single function call but shorter than the whole program's duration, and to store larger objects and more complex data structures.

Unlike stack storage, heap storage **isn't automatically reclaimed**; you have to do it explicitly. If you forget, the heap will eventually fill up, leading to crashes or to severely compromised functionality. You allow storage to be reclaimed (and thus reused to satisfy a subsequent `new`) by calling `free()` in C or using the `delete` operator (which usually calls `free()`) in C++.

Heap allocation is considerably slower than stack allocation, both because of the extra bookkeeping the memory manager must do, and often because of the extra bookkeeping the programmer must do.

2.2: Features of C++

## Memory Management Operator

➤ Memory Management Operator are:

- new
- Delete

➤ Example:

```
p = new int;  
delete p;  
  
p = new int[10];  
new int[10][20];  
  
delete [ ] p; // to delete entire array
```

January 28, 2016 | Proprietary and Confidential | + 12 +

 Capgemini CONSULTING TECHNOLOGY PARTNERING

### Memory Management Operators:

Like the malloc () function in C, the new operator in C++ is used to allocate contiguous, unnamed memory in the heap at execution time. Unlike malloc (), however the new operator no longer needs to use the sizeof keyword to specify the exact number of bytes needed. Instead you merely request the number of instances (or variables) of a particular type. The fact that different types occupy different amounts of storage is handled automatically by the compiler. If successful, the call to new will return a pointer to the space that is allocated else it will return NULL if the requested amount of space could not be found or some other error is detected. The difference between malloc and new is that while malloc always returns a pointer of type void\*(hence we need to type cast it to appropriate type), new returns a pointer of the type of object being allocated. To get memory using new operator for a single instance of some primitive type:

```
char* ptr_char = new char;  
int* ptr_int = new int;
```

Primitive types allocated from the heap via new can be initialized with some user-specified value by enclosing the value within parentheses immediately after the type name.

```
int* ptr_int = new int(65);
```

**(Contd..)**

To get memory for array of primitive type:

```
char* ptr_char = new char[20]; // Array of 20 characters  
int* ptr_int = new int[10]; // Array of 10 integers
```

The delete keyword is used to release the space that was reserved by new. It is analogous to the function free () in C, which takes as its argument a pointer to the space.

To delete a single instance from the heap:

```
int* ptr_int = new int;  
delete ptr_int;
```

To delete an array of instances from the heap

```
int* ptr_int = new int[10];  
delete [] ptr_int;
```

To allocate and delete multidimensional arrays:

```
int rows = 3;  
const int cols = 5;  
int (*ptr)[cols] = new int [rows][cols];  
delete [] ptr;
```

When allocating an array from the heap using new, the compiler must know all dimensions, except first. That is why the rows could be determined at execution time, but cols must be constant so that the compiler knows its value.  
To delete the array from the heap, the format of the delete statement is the same as that of a 1-dimensional array.

New and Delete are C++ methods to allocate and deallocate memory. The difference between new and malloc is when you allocate memory using new there is no way to reallocate memory. With malloc you can use the realloc() to extend the amount of memory quickly. With new we'll have to create a new chunk of memory with modified size and then copy the data from the original buffer and delete the original buffer which is a time consuming process.

The bonus of new/delete over alloc/calloc/realloc/free is the fact that new and delete call constructors and destructors of classes (potentially allowing you to clean more memory up easier).

2.2: Features of C++  
**Manipulators**

- Manipulators are operators used with the insertion operator(<<) to modify or manipulate the way data is displayed.
- The header file for manipulators is iomanip.h
- Manipulators are:
  - endl
  - setw()
  - flush
  - hex
  - oct
  - dec

January 28, 2016 | Proprietary and Confidential | +14+



### What are Manipulators?

Manipulators are operators used with the insertion operator << to modify – or manipulate – the way data is displayed. The header file for manipulators is iomanip.h.

#### The endl Manipulator

The endl manipulator sends a new line to the stream and also flushes the output buffer.

```
cout << "Area is" << area << endl ;
```

#### The setw Manipulator

Specifies the width of the value to be printed.

```
cout << setw(5) << m ;
```

#### The flush Manipulator

The flush manipulator flushes the buffer.

```
cout << "Enter a num :" << flush ;
```

#### The dec, oct & hex manipulators

The dec, oct and hex manipulators are used for decimal, octal and hexadecimal values. These manipulators set the stream state either to decimal, octal or hexadecimal. These manipulators work for both input and output streams.

## Demo: Manipulators

➤ Manipulator.cpp



January 28, 2016 | Proprietary and Confidential | + 15+

 Capgemini CONSULTING TECHNOLOGY PARTNERING

### Demo: manipulator.cpp

```
#include <iostream>
using namespace std;
#include <iomanip>
void main()
{
    cout << "Input a hex number ";
    int num ,x1=12345,x2= 67890, x3=1111;
    cout<<"Example of setw()"<<endl;
    cout << setw(12) << "iGatePatni" << setw(18) << "Training" << endl << setw(8) <<
    "Java" << setw(20)<< x1 << endl << setw(8) << "Oracle" << setw(20)<< x2
    << endl << setw(8) << "Unix" << setw(20)<< x3 << endl;
    cout<<"Example of setfill()"<<endl;
    cout << setw(10) << setfill('$') << 50 << 33 << endl;
    cout<<"Example of setprecision()"<<endl;
    double x = 0.1;
    cout << "fixed:" << setprecision(3) << x << endl;
    cout << "scientific:" << x << endl;
    cout<<"Example of hex,oct "<<endl;
    num=25 ;
    cout << "The hex no is " << hex << num << endl ;
    cout << " The octal no is " << oct << num << endl ;}
```

2.2: Features of C++

## Enumerated Data types

➤ Allows us to define our own data types.

- enum color { red,blue,green,yellow};
  - red=0,blue=1....
- enum color { red,blue=4,green=6};
  - color bg=blue;
  - color bg=4
  - color bg=(color); //valid
- enum { off,on}
  - int a=off; //valid in c++
- const int x=0,const int y=1,const int z=2;
  - enum {x,y,z} //valid

January 28, 2016 | Proprietary and Confidential | + 16 +



### What are Enumerated Data Types?

Enumerated Data types allow you to define your own data types. They simplify and clarify the program. Enums provide an elegant way of defining symbolic constants.

Example:

```
enum days_of_week {Sun,Mon,Tue,Wed,Thu,Fri,Sat};  
days_of_week d1,d2 ;  
d1 = Mon ;  
d2 = Thu ;  
int diff = d2 - d1 ;
```

Example:

```
enum boolean {False,True} ;
```

### Demo: Enumeration

➤ enum.cpp



January 28, 2016 | Proprietary and Confidential | +12+

**Capgemini**  
CONSULTING TECHNOLOGY PARTNERING

## Demo:enum.cpp

```
#include <iostream>
using namespace std;
void main() {
    enum days_of_week {Sun=3,Mon=2,Tue,Wed,Thu,Fri,Sat};
    int iDay;
    cout << "Enter A Day(0-Sun and 6-Sat):";
    cin >> iDay;
    cout << endl << (days_of_week)iDay;
    if (iDay==Sun || iDay==Sat) // same as if (iDay==0)
        cout << "Holiday!!!" << endl;
    else
        cout << "Oh No! Working Day!" << endl;
    iDay=Tue;
    cout << iDay;
    if (iDay>0)
        cout << "day is :" << iDay;
}
```

## Summary

- Some of the features inherited from C in C++ are arrays, operators, pointers, decision and controls constructs etc.
- C++ strictly enforces prototype, allows variable declaration anywhere in the program.
- Supports i/o operation with the help of I/O library.
- Manipulators – endl, setw(), flush, hex, oct, dec.
- Enumerators – enum provide an elegant way of defining symbolic constants.



## Review Question

- **Question 1: Manipulators are operators:**
  - True / False?
  
- **Question 2: Select the option that will allow you to define your own data types in C++:**
  - Option 1: Manipulators
  - Option 2: Enumerators
  - Option 3: We cannot define our own data types.





## Lesson Objectives

➤ In this lesson, you will learn about:

- Functions
  - Creating inline functions
  - Passing Default function arguments
  - Concept of function overloading
- Reference variables



January 08, 2016 | Proprietary and Confidential | +3 +

 Capgemini

### Lesson Objectives

This lesson covers functions and references. The lesson contents are:

#### Lesson 3: Functions

- 3.1. Inline functions
- 3.2. Default Function Arguments
- 3.3. Function Overloading
- 3.4. Reference Variables
- 3.5. Restrictions on Reference Variables

3.1: Functions

## Inline Functions

➤ **Each time there is a function call in the source file, the actual code from the function is inserted.**

- This avoids a jump to the function
- This saves execution time for short functions

➤ **Example:**

```
inline float lbstokg (float pounds)
{
    return 0.4535592 * pounds;
}
```

January 08, 2016 | Proprietary and Confidential | +3 +

 Capgemini

### Inline Functions:

Functions save memory space because all the calls to the function cause the same code to be executed; the function body need not be duplicated in the memory. When the compiler sees a function call, it generates a jump to the function. At the end of the function, it jumps back to the instruction following the call.

While the sequence of events may save memory space, it takes some extra time (Recall the explanation of stack allocation, for every function call, a stack frame containing its local variables is pushed on to the stack!). There must be an instruction for the jump to the function, instruction for saving registers, instructions for pushing arguments into the stack in the calling program and removing them from the stack in the function, instructions for restoring registers and an instruction to return to the calling program. The return value (if any) must also be dealt with. All these instructions slow down the program.

To save execution time in short functions you may elect to put the code in the function body directly in line with the code in the calling program. That is, each time there is a function call in the source file, the actual code from the function is inserted, instead of a jump to the function. An Inline function is written like a normal function in the source file but compiles into inline code instead of into a function. When the program is compiled, the function body is actually inserted into the program wherever the function call occurs.

Contd..

## Demo : Inline Functions

➤ inline.cpp



January 08, 2016 | Proprietary and Confidential | + 4 +

 Capgemini  
PERMISSION IS GRANTED TO REPRODUCE THIS MATERIAL FOR PERSONAL STUDY OR INTERNAL DISTRIBUTION

Inline functions must be written before they are called. The inline keyword is actually a hint to the compiler. The compiler may ignore the inline and simply generate code for the function. Generally, the compiler ignores the inline keyword if the code contains a loop, static variable or there is a recursive call to the function. This also varies from compiler to compiler.

Demo: inline.cpp

```
#include <iostream>
using namespace std;
inline float lbstokg(float pounds)
{
    return 0.4535592 * pounds ;
}
void main()
{
    float lbs ;
    cout << "Enter your weight in pounds " ;
    cin >> lbs ;
    cout << "Your weight in Kilograms is :" << lbstokg(lbs) ;
}
```

3.1: Functions

## Default Function Arguments

➤ The mandatory arguments, if present are always written first.

```
int average(int a,int b = 0,int c = 0)
{
}
void main()
{
    int a = average(2);
    int b = average(3,4);
}
```

The initialization takes place only if the argument is not passed.

January 08, 2016 | Proprietary and Confidential | +3 +

 Capgemini

### Default Function Arguments:

C++ allows you to call a function without specifying all its arguments. In such cases, the function assigns a default value to the arguments, which do not have matching arguments in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

A function's formal argument list now contains two types of formal arguments: mandatory and default. The order of these arguments must be as follows: the mandatory arguments, if present is written first, followed by any default arguments. A function may thus contain all mandatory arguments, all default arguments or a combination thereof.

When the function is called, you must first specify all of the mandatory actual arguments, if present, in the order in which they are declared. Next, you may override the default arguments, if any, but only in the order in which they were declared.

## Demo : Default Arguments

➤ default.cpp



January 08, 2016 | Proprietary and Confidential | +6+

 Capgemini

## Demo: default.cpp

```
#include <iostream>
using namespace std;
double area(int l=2) {
    return (3.14*l*l);
}
void main()
{
    cout << " Calling Overloaded functions " << endl;
    cout << "Area of Circle:" << area();
}
```

**3.1: Functions**

## Function Overloading

- Function overloading provides a way to have multiple functions with the same name.
- Functions that share the same name with different argument list are said to be overloaded.
  - float FindMax(float a, float b);
  - float FindMax(float a, float b, float c);
  - int FindMax(int a, int b);
- It is far more convenient to use the same name for functions, though each have different arguments.
- Some 'C' standard functions that do the same action with different values:
  - abs() – returns absolute value of an integer
  - labs() – returns absolute value of a long number
  - fabs() – returns absolute value of a float integer

January 08, 2016 | Proprietary and Confidential | +2 +

 Capgemini

### Function Overloading:

One way C++ achieves polymorphism is using function overloading.

In C++, two or more functions can share the same name as long as their arguments are different.

In this situation, the functions that share the same name are said to be overloaded, and the process is referred to as function overloading.

An overloaded function appears to perform different activities depending on the kind of data sent to it.

Function overloading can occur only within a single scope.

If two or more functions exist with the same name in the same scope, then the determination as to which function is actually called at runtime depends on the number and type(s) of the argument(s) that are supplied in the actual call of the function. Note that the variation in the argument list does not consider the return type of the overloaded functions. The return type may or may not be same.

Function Overloading can occur only within a single scope. Hence two different classes having the same function name is not function overloading.

It is far more convenient to use the same name for functions, though each have different arguments.

## Function Overloading (contd..)

➤ In C++ you can have following functions:

- abs(int)
- abs(long)
- abs(float)

### Why Function Overloading?

To see why function overloading is important, first consider three functions found in the standard library of all C compilers: `abs()`, `labs()`, and `fabs()`. The `abs()` function returns the absolute value of an integer, `labs()` returns the absolute value of a long, and `fabs()` returns the absolute value of a double.

Although these functions perform almost identical actions, in C, three slightly different names must be used to represent these essentially similar tasks. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, the programmer has to remember three things, not just one. However, in C++ you can use just one name for all three functions as illustrated below.

## Function Overloading: Example

### ➤ Code Snippet

```
void print(int i){  
    cout << " Here is int " << i << endl; }  
void print(double f){  
    cout << " Here is float " << f << endl; }  
void print(char* c){  
    cout << " Here is char* " << c << endl; }  
void main(){  
    print(10);  
    print(10.10);  
    print("ten");  
}
```

January 08, 2016 | Proprietary and Confidential | + 9 +



### Why Function Overloading?

In Function Overloading the compiler goes through a process known as “argument matching”, which involves an algorithm that matches the actual arguments in the function call against the argument list of all the functions with the same name. The compiler will then choose the function that best matches the actual argument(s) from among all of the functions by that name that are in scope and for which a set of conversions exists so that the function could possibly be called.

## Demo:Function Overloading

➤ **overload.cpp**



January 08, 2016 | Proprietary and Confidential | + 10 +

 Capgemini

Demo : overload.cpp

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

double area(int l=2) {
    return (3.14*l*l);
}
int area(int l,int b) {
    return l*b;
}
double area(int l,int b,float h) {
    return (0.5*l*b*h);
}
void main() {
    cout << " Calling Overloaded functions " << endl;
    cout << "Area of Circle:" << area(5);
    cout << "Area Of Rectangle :" << area(2,3);
    cout << "Area of Rectangle " << area(2,3,5);
}
```

3.1: Functions

## Reference Variables

➤ **A Reference Variable:**

- Provides an alias to another variable
- Unlike a pointer variable, does not have its own unique address

➤ **Example:**

```
float sum = 100;  
float &total = sum;
```

➤ **Both sum and total refer to same data:**

```
sum = 50;  
total += 50; //valid  
cout<<"sum=<<sum<<"total=<<total
```

January 28, 2016 | Proprietary and Confidential | + 11 +

 Capgemini

### What is a Reference Variable?

A reference variable is new type in C++ that provides the means by which an alias to another variable can be created.

What is an alias?

Just another name for something or someone.

This means that whenever the alias is used , the action is really taking place on the object to which the alias refers.

While a reference variable is conceptually similar to a pointer variable in that they both refer or point some other project,a reference variable is a hidden pointer because unlike a pointer variable , it does not have its own unique address.

In the example shown above, total is an alias for sum and whenever total is referenced sum is really being used.

Note:

Once a reference variable is initialized with some existing variable, it cannot be reassigned to some other variable as long as the reference variable remains in scope.

## Reference Variables (contd..)

➤ **Main advantage of reference variables, is that**

- They replace function calls that are pass by address with calls that are “pass by reference”.
- Easier to code function and function calls.

January 08, 2016

Proprietary and Confidential

+ 42 +



Why use Reference Variables?

Benefit of using reference variables:

One of the great benefits of using reference variable is that in many cases they can replace function calls which are “pass by address” with calls that are “pass by reference”.

This syntax does not involve your having to manually generate the address of a variable,nor do you have to create a pointer variable in which to receive the address. As a result it is much easier to code functions and function calls.

## Demo : Reference Variable

➤ swap.cpp



January 08, 2016 | Proprietary and Confidential | + 13 +



### Demo : swap.cpp

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

void swap (int& first, int& second) {
    int temp = first;
    first = second;
    second = temp;
}
void main(){
    int a = 2,b = 3;
    swap( a, b );
}
```

Here's a simple example of setting up a function to take an argument "by reference", implementing the swap function:

Both arguments are passed "by reference"--the caller of the function need not even be aware of it:

After the swap, 'a' will be 3 and 'b' will be 2. The fact that references require no extra work can lead to confusion at times when variables magically change after being passed into a function. So ,for arguments that the function is expected to change, using a pointer instead of a reference helps make this clear--pointers require that the caller explicitly pass in the memory address.

## Restrictions on Reference Variables

### ➤ Rules applicable to Reference variables:

- A reference must always be initialized
- Null references are prohibited
- You cannot have a reference to a reference
- You cannot create arrays of references
- You cannot have a pointer to a reference.

January 08, 2016 | Proprietary and Confidential | + 34 +



1) A pointer can be re-assigned:

```
int x = 5;
int y = 6;
int *p;
p = &x;
p = &y;
*p = 10;
assert(x == 5);
assert(y == 10);
```

A reference cannot, and must be assigned at initialization:

```
int x = 5;
int y = 6;
int &r = x;
```

2) You can have pointers to pointers to pointers offering extra levels of indirection.  
Whereas references only offer one level of indirection.

```
int x = 0;
int y = 0;
int *p = &x;
int *q = &y;
int **pp = &p;
pp = &q; /* pp = q
**pp = 4;
assert(y == 4);
assert(x == 0);
```



January 08, 2016 | Proprietary and Confidential | + 15 +



- 3) Pointer can be assigned NULL directly, whereas reference cannot. If you try hard enough, and you know how, you can make the address of a reference NULL. Likewise, if you try hard enough you can have a reference to a pointer, and then that reference can contain NULL.

```
int *p = NULL;  
int &r = NULL; <-- compiling error
```

- 4) Pointers can iterate over an array, you can use `++` to go to the next item that a pointer is pointing to, and `+ 4` to go to the 5th element. This is no matter what size the object is that the pointer points to.

Lab

➤ Lab 2



January 18, 2016 | Proprietary and Confidential | +16+

 Capgemini

## Summary

- **Inline function** is written like a normal function in the source file but compiles into inline code instead of a function.
- The function assigns a default value to the arguments, which do not have matching arguments in the function call.
- **Function overloading** - Using the same name for functions, though each have different arguments.
- **Reference variable** - provides an alias to another variable.



## Review Question

- **Question 1:** Which of the following is true about inline functions?
    - Option 1: They save memory space
    - Option 2: They reduce execution time
    - Option 3: They must be written before they are called.
  - **Question 2:** Default values can be assigned only to leading arguments?
    - True / False?
  - **Question 3:** Reference variables and pointers are one and the same.
    - True / False?



January 28, 2004 Proprietary and Confidential - 18 -





C++

**Lesson 4: Classes**

## Lesson Objectives

➤ In this lesson, you will learn about:

- Classes
  - Class member Functions
  - Concept of this pointer
- Static members
  - Static data
  - Static methods
- Type Qualifiers
  - const
  - mutable
  - volatile



January 18, 2016 | Proprietary and Confidential | + 3 +



### Lesson Objectives

This lesson covers classes and class members. The lesson contents are:

#### Lesson 4: Classes

- 4.1. Classes
- 4.2. Categories of Class Member Functions
- 4.3. this Pointer
- 4.4. Function Chaining
- 4.5. Static Data Members
- 4.6. Static Member Function
- 4.7. Type Qualifiers

4.1: Class Syntax

- Class in C++ is identical to structure
- Syntax:

```
class <class name>
{
    private :
        //Data and member functions
    public :
        //Data and member functions
};
```

January 26, 2016 | Proprietary and Confidential | +3 +

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS

#### How to write classes?

A class allows you to define, associated pieces of data together with the functions to be used with that data and create a new type of data according to the needs of the problem to be solved. This new type is also called abstract data type or ADT. A class allows data and functions to be hidden, if desired. A class definition consists of the class keyword, followed by the name of the class followed by a body, enclosed in braces, followed by semicolon. The class body contains variable definitions and function declarations. These variables and functions are intimately tied to the class and may only be used in association with an object belonging to that class. Although the variable definitions look like the ordinary definitions of local variables, no memory is allocated for them until a variable (object) of the class type is created. When this happens, all the memory for the variable is allocated at once.

The variables and functions (collectively called members) of a class are normally hidden from the outside world – the user cannot access them. These variables and functions are called private. The privacy can be made explicit using the keyword 'private'; members in a class default to private. To allow the client programmer access to members, use the public keyword.

## Demo : Creating Classes

➤ smalllobj.cpp



January 26, 2016 | Proprietary and Confidential | + 6 +

### Demo : smalllobj.cpp

```
#include <iostream>
using namespace std;
class smallobj {
private :
    int somedata ;
public :
    void setdata(int d){
        somedata = d ;
    }
    void smallobj:: showdata( ){
        cout << "Data is : " << somedata ;
    }
void main( ){
    smallobj s1,s2 ;
    s1.setdata(1032);
    s2.setdata(1784);
    s1.showdata();
    s2.showdata();
}
```

4.3: this Pointer

## Introduction

- When a member function is called, the compiler must know which instance the member function is working on.
- For this reason, each non-static member function has access to a pointer variable that points to the instance being manipulated.
- This pointer variable is called the **this** pointer.

January 26, 2016 | Proprietary and Confidential | +5+

 Capgemini CONSULTING TECHNOLOGY BUSINESS SERVICES

### What is the this Pointer?

When several instances of a class are created, each instance has its own separate copy of the member variables. But only one copy of each member function, per class, is stored in the memory and therefore is shared by all instances of that class. When a member function is called, the compiler must know which instance the member function is working on. For this reason, each nonstatic member function has access to a pointer variable that points to the instance being manipulated. This pointer variable is called the **this** pointer. The **this** pointer is passed automatically as an implicit argument to the member function when the function is called. In the earlier example when the member function `setdata()` is called for object `s1`

The **this** pointer is passed implicitly to the function. So in this case the **this** pointer has the address of `s1`. In the function, the statement

```
s1.setdata(1032);
```

Here the data member `somedata` does not have any object or class qualification. This statement will now be qualified using the **this** pointer.

```
somedata = d;
```

```
this -> somedata = d
```

4.2: Inline Class Member Function

## Example

➤ Code Snippet

```
class item
{ private :
    int itemno;           //data
    float cost;
public :
    void getdata(int a,float b); //methods
    void putdata(void);
    void setzero(void)         //inline member function
    { itemno = 0; cost = 0.0; }
};
```

January 26, 2016 | Proprietary and Confidential | + 6 +

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS

#### Inline Class Member Function:

If you define a member function within the scope of the class definition, then the member function becomes automatically inline. The keyword `inline` need not be written. In the example: `setzero()` are written within the scope of the class definition, they would be inline functions, even though the keyword `inline` is not specified.

4.2: Inline Class Member Function

## Categories of Class Member Functions

➤ **Class Member Functions are:**

- **Manager Functions** -Constructors and Destructors
- **Accessor Functions** - constant functions that return information about an object's current state.
- **Implementer Functions** - functions that make modifications to the data members

January 26, 2016 | Proprietary and Confidential | +2+

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

Class Member Functions are:

Manager Functions : These functions are used to perform “initialization” and “cleanup” of data members of the class.

Constructors and Destructors functions come under this category.

Accessor Functions : These are constant functions that return information about an object's current state.

To ensure that these functions do not make any kind of modification to the data members, you add the keyword const after the formal argument list.

Implementer Functions : These are the functions that make modifications to the data members.

This is how the state of an object can be changed. These functions are also known as mutators.

### 4.3: this Pointer this Pointer (contd..)

- The this pointer is passed automatically as an implicit argument to the member function when the function is called.
- Example:

```
itemno = a;           this->itemno = a;  
cost = b;            this->cost = b;
```



January 26, 2016 | Proprietary and Confidential | 8 +

(Contd..):

Now it is clear that the data member somedata of object s1 is to be manipulated. The this pointer is implicitly declared. It can be used explicitly where you deem appropriate. The this pointer has absolutely no meaning outside the member function.

The this pointer is passed automatically as an implicit argument to the member function when the function is called

Hence, in the example above the member variables itemno and cost can also be referred as this->itemno and this->cost.

4.4: Function Chaining  
**Introduction**

- Function chaining allows you to pass multiple messages to an object in the same statement.
- You can call multiple member functions for a particular object in a single statement.
- Example:
  - `instance.message1( ).message2( ).message( );`

```
it1.getdata(10,5.60).putdata();
```

January 26, 2016 | Proprietary and Confidential | 9 | Capgemini CONSULTING TECHNOLOGY SERVICES

#### What is Function Chaining?

Function chaining or function concatenation allows you to pass multiple messages to an object in the same statement. i.e. You can call multiple member functions for a particular object in a single statement.

```
instance.message1( ).message2( ).message( );
```

The “trick” is to make sure that each method that carries out a message returns the invoking instance (`*this`) by reference. This instance then may be used as the (implicit) invoking instance of the next function call without having to write it again explicitly.

4.1: Class  
Example

January 26, 2016 | Proprietary and Confidential | +10+

 Capgemini CONSULTING TECHNOLOGY INNOVATION

#### Writing Classes: An example

An object is an instance of a class. Here we have two objects s1 and s2. A key feature of object-oriented programming is data hiding. It means that data is concealed within a class. Private data or functions can only be accessed from within the class. Public data or functions can be accessed from outside the class. Because setdata() is a member function of the smallobj class, it must always be called in connection with an object of this class. Therefore s1.setdata().

The member functions of the class can be written outside the class. This is done using the scope resolution operator, to specify that the function belong to a particular class as shown for showdata().

4.5: Static Data Members

## Characteristics

- **Static members are automatically initialized to zero.**
  - You may initialize them to some other value.
- **Only one copy of the variable is created for multiple objects.**
- **Visible only within the class but life of static member is life of program.**
- **It is completely independent of any and all class instantiations.**
- **It is created and initialized before main( ).**

January 26, 2016 | Proprietary and Confidential | +1+

 Capgemini CONSULTING TECHNOLOGY BUSINESS

### What are Static Data Members?

In C++, you may declare static variables as data members.

Static members are automatically initialized to zero. You may initialize them to some other value.

Only one copy of the variable is created for multiple objects.

Visible only within the class but life of static member is life of program.

It is completely independent of any and all class instantiations.

It is created and initialized before main( ).

Static data members are instance-independent and are only declared within the scope of the class definition. The definition is done in the global part of the program after class definition. This is the case for all static members private, public and protected.

When writing the definition, you must not repeat the word static. However, you must repeat the type of the variable, followed by the class name, scope resolution operator and variable name. Initialization may be done here ; otherwise, it is initialized to zero.

4.6: Static Member Functions

## Description

- A static member function can have access to only static member variables.
- A static member function can be called using the class.
- Example:

class name :: function name

January 26, 2016 | Proprietary and Confidential | v12 |

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Static Member Functions:

In C++, you may declare member functions as static.

A static member function can have access to only static member variables

While it is a member function, there is no this pointer.

A static member function can be called using the class name, since it is instance-independent.

class name :: function name

Since C++ requires an instance of the class to invoke a nonstatic member function, and since static class members are instance-independent, it does not make sense to use the member function showcount() to display variable count in the earlier example.

## Demo : Static Members

> item.cpp



January 26, 2016 | Proprietary and Confidential | +13+

Capgemini CONSULTING TECHNOLOGY OUTSOURCING

## Demo: item.cpp

```
include <iostream>
using namespace std;
class item {
    static int count ; //count is declared static
    int number ;
    public :
    void getcount(int a){
        number = a ;
        count++ ;
    }
    static void showcount( ){
        cout << count ;
    }
};
int item::count ; //count is defined here ;
void main(void){
    item it1,it2;
    it1.getcount(5);
    it2.getcount(5);
    item.showcount();
}
```

4.7: Type Qualifiers

## Type Qualifiers

- A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:
  - The value of an object can be changed
  - The value of an object must always be read from memory rather than from a register
  - More than one pointer can access a modifiable memory address
- C++ identifies the following identifiers
  - const
  - volatile:

January 26, 2016 | Proprietary and Confidential | +14+

 Capgemini CONSULTING TECHNOLOGY BUSINESS

### Type Qualifiers

#### const

The const qualifier explicitly declares a data object as something that cannot be changed. Its value is set at initialization. You cannot use const data objects in expressions requiring a modifiable lvalue. For example, a const data object cannot appear on the lefthand side of an assignment statement. In C++ a global const object without an explicit storage class is considered static by default, with internal linkage.

```
const int pi=3.14
```

#### volatile

The volatile qualifier declares a data object that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed.

Note: An item can be both const and volatile. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process

4.8: Const Member Functions

## Description

➤ **Const Member Functions:**

- Cannot modify the value of any data member(except mutable).
- The keyword must be present in the declaration as well as the definition

➤ **Syntax:**

return\_type function\_name const;

➤ **Example**

int getMonth() const;

January 16, 2016 | Proprietary and Confidential | +95+

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS

### Const Member Functions:

The const member functions are basically the Accessor functions. Declaring a member function with the const keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.

To declare a constant member function, place the const keyword after the closing parenthesis of the argument list. The const keyword is required in both the declaration and the definition. A constant member function cannot modify any data members(except mutable) or call any member functions that aren't constant. Constant functions may be used with constant objects as well.

### Mutable:

The mutable storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as const. You cannot use the mutable specifier with names declared as static or const, or reference members. They are defined using the keyword mutable.

Let's summarize the concept of cost objects and cost member functions:

(Cont...)

## Demo : Constant Members

- **constant\_function.cpp**
- **constantsdefine.cpp**
- **mutable\_variable.cpp**



January 26, 2016 | Proprietary and Confidential | +16 +

(Contd...)

If you do not want any member function to change the private data of your object then declare the object as a const object. You will have to initialize this object using a constructor. After that, the object will not change during the entire program execution.

Set all mutator/ implementer functions (functions changing the values of data members) as non-constant functions

Set all accessor functions (functions which only inspect the values of data members) as const so that they do not accidentally change the values of data members.

Non const functions will not be available to const objects.

Constant functions will be available to const as well as non-const objects.

Demo:constant\_function.cpp

```
#include<iostream>
using namespace std;
class Date {
public:
    Date( int mn, int dy, int yr );
    int getMonth() const; // A read-only function
    void setMonth( int mn ); // A write function; can't be const
```

### **#define vs Const**

1. **#define** is taken care of by the pre-processor and the compiler is used in **const**.
2. Values defined with **const** are subject to type checking
3. **#define** is text substitution, not a variable.

To understand this refer to this code.

#### **Demo: constantvsdefine.cpp**

```
#include<iostream>
using namespace std;
const int NUMBER = -42;
#define NUMBER=-42;
void main()
{
    int x = -NUMBER;
}
```

Now try executing the code by commenting the **const** definition and uncommenting **#define**.

```
private:
    int month;
}
int Date::getMonth() const {
    return month; // Doesn't modify anything
}
void Date::setMonth( int mn ){
    month = mn; // Modifies data member
}
void main(){
    Date MyDate( 7, 4, 1998 );
    const Date BirthDate( 1, 18, 1953 );
    MyDate.setMonth( 4 ); // Okay
    BirthDate.getMonth(); // Okay
    // BirthDate.setMonth( 4 );// Error
}
```

Demo: mutable\_variable.cpp

```
#include<iostream>
using namespace std;
class sample {
    int a ;
    mutable int b ;
public:
    sample() {
        a=10;
        b=20;
    }
    void show( ) const ;
};

void sample :: show( ) const{
    cout<< a ;
    //cout << a++ ; // error. Show cannot change a
    cout<< b++ ; // valid as b is mutable
}

void main() {
    sample s;
    s.show();
}
```

Now try executing the code by commenting the line in show method where it increments the value of a normal variable.

**Lab**

➤ **Lab 3**



January 26, 2016 | Proprietary and Confidential | +79+

**Capgemini**  
CONSULTING TECHNOLOGY OUTSOURCING

Add the notes here.

## Summary

- Class body contains variable definitions and function declarations.
- Class member functions are – manager, accessor and implementer.
- this pointer points to the current instance of the class.
- You can call multiple member functions for a particular object in a single statement using function chaining.
- Static data members - Only one copy of the variable is created for multiple objects.
- Static functions - A static member function can be called using the class name.



## Review Question

- **Question 1:** Which of the following is implicitly passed as an argument when an object makes a function call.
  - Option 1: this pointer
  - Option 2: Class name
  - Option 3: Object
- **Question 2:** Which of the following is true for C++.
  - Option 1: An object can be a class
  - Option 2: A class must have at least one member function
  - Option 3: An object can exist without a class
- **Question 3:** Static data can be accessed only by static member functions.
  - True / False?



C++  
**Constructors and Destructors**

January 08, 2008 | Proprietary and Confidential | 17



## Lesson Objectives

➤ In this lesson, you will learn about:

- Rules for creating constructors
- Constructor overloading
- Concept of copy constructor
- Initialization section of a constructor
- Destructor



January 08, 2016 | Proprietary and Confidential | 22



### Lesson Objectives

This lesson covers constructors and destructors in C++. The lesson contents are:

#### Lesson 5: Constructors and Destructors

- 5.1. Constructors
- 5.2. Constructor Overloading
- 5.3. Copy Constructor
- 5.4. Constructor Initialization Section
- 5.5. Destructor

5.1 Constructors

## What is a Constructor?

- Constructor is a non-static member function that is automatically executed whenever an instance of the class to which the constructor belongs, comes to existence.
- Rules for creating constructors:
  - Name of constructors should be the same as name of the class
  - It is declared with no return type not even void
  - Cannot be declared as static or const
  - Should have public or protected access

January 08, 2016 | Proprietary and Confidential | 4 / 34

 Capgemini  
CONSULTING SERVICES

### What are Constructors?

Constructors are non-static member functions that determine how the object of a class is created, initialized and copied. A constructor executes automatically when an object is created. Constructors are also invoked when local or temporary objects of a class are created.

The constructor can be overloaded to accommodate many different forms of initializations for instances of the class. The execution of the constructor does not reserve memory for the instance itself. The compiler generates the code to do this, either in static memory, on the stack, or on the heap, after which control is handed over to the constructor to do the initialization of the data members.

### Rules for Creating Constructors:

Name of Constructors should be the same as name of the class to which it belongs.

It is declared with no return type, not even void.

It cannot be declared as static or const.

It should have public or protected access.

It can be overloaded.

## Example : Default Constructors

### Code Snippet

```
class sample
{
    private :
        int a;      float b;
    public :
        sample(void)
        {
            a = 0; b = 0.0;
        }
        void showdata()
        {
            cout << a << b;
        }
};

void main(void)
{
    sample s1;
    s1.showdata();
}
```

January 08, 2008

Proprietary and Confidential



When is the Constructor called?

A constructor is called when:

An instance of some class comes into existence and causes the constructor function to be invoked.

A global or static local variable of the class is defined (the constructor is called before main ()).

An auto variable of the class is defined within a block and the location of its definition is reached.

A temporary instance of the class needs to be created.

An instance of the class is obtained from free memory via the new operator.

The default constructor:

If you fail to write a constructor and destructor function, the compiler automatically supplies them for you. These functions have public access within the class and do nothing.

The term default constructor refers to the constructor that takes no arguments, whether it is supplied automatically by the compiler or is written by you. The constructor that takes all default arguments may also serve as the default constructor.

## Example : Parameterized Constructors

### Code Snippet

```
class sample
{
    private :
        int a; float b;
    public :
        sample(int m, float n)
        {
            a=m; b=n;
        }
        void showdata()
        {
            cout << a << b;
        }
};
void main(void)
{
    sample s1(10 , 45.7);
    s1.showdata();
}
```

### Parameterized Constructors

Constructor that accepts arguments are known as parameterized constructors. There may be situations, where it is necessary to initialize various data elements of different objects with different values when they are created. This objective can be achieved through parameterized constructors.

## Example: Constructor Overloading

### Code Snippet

```
class sample
{
    private :
        int a;
        float b;
public:
    sample(int m,float n)
    {
        a = m; b = n; }
    sample(void)
    {
        a = 0; b = 0.0; }
    sample(float n)
    {
        a = 0; b = n; }
};
```

January 08, 2006 | Proprietary and Confidential | 8-6



### Constructor Overloading

Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter lists. A constructor uses its arguments to initialize the new object's state. The compiler differentiates these constructors by taking into account the number of parameters in the list and their types.

## Demo: Parameterized Constructor

➤ Parameterized\_constructor.cpp



January 08, 2016 | Proprietary and Confidential | 2

 Capgemini  
CONSULTING TECHNOLOGIES SERVICES

### Demo: parameterized\_constructor.cpp

```
#include<iostream>
using namespace std;
class Student
{
    char *stud_id,*stud_name;
public:
    Student()
    {
        stud_id="2";
        stud_name="xyz";
    }
    Student(char *id,char *name)
    {
        stud_id=id;
        stud_name=name;
    }
    void display()
    {
        cout<<" Student Id:"<<stud_id;
        cout<<"\n Student Name :"<<stud_name;
    }
}
int main()
{
    Student s1("3","abc");
    s1.display();
    return 1;
}
```

## What is a Copy Constructor?

- Copy Constructor is the constructor used to handle the creation and initialization via an existing instance.
- Example

```
class integer {  
    private :  
        int code ; float sal ;  
    public :  
        integer(integer& i )//copy constructor.  
        //Argument passed by reference  
        { code = i.code ; sal = i.sal ; }  
        void showdata()  
        { cout << code << sal ; }  
};
```

January 08, 2006 | Proprietary and Confidential | 8



### What is a Copy Constructor?

The constructor to handle the creation and initialization of a class instance via an existing instance is called the copy constructor.

Like the default constructor, if you do not write a copy constructor, then the compiler will automatically provide one for you. The default copy constructor performs member wise copy (also called as shallow copy) from the existing object to the new instance. This member wise copy is recursive in the sense that if a data member is also an instance of some other class, then all its data members are also copied.

The general rule is that whenever there are pointer data members as part of the class definition, and then you must write your own copy constructor.

### When is the copy constructor is called?

It is called when:

An instance of a class is created from an existing instance of the class.  
A function is called and an existing instance of the class is passed as an argument.  
An unnamed temporary variable is created to hold the return value of a function returning an instance of the class.

## Demo : Copy Constructor

> **Copy\_constructor.cpp**



January 08, 2008 Proprietary and Confidential 8

 Capgemini CONSULTING TECHNOLOGIES SERVICES

### Demo: copy\_constructor.cpp

```
#include<iostream>
using namespace std;
class integer
{
private : int code ;
        float sal ;
public: integer(integer& i ) //Copy constructor
{
        code = i.code ;
        sal = i.sal ;
}
void showdata()
{
        cout << code << sal ;
}
};
void main(void)
{
        integer s1(101,10000);
        integer s2(s1); //Copy constructor invoked
        s1.showdata();
        s2.showdata();
}
```

## Initialization Section in a Constructor

### ➤ Code Snippet

```
class employee
{
    private :
        int code ;
        float sal ;

    public :
        employee( int cd, float s ) : code ( cd ), sal ( s ) { }

        employee (employee& e) : code ( e.code ), sal ( e.sal ) {
    };
}
```

January 08, 2008

Proprietary and Confidential



### Base member initialization:

Data members in a class can be initialized in a constructor function using the syntax as shown in the example on the slide.

5.2: Destructors

## What is a Destructor?

- Destructor is a non-static member function
- Automatically executes whenever an instance of the class is destroyed
- Rules for creating destructors:
  - Name of destructors should be the same as name of the class and first character is ~ (tilde)
  - It is declared with no return type not even void
  - Cannot be declared as static or const
  - Takes no arguments
  - Should have public access

January 08, 2016 | Proprietary and Confidential | 10 / 14

 Capgemini  
CONSULTING SERVICES

### What is a Destructor?

A destructor is a function, which is called automatically whenever an instance of the class goes out of existence. The destructor is used to release space on the heap that the instance currently has reserved.

#### Rules for Destructors:

Its name is the same as that of the class to which it belongs, except that the first character of the name must be a tilde(~).  
It is declared with no return type, not even void, since it cannot return a value.  
It cannot be declared static or const.  
It takes no input arguments, and so cannot be overloaded.  
It should have public access in the class declaration.

Contd...

## Example

### Code Snippet

```
class sample
{
    private :
        int * a;
    public :
        sample(void)
        {
            a = new int;
        }
        ~sample()
        {
            delete a;
        }
};
```

January 08, 2016

Proprietary and Confidential



When is a destructor function called?

The destructor function for a class is called:

After the end of main( ) for all static, local to main( ), and global instances of the class.

At the end of each block containing an auto variable of the class.

At the end of each function containing an instance of the class as argument.

To destroy any unnamed temporary instances of the class after their use.

When an instance allocated on the heap is destroyed via delete.

When an object containing an instance of the class is destroyed.

When an object of a class derived from the class is destroyed.

**Lab**

➤ Lab 4

 Hands On

January 28, 2016 | Proprietary and Confidential | 17



## Summary

- Constructors are non-static member functions that determine how the object of a class is created, initialized and copied.
- A constructor to handle the creation and initialization of a class instance via an existing instance is called the copy constructor.
- Destructors are non-static member functions that automatically execute whenever an instance of the class to which the destructor belongs to.



## Review Question

- **Question 1:** If constructor for a class is not specified the default constructor is used.
  - True / False
  - Option 2
  - Option 3
- **Question 2:** A constructor is used to:
  - Option 1: Allocate memory
  - Option 2: Initialize a newly created object
  - Option 3: Declare a variable
- **Question 3:** Can we write overloaded destructor?
  - Yes / No



## C++

### Lesson 6: Friend Function and Friend Class

January 08, 2008

Proprietary and Confidential



## Lesson Objectives

➤ To understand the following:

- Concept of friend functions and classes.
- Write friend functions.
- Need for namespaces.
- Std namespace.



January 08, 2016

Proprietary and Confidential



### Lesson Objectives

This lesson covers friend functions and friend classes. The lesson contents are:

#### Lesson 6: Friend Function and Class

- 6.1. Friend Function
- 6.2. Friend Class
- 6.3. Namespace
- 6.4. Using keyword
- 6.5. std namespace

## 6.1. Friend Functions

- Not member functions of a class
  - They have access to private members of a class.
- Outside functions are denied access to class private data.
- Only class member functions can access private members

January 08, 2008 | Proprietary and Confidential | 34



Any data that is declared private inside a class is not accessible from outside the class. A function which is not a member can never access private data. However, there may be some cases, where a programmer needs access to the private data from non-member functions and external classes. C++ offers some exceptions in such cases.

A class can allow non-member functions and other classes to access its own private data, by making them as friends.

Declaring the function to be a friend of the class in which the data members are located can solve this dilemma. That is, the class in which the private members are located bestows friendship upon the function. Vice-versa is not allowed.

As a friend function is always a non-member function of the class that bestows the friendship, it cannot be called via some instance of the class. Therefore there is no this pointer. Friend function is usually called with at least one argument consisting of an instance of the class, which bestowed the friendship.

Imagine that you want a function to operate on objects of two different classes. Perhaps the function takes objects of the two classes as arguments and operates on their private data.

## Features

- Friend function is out of the scope of this class.
- You cannot call them by using objects.
- Usually, they comprise objects as arguments.

January 08, 2016

Proprietary and Confidential



Following are some things that you should know about friend functions:

When friends are specified within a class, this does not give the class itself access to the friend function. That function is not within the scope of the class; it's only an indication that the class will grant access to the function.

when we are calling a friend function we don't need to call it as a method of an object of a class

Usually , it has the objects as arguments.

## Example

### ➤ Code Snippet

```
class complex {  
    .....  
public: friend complex add (complex &,complex &);  
};  
  
complex add (complex &c1 , complex &c2) {  
    .....  
}  
void main(void) {  
    complex c1(1.2,1.3),c2(2.0,3.5),c3;  
    c3 = add(c1,c2);  
    .....  
}
```

January 08, 2008

Proprietary and Confidential



Points to be remembered while creating friend functions:

The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

It is possible to declare a function as friend in any number of classes.

When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.

A friend function, even though it is not a member function, would have the rights to access the private members of the class.

It is possible to declare the friend function as either private or public.

The function can be invoked without the use of an object. The friend function has its argument as objects.

## Demo: Friend Functions

- friend\_function.cpp
- friend\_function1.cpp



January 08, 2008 Proprietary and Confidential 6

 Capgemini  
CENTRAL TECHNOLOGY SERVICES

### Demo : friend\_function.cpp

```
#include <iostream>
using namespace std;
class beta ; //Forward declaration
class alpha {
    private :
        int data ;
    public :
        alpha(){
            data = 3 ;
        }
        friend int frifunc(alpha , beta) ;
};
class beta {
    private :
        int data ;
    public :
        beta(){
            data = 7 ;
        }
        friend int frifunc(alpha , beta) ;
};
int frifunc(alpha a, beta b){
    return(a.data + b.data);
}
```

```
void main(){
    alpha aa ;
    beta bb ;
    cout << frifunc(aa,bb) ;
}
```

### Demo : friend\_function1.cpp

```
#include<iostream>
using namespace std;
class X;
class Y{
public: void print(X& x);
};
class X{
int a, b;
friend void Y::print(X& x);
public: X(): a(1), b(2){ }
};
void Y::print(X& x){
cout << "a is " << x.a << endl;
cout << "b is " << x.b << endl;
}
void main() {
    X xobj;
    Y yobj;
    yobj.print(xobj);
}
```

## 6.2. Friend Class

- A class declared a friend of another class has access to private members defined within the other class
  - All member functions of the friend class are friend functions of the class that bestows the friendship.

January 08, 2016

Proprietary and Confidential



It is possible for one class to grant friendship to another class by writing the keyword `friend` followed by the class's name. Keyword `class` is optional. When this is the case, the friend class has access to private members defined within the other class i.e. all member functions of the friend class are friend functions of the class that bestows the friendship.

When one class is a friend of another class, it only has access to the members within the other class. It does not inherit the other class. The members of the first class do not become members of the friend class. The friendship is unidirectional by default.

## Example

### ➤ Code Snippet

```
class friendclass
{
public:
    int subtractfrom(int x)
    {
        CPP_Tutorial var2;
        return var2.private_data - x;
    }
};

void main()
{
    friendclass var3;
    cout << "Added Result for this C++ tutorial: "<<
    var3.subtractfrom(2);
}
```

## Demo: Friend Class

➤ friend\_class.cpp



January 08, 2008 Proprietary and Confidential 100

 Capgemini  
CENTRALISATION TECHNOLOGIES INTEGRATION

## Demo : friend\_class.cpp

```
#include <iostream>
using namespace std;
class CSquare;
class CRectangle {
    int width, height;
public:
    int area ()
    {return (width * height);}
    void convert (CSquare a);
};
class CSquare {
private: int side;
public:
    void set_side (int a)
    {side=a;}
    friend class CRectangle;
};
```

```
void CRectangle::convert (CSquare a) {  
    width = a.side;  
    height = a.side;  
}  
  
int main () {  
    CSquare sqr;  
    CRectangle rect;  
    sqr.set_side(4);  
    rect.convert(sqr);  
    cout<<rect.area();  
    return 0;  
}
```

(Contd.....)

When one class is a friend of another class, it only has access to the members within the other class. It does not inherit the other class. The members of the first class do not become members of the friend class i.e. friendship is unidirectional by default

### 6.3. Namespace

- Used to avoid Namespace collision
- Allows you to use the same name in different contexts.
  - Avoids subsequent conflicts arising.
  - \* Example:

```
//vendor1.h  
class String {  
    ...  
};
```

```
//vendor2.h  
class String {  
    ...  
};
```

January 08, 2006 | Proprietary and Confidential | 101 |



#### Namespaces

A namespace is a section of code, delimited and referred to using a specific name. A namespace is created to set apart a portion of code with the goal to reduce, otherwise eliminate, confusion. This is done by giving a common name to that portion of code so that, when referring to it, only entities that are part of that section are recognized.

A namespace is not a variable. It is neither a function, nor a class. It is a technique to name a section of code and refer to that section with a name.

Their purpose is to localize names of identifiers to avoid name collisions. Name collisions are compounded when two or more third-party libraries are used by the same program. In this case, it is likely that a name defined by one library conflicts with the same name defined by the other library. The situation can be particularly troublesome for class names. For example, if your program defines a class ThreeDCircle and a library your program uses defines a class by the same name, a conflict arises. Creation of the namespace keyword is a response to these problems. It localizes the visibility of names declared within it. Namespaces allow the same name to be used in different contexts without conflicts. Perhaps the most noticeable beneficiary of namespace is the C++ standard library.

Prior to namespace, the entire C++ library was defined in the global namespace (the only namespace). With namespace, C++ library is now defined in its own namespace, called std, reducing the possibility of name collisions. You can also create your own namespaces in your program to localize the visibility of any names you think may cause conflicts. This is especially important when you create class or function libraries.

## Namespace

```
#include "vendor1.h"
#include "vendor2.h"
```

➤ This usage in a program triggers a compilation error as class String is defined twice.  
— Solve this problem by using Namespace.

```
namespace Vendor1 {
    class String {
        ...
    };
}
```

```
namespace Vendor2 {
    class String {
        ...
    };
}
```

January 08, 2016 | Proprietary and Confidential | 19 |

 Capgemini  
CENTRALTECHNOLOGIES INSTITUTE

### Why Namespace?

What problem do namespaces solve? Suppose you buy two different general-purpose class libraries from two different vendors. Each library has some features you wish to use. You include headers for each class library.

### In the above example:

This usage triggers a compiler error, because class String is defined twice. In other words, each vendor has included a String class in the class library, that leads to a compile-time clash. Even if you somehow get around this compile-time problem, there is a further problem of link-time clashes, where two libraries contain some identically-named symbols.

## 6.4. Using Keyword

### ➤ Syntax

```
using namespace name;  
using name::member
```

### ➤ Example

```
using SampleNameSpace::lowerbound;  
// only lowerbound is visible  
lowerbound = 10;  
// OK because lowerbound is visible  
using namespace SampleNameSpace;  
// all members are visible  
upperbound = 100;  
// OK because all members are now visible
```

January 08, 2008

Proprietary and Confidential



using

Suppose your program includes frequent references to the members of a namespace. To specify the namespace and scope resolution operator each time you refer to one quickly becomes a tedious chore. The using statement was invented to alleviate this problem. The using statement has these two general forms:

```
using namespace name;  
using name::member;
```

In the first form, name specifies the name of the namespace you wish to access. All members defined within the specified namespace are brought into view (i.e., they become part of the current namespace) and may be used without qualification. In the second form, only a specific member of the namespace is made visible. For example, assuming SampleNameSpace as shown above, the following using statements and assignments are valid:

```
using SampleNameSpace::lowerbound; // only lowerbound is visible  
lowerbound = 10; // OK because lowerbound is visible  
using namespace SampleNameSpace; // all members are visible  
upperbound = 100; // OK because all members are now visible
```

## 6.5. Example

### ➤ Code Snippet

```
int main()
{
    int val;
    std::cout << "Enter a number: ";
    std::cin >> val;
    std::cout << "This is your number: ";
    std::cout << std::hex << val;
    return 0;
}
```

January 08, 2008 | Proprietary and Confidential | v5.1



### std Namespace

Standard C++ defines its entire library in its own namespace called std.  
using namespace std;

This brings the std namespace into the current namespace. It gives you direct access to names of functions and classes defined within the library. You do not need to qualify each one with std.

Of course, you can explicitly qualify each name with "std:::" if you wish. For example, the following program does not bring the library into the global namespace.

```
// Use explicit std:: qualification. #include <iostream>
```

In the above Example :

Here, cout, cin, and the manipulator hex are explicitly qualified by their namespace. That is, to write to standard output, you must specify std::cout; to read from standard input, you must use std::cin; and the hex manipulator must be referred to as std::hex. You may not wish to bring the standard C++ library into the global namespace if your program makes only limited use of it. However, if your program contains hundreds of references to library names, then it is easier to include std in the current namespace than qualifying each name individually.

## 6.5. Demo: Namespace

➤ **namespace.cpp**



January 08, 2008 | Proprietary and Confidential | 16 |

Demo : namespace.cpp

```
#include <iostream>
using namespace std;
namespace Mine
{
    int a;
}
int main()
{
    Mine::a = 140;
    cout << "Value of a = " << Mine::a * 2 << endl;
    return 0;
}
```

Lab

➤ Lab 4



January 08, 2008 | Proprietary and Confidential | 17 /

 Capgemini  
CONSULTING TECHNOLOGY SERVICES

## Summary

- **Friend function:**
  - Usually called with at least one argument.
    - Argument comprises instance of the class, that bestows the friendship.
- **Class**
  - Allows non-member functions and other classes to access its own private data, by making them as friends.
- **Namespaces**
  - Avoid class name collision.
  - Standard C++ defines its entire library in its own namespace "std".



### Review Question

- When one class is a friend of another class it actually inherits the other class.
  - True
  - False
- The entire C++ library is defined in \_\_\_\_\_
- How do the following two statements differ?
  - using namespace A
  - using namespace A : B



# C++

## Lesson 7: Operator Overloading

January 18, 2016 | Proprietary and Confidential | -1-



## Lesson Objectives

➤ To understand:

- How to overload different types of operators:
  - Unary
  - Binary
- Use of friend and member functions to overload operators
- Restrictions on operator overloading



January 26, 2016 | Proprietary and Confidential | - 2 -



### Lesson Objectives

This lesson covers about overloading unary and binary operators. The lesson contents are:

#### Lesson 7: Operator Overloading

- 7.1. Operator Overloading
- 7.2. Operator Functions
- 7.3. Unary Operator Overloading
- 7.4. Binary Operator Overloading
- 7.5. Restrictions on Operator Overloading

## Introduction

➤ **Operator Overloading**

- Ability to tell the compiler how to perform a certain operation when a corresponding operator is used on one or more user-defined data types.
- Example:
  - -48              negative value
  - 80-712           Subtraction
  - 558-9-27        Subtraction
  - --sum            pre decrement
  - sum-            post decrement

January 18, 2016 | Proprietary and Confidential | - 3 -

 Capgemini  
www.capgemini.com

## Introduction

- When you create a new class, you need to be able to create new operators specifically designed to operate upon data members of that class.
  - Overload the existing operators.
  - Meaning you give to the operator should make sense to a reader of your code.

January 16, 2016 | Proprietary and Confidential | + 4 +



Whenever a new class is created, you need the ability to create new operators specifically designed to operate upon data members of that class. For this, you need to overload existing operators.

Although most of the existing operators have no inherent meaning in regard to any user-defined class, certain operators still work with any new class. For example the '&' still yields the address of an instance of some class type, while the '=' (by default) causes all data members of one instance of a class to be assigned to the corresponding data members of another instance of the same class. In addition, the direct member operator (dot) and sizeof operator are automatically valid.

Whatever meaning you give to the operator should make sense to someone reading your code. For example, obviously you should not overload the '+' operator to mean any kind of subtraction.

The slide has a header 'Introduction Requirements of Overloaded Operators'. Below the header is a bulleted list of requirements:

- **Intuitive Meaning:**
  - “+” represents Addition
- **Associative as appropriate:**
  - a + b
- **Result in an object, c of the same class.**
- **If these conditions are not satisfied use Member Functions instead.**

At the bottom of the slide, there is a footer bar with the text 'January 08, 2010 Proprietary and Confidential 18 / 84' and the Capgemini logo.

#### Rules for Overloading an Operator

- Only operators in the C++ precedence chart can be overloaded.
- You cannot rebuke pre-defined operator precedence rules. For example, binary ‘+’ cannot get a higher precedence than binary ‘\*’.
- No default arguments are allowed in overloaded operator functions.
- As with pre-defined operators, overloaded operators may be *unary* or *binary*. If it normally is unary, then it cannot be defined as binary. If it is normally binary, then it cannot be unary. However, if operators can be unary and binary, then you can overload them in either or both ways.
- Operator function for a class may be either a *non-static member* or *global friend function*.
- At least one argument to the overloaded function (implicit or explicit) must be an instance of the class to which the operator belongs.
- You cannot overload following operators:
  - direct member      \* direct pointer-to-member
  - :: scope resolution      ?: ternary
- Overload following operators as member and not friend functions:
  - = assignment      () function call
  - [] subscript      -> indirect member operator
- Following operators are normally overloaded as friend functions, not as members:
  - << as a stream operator
  - >> as a stream operator

### Forms of Overloaded Operators

- Member Functions
- Friend Functions

Overloading Type	Unary (1 Argument)	Binary (2 Arguments)
Member	1 implicit, 0 explicit	1 implicit, 1 explicit
Friend	0 implicit, 1 explicit	0 implicit, 2 explicit

Industry 4.0, Intel® Proprietary and Confidential © Capgemini 2018

 Capgemini CONSULTING TECHNOLOGIES INTEGRATING

**Member functions:** When you overload a unary operator using as a member function, you do not send any argument to the operator function. This is because the function gets access to the calling object through its `this` pointer.

**Friend function:** As the friend function is a non-member function, the object needs to be explicitly passed to it. Note in the above program, the object `s1` is passed by reference to the operator function. This is to ensure that the modifications done inside the function are applicable to the actual object.

- If a function is a member function, then `this` is implicitly available for one of the arguments.
- When an operator function is implemented as a member function, the left-most (or only in the case of unary operators) operand must be a class object (or a reference to a class object) of operator's class.  
**e.g.: `a + b` // a is an object**
- If the left operand must be an object of a different class or a built-in type, this operator must be implemented as a non-class member.  
**e.g. `<<, >>` operators**
- An operator function implemented as a non-member must be a friend if it needs to access non-public data members of that class.

- The overloaded `<<` operator must have a left operand of type `ostream`. Therefore, it must be a non-member function. Also, it may require access to the private data members of the class. Thus, it needs to be a friend function for that class.
- A similar observation holds true for `>>` operator which has a left operand of type `istream`.
- Operator member functions are classed only when the left operand of a binary operator is specifically an object of that class or when the single operand of a unary operator is an object of that class.
- If the operator needs to be commutative ( $a + b = b + a$ ), then making it a non-member function is necessary.

## Unary Operator Overloading

- Operate on only one operand.
  - Example: `++`, `-!`, `-`, `- unary minus.`
- Syntax

```
return_type classname :: operator operator symbol(argument)  
{  
.....  
statements;  
}
```

### Demo: Unary Operator Overloading

- unary\_member.cpp
- unary\_friend.cpp



January 26, 2016 | Proprietary and Confidential | + 9 +



#### Demo : unary\_member.cpp

```
#include<iostream>
using namespace std;
class number {
    int n;
public:
    number(int x = 0):n(x){}
    number operator-()
    {
        return number(-n);
    }
}
int main() {
    number a(1), b(2), c, d;
    d = -b;           //d.n=-2
    c = a.operator-(); //c.n = -1
    return 1;
}
```

**Demo : unary\_friend.cpp**

```
#include<iostream>
using namespace std;
class number {
    int n;
public:
    number(int x = 0):n(x){}
    friend number operator-(number &num) {
        return (-num.n);
    }
    void display(){
        cout<<"\nNumber :"<<n;
    }
};
int main() {
    number a(1), b(2), c, d;
    d = -b;           //d.n=-2
    c=operator-(a); //c.n = -1
    d.display();
    c.display();
    return 1;
}
```

As the friend function is a non-member function, the object needs to be explicitly passed to it. Note in the above program, the object s1 is passed by reference to the operator function. This is to ensure that the modifications done inside the function are applicable to the actual object.

**Overloading increment operators (post increment and pre-increment)**

```
#include<iostream>
using namespace std;
class sample {
public:
    int a ;
    sample () : a(0) {}
    sample(int x) : a(x) {}
```

Overloading increment operators(post increment and pre-increment) Contd..

```
sample friend operator ++(sample&);  
//prefix version  
sample friend operator ++(sample&, int );  
//postfix version. int is a pseudo arg.  
void showdata() {  
    cout<<"\n A = "<<a<<endl;  
}  
sample operator ++(sample& s){//prefix  
    s.a=s.a+1; //the object is modified  
    return s; //and then returned  
}  
sample operator ++(sample& s, int ){//postfix  
    sample tmp=s; // original object saved as tmp  
    s.a=s.a+1; //object is modified  
    return tmp; //tmp is returned  
}  
void main(){  
    sample s1(10);  
    sample s2;  
    s2=++s1; //prefix version invoked  
    s2.showdata(); //Output : A = 11  
    s1.a=10;  
    sample s3;  
    s3=s1++; //postfix version invoked  
    s3.showdata(); //Output : A = 10  
}
```

To differentiate between the prefix and postfix versions of overloaded increment and decrement operators, we need to make use of a pseudo argument. The compiler uses the pseudo argument to distinguish between the prefix and postfix increment operators.

You can overload the prefix increment operator `++` with either a friend function operator that has one argument of class type or a reference to class type, or with a member function operator that has no arguments.

The postfix increment operator `++` can be overloaded for a class type by declaring a friend function operator `operator++()` with two arguments, the first having class type and the second having type `int`. Alternatively, you can declare a member function

operator operator++() with one argument having type **int**.

## Binary Operator Overloading

- **Binary Operator Overloading is also done with member as well as friend functions.**
  - Member functions take one argument.
  - Friend functions takes two arguments.

## Demo: Binary Operator Overloading

- binary\_member.cpp
- binary\_friend.cpp



January 26, 2016 | Proprietary and Confidential | +13 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Demo : binary\_member.cpp

```
#include<iostream>
using namespace std;
class sample {
    int a,b;
public: sample(){}
    sample(int a,int b){
        this->a=a;
        this->b=b;
    }
    sample operator+(sample s);
    void disp(){
        cout<<"a= "<<a<<"\t b= "<<b<<endl;
    }
};
```

**Demo : binary\_member.cpp (Contd..)**

```
sample sample :: operator +(sample s)
{
    sample s1;
    s1.a=s.a+a;
    s1.b=s.b+b;
    return s1;
}
int main()
{
    sample s1(10,2),s2(5,1),s3;
    s3=s1+s2;
    s3.disp();
    return 0;
}
```

**Demo : binary\_friend.cpp**

```
#include<iostream>
using namespace std;
class sample {
    int a,b;
public: sample(){}
    sample(int a,int b){
        this->a=a;
        this->b=b;
    }
    friend sample operator+(sample s1,sample s2);
    void disp(){
        cout<<"a= "<<a<<"\t b= "<<b<<endl;
    }
};
sample operator +(sample s1,sample s2){
    sample s3;
    s3.a=s1.a+s2.a;
    s3.b=s1.b+s2.b;
    return s3;
}
```

Demo : binary\_friend.cpp(Contd..)

```
int main()
{
    sample s1(10,2),s2(5,1);
    s1=s1+s2;
    s1.disp();
    return 0;
}
```

Overloading <<(insertion) operator

```
#include <iostream>
#include <cstring>
using namespace std;
class Employee {
    char name[80];
    int empcode;
    int prefix;
    char *BU;
public:
    Employee(char *n, int a, int p, char * nm) {
        strcpy(name, n);
        empcode = a;
        prefix = p;
        strcpy(BU,nm);
    }
    friend ostream &operator<<(ostream &stream, Employee o);
};
// Display name BU.
ostream &operator<<(ostream &stream, Employee o){
    stream << o.name << " ";
    stream << "(" << o.empcode << ")";
    stream << o.prefix << "-" << o.BU << "\n";
    return stream; // must return stream
}
```

Overloading <<(insertion) operator(Contd..)

```
int main(){
    Employee a("A", 111, 555, "PES");
    Employee b("B", 312, 555, "PACE");
    Employee c("C", 212, 555, "IMS");
    cout << a << b << c;
    return 0;
}
```

### Restrictions on Overloaded Operators

- New operators cannot be created.
- Fundamental Data Types (e.g. int) cannot be overloaded.
- Operator priority cannot be changed.
- The number of arguments of cannot be changed.  
    -- + can take only two arguments (there is no unary +).
- Two separate overloaded functions (with different signatures) can be created for operators which exist in pre-fix and post-fix form -- ++.

January 08, 2010 | Proprietary and Confidential | 17 / 41



Overloaded operators are not implicitly *associative* or *commutative*. Even if the original operators were associative or commutative, associativity and commutativity must be explicitly implemented. For associativity, this means returning an instance of the class.

If you overload the operator “+”, it does not automatically overload related operators (+=, ++, etc). If needed, these related operators can be explicitly overloaded.

**Lab**

➤ **Lab 5**



January 08, 2016 | Proprietary and Confidential | 10 / 10

**Capgemini**  
www.capgemini.com

## Summary

- Operator overloading is the ability to create new operators specifically designed to operate upon the data members of a new class.
- Operators can be overloaded using either member or friend function.
- Operator overloading function should not change the meaning of the operator.



### Review Question

- Precedence associated with an operator can be changed using operator overloading.
  - True /False
- How will you overload “+” operator to operate upon two objects of the same class?
  - A. friend function
  - B. member function
- Can we overload all the available operators?
  - Yes / No



## C++

### Lesson 8: Inheritance

January 26, 2016 | Proprietary and Confidential | 4 / 1



## Lesson Objectives

➤ To understand:

- Inheritance:
  - Mechanism
  - Advantages
  - Types:
    - single, multiple, multilevel, hybrid.
  - Sequence of constructors invoked.
  - Need of virtual base class.



January 28, 2016 | Proprietary and Confidential | + 2 +

 Capgemini

### Lesson Objectives

This lesson covers about inheritance and types of inheritance. The lesson contents are:

#### Lesson 8: Inheritance

- 8.1. Inheritance Concepts
- 8.2. Inheritance Types
- 8.3. Access Specifiers
- 8.4. Multilevel Inheritance
- 8.5. Multiple Inheritance
- 8.6. Constructors in Derived Class
- 8.7. Hybrid Inheritance
- 8.8. Virtual Base Class

## Introduction

### ➤ Inheritance:

- Mechanism to derive a new class from an existing class.
- Derived Class:
  - Class which derives characteristics.
- Base Class:
  - Class from which it derives is the base class.
- Derived class inherits all capabilities of the base class
- It can also add refinements of its own.
- Base class is unchanged by this process.

January 26, 2016 | Proprietary and Confidential | 13 \*



Inheritance is a process by which one class acquires the members of an existing class. It describes a hierarchical relationship between classes. For example, a lion is a kind of animal. Now talking in terms of object-oriented programming, you can say that Lion is a subclass of Animal, and Animal is a super class of Lion.

With inheritance, a class inherits the general attributes and behaviors from its super class, and defines only those members that make it unique. Inheritance exposes, extends, or alters the attributes and behaviors of the super class. A subclass can extend but cannot narrow its super class.

Mechanism to derive a new class from the existing class is called inheritance.

Inheritance is the process to create new classes called derived classes from base classes. Derived class inherits all capabilities of the base class, but can also add refinements of its own. Base class is unchanged by this process.

Derived class inherits all capabilities of the base class, but can also add refinements of its own. Base class is unchanged by this process.

Inheritance allows you to collect related classes into a hierarchy with the classes at the top, to serve as abstractions for those below. This implies that the derived class is a specialization of its parent class. In other words, the derived class is a type of base class, but with more detail added.

## Introduction

Inheritance creates a hierarchy of related classes (types) which share code and interface.

```
graph TD; Card[Card] -- "inherits from (ISA)" --> Visa[Visa]; Card -- "inherits from (ISA)" --> MasterCard[Master card]; Card -- "inherits from (ISA)" --> AmericanExpress[American Express]; Visa -- "hologram" --> Visa; MasterCard -- "pin" --> MasterCard; AmericanExpress -- "category" --> AmericanExpress;
```

The diagram illustrates inheritance. A base class 'Card' is shown at the top, with three derived classes below it: 'Visa', 'Master card', and 'American Express'. Arrows point from each derived class back to the base class 'Card', labeled 'inherits from (ISA)'. Additional arrows point from each derived class to itself, labeled 'hologram' (under Visa), 'pin' (under Master card), and 'category' (under American Express). Labels 'logo' and 'owner's name' are also associated with the 'Card' class.

January 26, 2016 | Proprietary and Confidential | + 4 +

Capgemini

In order to derive a class from another, use a colon (:) in the derived class declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name  
{ /*...*/};
```

Here, `derived_class_name` is the name of the derived class and `base_class_name` is the name of the class on which it is based.

public access specifier may be replaced by protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class. The visibility-mode specifies the privileges that the derived class has as far as access to members of its parent class are concerned.

## Inheritance types

➤ **Different types of Inheritance are as follows:**

- Single Inheritance
- Multiple Inheritance
- Multi-level Inheritance
- Hybrid Inheritance

January 26, 2016 | Proprietary and Confidential | 45 \*



### Single Inheritance

Single inheritance is the one where you have a single base class and a single derived class.

### Multiple Inheritance

Multiple inheritance means that one subclass can have more than one superclass. This enables the subclass to inherit properties of more than one superclass and to “merge” their properties. Multiple inheritance is achieved whenever more than one class acts as base classes for other classes. This makes the members of the base classes accessible in the derived class, resulting in better integration and broader re-usability.

### Multi-level Inheritance

Multilevel inheritance is the mechanism by which one class derives its characteristic from the base class and another class is inherited from the derived class. For instance, a class A serves as a base class for class B which in turn serves as base class for another class C. The class B, which forms the link between the classes A and C, is known as the intermediate base class.

### Hybrid Inheritance

It is a combination of multiple and multi-level inheritance.

## Deriving a Class

### ➤ Syntax

```
class derived-class : visibility-mode base-class{  
    // derived class members  
}
```

### ➤ Visibility-modes are

- public
- Private
- protected

January 26, 2016 | Proprietary and Confidential | 46 / 46



Visibility-mode specifies privileges that the derived class has as far as access to members of its parent class are concerned. To control the access of a class member, you use one of the access specifiers public, private, or protected as a label in a class member list.

Members of classes declared with the keyword class are private by default. Members of classes declared with the keyword struct or union are public by default.

## Demo: Deriving a Class

> InheritDemo1.cpp



```
class Employee;
class Payslip : public Employee;
```

January 28, 2016 | Proprietary and Confidential | +2+

Capgemini

## Demo : single.cpp

```
#include<iostream>
using namespace std;
class Employee{
    char *emp_id;
    char* emp_name;
    float salary;
public:
    Employee(){}
    salary=10000;
}
void get_info();
void print_info();
float get_salary();
};

void Employee :: get_info(){
    cout<<"Employee Id:";cin>>emp_id;
    cout<<"Employee Name:";gets(emp_name);
}
```

## Demo: Deriving a Class

InheritDemo2.cpp



```
student
  ^
grad_student
```

January 26, 2016 Proprietary and Confidential 8 Capgemini

## Demo : InheritDemo1.cpp(Contd..)

```
float Employee :: get_salary(){
    return salary;
}
void Employee :: print_info(){
    cout<<"\nEmployee Id:"<<emp_id;
    cout<<"\nEmployee Name:"<<emp_name;
}
class Payslip:public Employee{
    float da,pf,net_salary;
    public:
        Payslip(){
            da=0;
            pf=0;
            net_salary=0;
        }
        float calculate_salary(){
            da=2.5*get_salary()/100;
            pf=8*get_salary()/100;
            net_salary=get_salary()-(da+pf);
            return net_salary;
        }
        void print_salarySlip(){
            print_info();
            cout<<"\n Net Salary:"<<calculate_salary();
        }
}
int main(){
    Payslip p1;
    p1.get_info();
    p1.print_salarySlip();
    return 0;
}
```

Demo:InheritDemo2.cpp

```
class student {  
public:  
    student(char* nm, int id, int y);  
    void print();  
  
private:  
    int student_id;  
    int year;  
    char name[30];  
};  
student::student(char* nm, int id, int y) {  
    student_id = id;  
    year = y;  
    strcpy(name, nm);  
}  
void student::print() {  
    cout << "\n" << name << ", " << student_id << ", "  
    << year << endl;  
}  
class grad_student: public student {  
public: grad_student(char* nm, int id, int y, char* d, char* th);  
    void print();  
private: char dept[10];  
    char thesis[80];  
};  
grad_student::grad_student(char* nm,int id, int y, char* d, char* th)  
:student(nm, id, y){  
    strcpy(dept, d);  
    strcpy(thesis, th);  
}  
void grad_student::print() {  
    student::print();  
    cout << dept << ", " << thesis << endl;  
}  
int main() {  
    student s1("Jane Doe", 100, 1);  
    grad_student gs1("John Smith", 200, 4, "Pharmacy", "Retail Thesis");  
    cout << "Student classes example:\n";  
    cout << "\n Student s1:";  
    s1.print();  
    cout << "\n Grad student gs1:";  
    gs1.print();  
}
```

**Overriding Member functions**

You can use functions in a derived class that have the same name and same signature as those in the base class. You might want to do this so that calls in your program work in the same way for objects of both base and derived classes and to add more functionality.

If an instance or member function of the derived class refers to a data member of the base class that has been overridden, the member from the derived class is used by default. To refer to the member function of the base class that has been overridden in the derived class, base class name and the scope resolution operator are required. Refer to the following example:

**base-classname::member-function-name.**

Access Specifiers		
private	protected	Public
Always inaccessible regardless of derivation access	Private in derived class if you use private derivation	Private in derived class if you use private derivation
	Protected in derived class if you use protected derivation	Protected in derived class if you use protected derivation
	Protected in derived class if you use public derivation	Public in derived class if you use public derivation

January 28, 2016 | Proprietary and Confidential | 10 |



Let us now see how private, protected and public members of a class fall into various sections of a derived class when inherited with different access specifiers.

#### Private Derivation

If a new class is derived privately from its parent class, then private members, inherited from the base class, are inaccessible to new member functions in the derived class.

Public members inherited from the base class have private access privilege. They are treated as though they were declared as new private members of the derived class, so that new members functions can access them. However, if another private derivation occurs from this derived class, then these members are inaccessible to new member functions.

**Example: Base class inherited as private**

```
class base{
    int x;
public:
    void setx(int n) { x = n; }
    void showx(void){
        cout << "x = " << x << "\n";
    }
};

// Private Inheritance
class derived : private base{
    int y;
public:
    // setx() is accessible from within derived
    void setxy(int n, int m){
        setx(n);
        y = m;
    }
    // showx() is accessible from within derived
    void showxy(void){
        showx();
        cout << "y = " << y << "\n";
    }
};
void main() {
    derived ob;
    ob.setxy(10, 20);
    ob.showxy();
}
```

**Output:**

x=10  
y=20

**Public Derivation**

Public derivations are much more common than private derivations. If a new class is derived publicly from its parent class, then private members inherited from the base class are inaccessible to new member functions in the derived class.  
Public members inherited from the base class may be accessed by new member functions in the derived class and by instances of the derived class.

Example: public\_in.cpp  
The derived class inherits the members of the \*\* base class as public members

```
class base{  
    int x;  
public:  
    void setx(int n) { x=n; };  
    void showx(void) {  
        cout << "x = " << x << endl;  
    }  
};  
// Public Inheritance  
class derived : public base{  
    int y;  
public:  
    void sety(int n) { y=n; };  
    void showy(void) {  
        cout << "y = " << y << endl;  
    }  
};  
int main() {  
    derived ob; ob.setx(10);  
    // access member of base class  
    ob.sety(20);  
    // access member of derived class  
    ob.showx();  
    // access member of base class  
    ob.showy();  
    // access member of derived class  
    return 0;  
}
```

**Output:**  
x=10  
y=20

#### **Protected Access Rights**

Declaring the data members as private is much too restrictive because new member functions in the derived class need to gain access to it. This is possible by using an access specification called protected.

The members of a class that have been defined as protected can be accessed from within the class and from the derived classes. However, as far as instance of the derived class are concerned, protected and private are one and the same, so direct access is not possible.

```
class sample {
    {
private :
    //Visible to member functions within the class.
protected:
    //Visible to member functions of its own class and
    derived classes.
public :
    //Visible throughout the program.
};
```

**Example: Protected Members inherited as public**

```
class base {
protected:
    int a, b; // private to base
public:
    void setab(int n, int m) { a=n; b=m; }
};

class derived : public base {
    int c; public: void setc(int n) { c=n; }
    // this function has access to a and b from base
    void showabc() {
        cout << "a = " << a << ' ' << "b = " << b
        << ' ' << "c = " << c << "\n";
    }
};

void main() {
    derived ob;
    /* a and b are not accessible here because they are
    ** private to both base and derived */
    ob.setab(1, 2);
    ob.setc(3);
    ob.showabc();
}
```

**Output:**

```
a=1
b=2
c=3
```

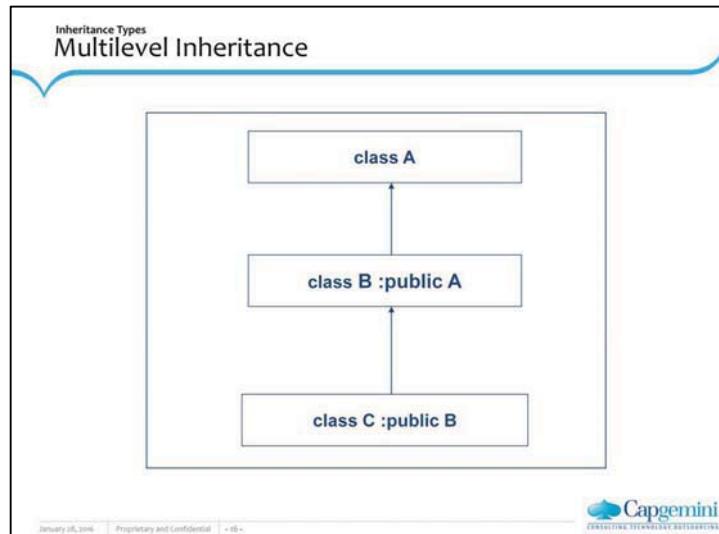
**Protected derivation**

The only difference between a public and a protected derivation is how the public members of the parent class are inherited.

*Private* members inherited from the base class are inaccessible to new member functions in the derived class.

*Protected* members inherited from the base class have protected access privilege.  
*Public* members inherited from the base class have protected access privilege.

**Note:** The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.



In multilevel inheritance, a class inherits from a derived class (or subclass). For example, if class C inherits from class B, and class B inherits from class A, class C will acquire all the members declared in class B as well as all those declared in class A.

Demo: multilevel.cpp

```
class A {  
protected:int x;  
public:  
    void showA(){  
        cout<<"enter a value for x:"<<endl; cin>>x;  
    }  
};  
class B: public A {  
protected: int y;  
public:  
    void showB(){  
        cout<<"enter value for y:"; cin>>y;  
    }  
};
```

Demo : multilevel.cpp(Contd..)

```
class C: public B {  
    public:  
        void showC() {  
            showA();  
            showB();  
            cout<<"x*y ="<<x*y;  
        }  
};  
void main() {  
    C ob1;  
    ob1.showc();  
}
```

Inheritance Types

## Multiple Inheritance

➤ Implies that a derived class has more than one parent or base classes.  
— Class is derived from more than one base class.

```
graph TD; classX[class X] --> classZ["class Z :public X,public Y"]; classY[class Y] --> classZ;
```

January 26, 2016 | Proprietary and Confidential | +18+

Capgemini

A class can be derived from more than one base classes. This is called Multiple Inheritance. It implies that a derived class has more than one parent or base class.

In multiple inheritance, a class inherits from several super classes. For example, class Z inherits from both class X and class Y. It acquires all members declared in classes X and Y.

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration.

Demo: multiple.cpp

```
class Base1 {  
public:  
    Base1( int parameterValue )  
    {  
        value = parameterValue;  
    }  
    int getData() const  
    {  
        return value;  
    }  
}
```

Demo : multiple.cpp(Contd..)

```
protected:  
int value;  
};  
class Base2{  
public:  
    Base2( char characterData ) {  
        letter = characterData;  
    }  
    char getData() const {  
        return letter;  
    }  
protected:  
    char letter;  
};  
class Derived : public Base1, public Base2{  
public:  
    Derived( int integer, char character, double double1 )  
        : Base1( integer ), Base2( character ), real( double1 ) {}  
    double getReal() const {  
        return real;  
    }  
    void display(){  
        cout << " Integer: " << value << "\n Character: "  
            << letter << "\nReal number: " << real;  
    }  
private:  
    double real;  
};  
int main(){  
    Base1 base1( 10 ), *base1Ptr = o;  
    Base2 base2( 'Z' ), *base2Ptr = o;  
    Derived derived( 7, 'A', 3.5 );  
  
    cout << base1.getData()  
        << base2.getData();  
  
    derived.display();  
  
    cout << derived.Base1::getData()  
        << derived.Base2::getData()  
        << derived.getReal() << "\n\n";  
    return o;  
}
```

## Constructors in derived class

- Base class constructors are called first.
- In multiple inheritance, base classes are constructed in the order of appearance in the derived class declaration.
- In multilevel inheritance, constructors are executed in the order of inheritance.

January 28, 2016 | Proprietary and Confidential | 20+



The derived class instance always contains data members inherited from the base class. Therefore, whenever a derived class is instantiated, the data members from both; the base class and derived class must be initialized. This requires two constructors to be executed; one for the base class and another for the derived class.

The compiler takes care of this automatically and calls both constructors. As soon as the derived class constructor gets control and establishes its formal arguments, the base class constructor is called immediately i.e. before the derived class initialization is done.

In principle, a derived class inherits every member of a base class except:

its constructor and its destructor  
its operator=() members  
its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

## Background activities-object creation

- Space is allocated (on a stack or heap) for the full object.
- Base class constructor initializes data inherited from the base class.
- Derived class constructor initializes data added in the derived class. The object is then usable.
- When the object is destroyed (goes out of scope or is deleted):
  - Derived class destructor is called on the object.
  - Base class destructor is called on the object.
  - Finally, the allocated space for the full object is reclaimed.

January 26, 2016 | Proprietary and Confidential | +31+



Space is allocated (on the stack or the heap) for the full object (enough space to store data members inherited from the base class and those defined in the derived class).

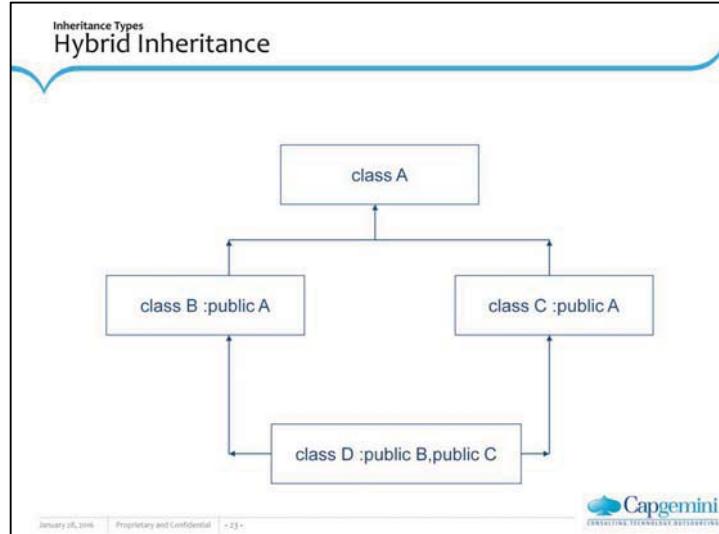
Base class constructor is called to initialize data members inherited from the base class.

Derived class constructor is then called to initialize data members added in the derived class. The derived-class object is then usable

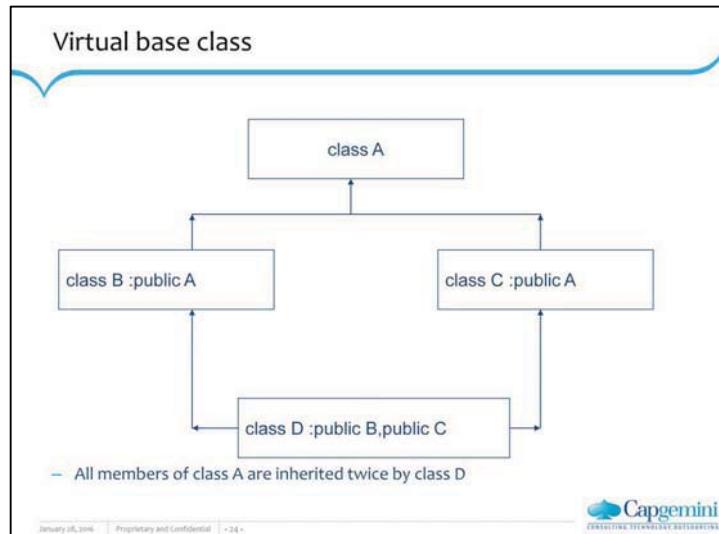
When the object is destroyed (goes out of scope or is deleted) the derived class destructor is called on the object first. Then, the base class destructor is called. Finally the allocated space for the full object is reclaimed.

Demo : cons\_order.cpp(Contd..)

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



Hybrid inheritance is the combination of multiple and multilevel inheritance. Following example illustrates hybrid inheritance. Class B and Class C are inherited from Class A. Class D is inherited from Class B and Class C. The typical problem arises in this scenario. Members of Class A are inherited twice in Class D i.e. via classes B and C. This creates an ambiguous situation. The problem is often referred to as 'Dreaded Diamond' problem.



#### Ambiguities in Multiple Inheritance

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. If you have a class, which is derived from two parent classes, both of which have a function with a common name. In such cases, use the class name followed by the scope resolution operator while calling the function.

In the diagram shown above, class D inherits the characteristics of class A twice, once from class B and then from class C. This means there are 2 copies of A present in an object of type D. This leads to confusion. Concept of virtual class helps remove this ambiguity. Let us try to understand this concept with an example.

To overcome the dreaded diamond problem, the super-most base class is made a virtual base class. Only one copy of members of class A is now inherited by class D.

In order to declare a member of a class as virtual, we must precede its declaration with the keyword `virtual`.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in a derived object. Declare the base class as 'Virtual' when it is inherited. To accomplish this, precede the base class name with the keyword virtual.

**Demo : virtual\_base.cpp**

```
class base {  
public:  
    void print{ cout << "Base class function"; }  
};  
  
class derived1 : virtual public base {  
public:  
    derived1() { cout << "Constructing derived1\n"; }  
    ~derived1() { cout << "Destructing derived1\n"; }  
};  
  
class derived2: virtual public base {  
public:  
    derived2() { cout << "Constructing derived2\n"; }  
    ~derived2() { cout << "Destructing derived2\n"; }  
};  
  
class derived3: public derived1, derived2 {  
public:  
    derived2() { cout << "Constructing derived2\n"; }  
    ~derived2() { cout << "Destructing derived2\n"; }  
};  
  
int main(){  
    derived3 ob;  
    ob.print();  
    return 0;  
}
```

Try executing the above program first without the keyword virtual and analyze the results and then execute it using the keyword virtual.

## Summary

➤ **Inheritance:**

- Derive a new class from the existing class.
- Types of Inheritance:
  - Single, multiple, multilevel, hybrid

➤ **Virtual Base class:**

- Resolves an element of ambiguity.
- Introduced into a C++ program when multiple base classes are inherited.



## Review Question

- **Question 1: Inheritance is a way to:**
  - Option 1: Organize data.
  - Option 2: Reuse Data.
  - Option 3: Improve data hiding and data encapsulation.
- **Question 2: When you derive from a private base class, the base class public, protected and private members become private members of the derived class.**
  - Option 1: True
  - Option 2: False



### Review Question

- **Question 3:** A base-class initializer must always be provided in the derived-class constructor to call the base-class constructor.
- Option 1: True
  - Option 2: False



## C++

### Lesson 9: Polymorphism

January 08, 2016 Proprietary and Confidential - 1 -



## Lesson Objectives

➤ **To understand:**

- Polymorphism Types
- Pointers to objects
- Pointers to derived classes
- Virtual Functions:
  - How do virtual functions work?
  - Rules-virtual functions
  - Virtual destructor functions
- Abstract Base Class



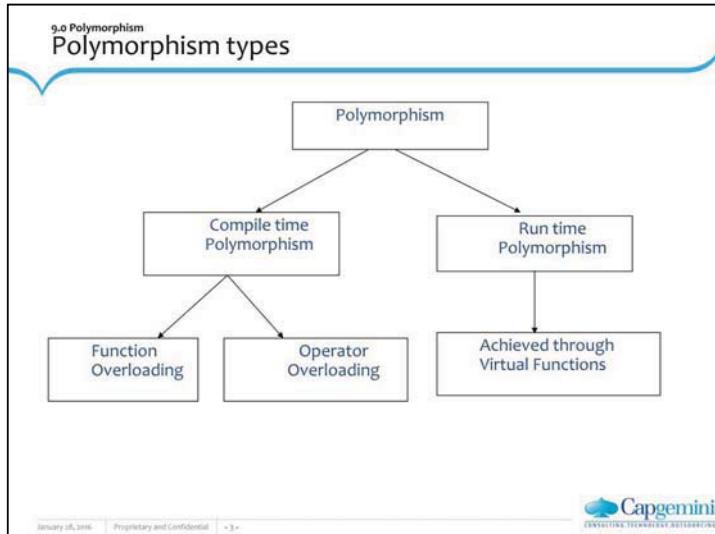
January 08, 2016 Proprietary and Confidential - 3 -

### Lesson Objectives

This lesson covers about polymorphism. The lesson contents are:

#### Lesson 9: Polymorphism

- 9.1. Polymorphism Types
- 9.2. Pointers to objects
- 9.3. Pointers to derived Class
- 9.4. Virtual Functions
- 9.5. Virtual Destructor Functions
- 9.6. Pure virtual Functions
- 9.7. Abstract Base Class



Literally the word polymorphism means the ability to take multiple forms. (Poly – many , morphism – ability to take forms ).

Polymorphism is the ability to use an operator or function in different ways. It gives different meanings or functions to the operators or functions. Poly, refers to many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

In C++, polymorphism can be achieved in two ways i.e. static and dynamic polymorphism.

Compile-time polymorphism is implemented in C++ using overloaded functions and operators. The overloaded member functions are selected to invoke by matching arguments at compile-time i.e. code associated with a call is decided at the time of compilation itself. This is called static or early binding.

When you use a pointer of type base class, the only class members that you can reference are of the base class, even if the pointer points to the derived class object.

Demo : ptr\_obj.cpp

```
class sample{  
    private :  
        int l,m;  
    public :  
        sample(int a=0,int b=0){  
            l = a; m = b ;  
        }  
        void display(void){  
            cout << "l = " << l << "\n" << m << "\n" ;  
        }  
};  
void main(void){  
    sample x(2,3);  
    sample *p = &x;  
    x.display();  
    p->display();  
    (*p).display();  
}
```

Demo : ptr\_derived.cpp

```
class employee  
{  
    public : int empno;  
    public : void display(void) {  
        cout << empno ; }  
};  
class professor : public employee  
{  
    public : int exp;  
    public : void display(void) {  
        cout << "Emp No : = " << empno  
            << "Experience : = " << exp ;  
    }  
};
```

**Demo : ptr\_derived.cpp(Contd..)**

```
void main(void) {  
    employee *e,emp;  
    e=&emp; //parent pointing to it's object  
    e->empno=111;  
    e->display();  
    //calls the parent function professor pro;  
    e=&pro; // parent pointing to child object  
    e->empno=222;  
    e->exp=20; //Invalid  
    e->display(); // calls base class function  
    professor *p;  
    p=&pro;  
    //child pointing to it's own object  
    p->exp = 2;  
    p->display();  
    //calls the child function  
    //parent pointing to members of child  
    ((professor *)e)->exp=20; //Valid  
    ((professor *)e)->display();  
}
```

The output shows that every time the function from the base class is executed, the compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the pointer. The reason is that the compiler only knows about the type of the pointer itself, and nothing about the object that the pointer eventually points at during execution time. Therefore, the class member that is hard-coded is used to generate an offset address relative to the start of the base class. This is called early or static binding since the member that is referenced is bound to the type of the pointer being used.

The goal of object-oriented programming language is to send a message to whatever instance happens to be pointed at by the pointer, as opposed to the type of the pointer itself. This is important because you can control the content (address) of the pointer at execution time, whereas the type of the pointer variable itself cannot be changed once it is hard-coded. In this manner, the instance itself can figure out what to do with the message and thereby implement the concept of polymorphism.

Therefore, what is needed is a method to delay the choice of which member function (method) gets executed until execution time. Having such a method would imply that by using a single pointer and member function name, the same function call (method) can be sent to the instance that has its address stored in the pointer. If these instances implement the same function (method) differently, then different results can be obtained. Such a method is called “late or dynamic binding” because the compiler no longer makes the decision about which function gets called.

9.0 Polymorphism

## Virtual Functions

- **Dynamic (Late) Binding:**
  - Select appropriate member function at execution time.
  - Achieved by means of virtual functions.
- **Virtual specifier:**
  - indicates that a derived class object may use an alternate implementation.
    - Non-Virtual function: Statically bound; on references, pointers and objects.
    - Virtual functions: Dynamically bound on references and pointers, but not on objects.

 Capgemini  
PERFORMANT TECHNOLOGIES

January 08, 2016 | Proprietary and Confidential | - 6 -

Virtual means existing in effect but not in reality. A virtual function exists but nevertheless appears real to some parts of a program. C++ implements late binding using virtual functions.

Suppose you declare a function in a base class definition as virtual. This is declared exactly the same way, including the return type, in one or more derived classes. All calls to that function using pointers or references of type base class invoke the function that is specified by the object being pointed at, and not by the type of the pointer itself.

```
class Base
{
public:
    virtual void show()
    {
        cout << "Base : " << endl;
    }
};
```

The keyword `virtual` is written before the return type (if any) of the function in the base class definition. It cannot appear outside a class definition. It is optional when you declare the same function in a derived class. It is through the use of virtual functions that C++ achieves the object-oriented goal of polymorphism.

When same names are used both in base and derived class, then function base class need to be made virtual. When a function is made virtual, C++ determines which function to use at run-time based on the type of object pointed to by the base pointer rather than the type of the pointer.

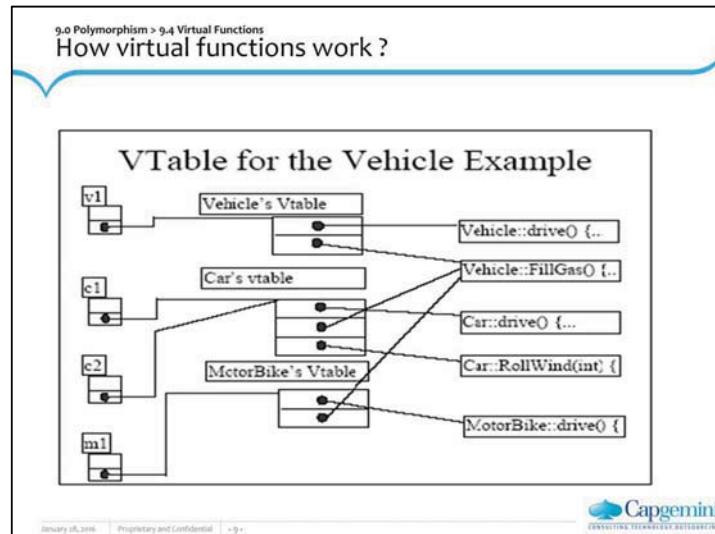
Demo :virtual\_func.cpp

```
class employee{  
    public : int empno;  
    public:employee(int n=111)  
    {           empno = n; }  
    virtual void display(void) {  
        cout << empno ;  
    }  
};  
class professor : public employee{  
    private: int exp;  
    public: professor(int e=1) {      exp = e; }  
    void display(void) {  
        cout << "Emp No : " << empno ;  
        cout << "Experience : " << exp ;  
    }  
};  
void main(void){  
    employee emp(555);  
    professor pro(2);  
    employee* e=&emp;  
    e->display(); // calls base class function  
    e=&pro;  
    e->display(); // calls derived class function  
}
```

9.0 Polymorphism > 9.4 Virtual Functions  
How virtual functions work ?

➤ Consider the following implementation

```
class Vehicle {...
    virtual void drive();
    virtual void FillGas();
}
class Car : public Vehicle {...;
    virtual void drive();
    virtual void RollWindow(int);
}
class MotorBike :public Vehicle {...;
    virtual void drive();
}
Vehicle vt;
Car c1, c2;
MotorBike m1;
```



#### How Virtual Functions Work

Late binding requires some overhead but provides increased power and flexibility. First, the compiler creates a table called the virtual function table, usually referred to as the 'vtbl'. This table contains the address of all of the virtual functions within the class. The first address occupies position 0, the next occupies position 1, and so on. For each derived class, the table is duplicated, and if a virtual function is redefined, then its address replaces the address of the corresponding function from the base class. New virtual functions, if any, are appended at the end of the table.

Whenever an instance of a class is created, the compiler reserves some extra bytes for use as a pointer to the 'vtbl'. This pointer is referred to as the 'vptr', or virtual pointer. It automatically gets initialized to point to the 'vtbl' for the class to which it belongs.

Create a pointer of type base and store into it the address of a base or derived instance. Then, execute a function using this pointer. The compiler fetches the address pointed at by 'vptr' of the invoking instance (\*this). Offset this address by the index of the function in question (0 for first, 1 for next). Fetch the address from the 'vtbl', and execute the function that this address points to.

9.0 Polymorphism > 9.4 Virtual Functions

## Rules - Virtual functions

- Access with object pointers.
- Can be a friend of another class.
- Mandatory to define a base class virtual function.
- Function prototypes in base and derived classes should be the same.
- Cannot comprise virtual constructors.
- A base pointer can point to derived object.
  - Pointer to derived class cannot access an object of base type.
- Cannot be static members.

January 08, 2016 | Proprietary and Confidential | + 10 +



### Rules – Virtual Functions:

Virtual functions cannot be static members.

They are accessed using object pointers or references.

A virtual function can be a friend of another class.

A virtual function in a base class must be defined, even though it may not be used.

The prototypes of virtual functions in base and derived classes should be the same. Otherwise, they are not treated as virtual functions.

Virtual functions cannot have virtual constructors.

A base pointer can point to any derived-type object. Pointer to a derived class cannot access an object of base type.

9.0 Polymorphism > 9.4 Virtual Functions

## Virtual destructor functions

- All classes that have virtual functions should have a destructor in the base class.

January 05, 2016 Proprietary and Confidential - 11 -

 Capgemini  
CONSULTING TECHNOLOGY SERVICES

If instances are created at run time on the heap via the new operator, then their destructors are automatically called when you execute the delete statement to release the space occupied by the instance itself. Because a derived class instance always contains a base class instance, it is necessary to invoke the destructor of both classes in order to ensure that all of the space on the heap is released.

Demo :virtual \_dest.cpp

```
#include <iostream.h>
class base{
public:
    virtual void identify( ){
        cout << "Base class " << endl;
    }
    virtual ~base( ){
        cout << "Base destructor " << endl;
    }
};
class derived : public base{
public:
    virtual void identify( ){
        cout << "Derived class " << endl;
    }
    virtual ~derived( ){
        cout << "Derived destructor " << endl;
    }
};
void main( ){
    base* ptr_base = new derived;
    ptr_base → identify();
    delete ptr_base;
}
```

Output is :

Derived class  
Derived destructor  
Base destructor

9.0 Polymorphism

## Pure virtual functions

- A Pure Virtual Function is a virtual Function with no implementation
- Syntax

```
class Base
{
public:
    virtual void show() = 0;
};
```

January 05, 2016 | Proprietary and Confidential | +13 +

 Capgemini

### Demo :pure\_virtual\_func.cpp

```
class Shape {
public:
    Shape(){}
    virtual ~Shape(){}
    virtual long GetArea() = 0;
    virtual long GetPerim()= 0;
    virtual void Draw() = 0;
private:
};
void Shape::Draw() {
    std::cout << "Abstract drawing mechanism!\n";
}
class Circle : public Shape {
public:
    Circle(int radius):itsRadius(radius){}
    ~Circle(){}
    long GetArea() { return 3 * itsRadius * itsRadius; }
    long GetPerim() { return 9 * itsRadius; }
    void Draw();
private:
    int itsRadius;
    int itsCircumference;
};
```

Demo :pure\_virtual\_func.cpp(Contd..)

```
void Circle::Draw() {
    std::cout << "Circle drawing routine here!\n";
    Shape::Draw();
}
class Rectangle : public Shape {
public:
    Rectangle(int len, int width):
        itsLength(len), itsWidth(width){}
    virtual ~Rectangle(){}
    long GetArea() { return itsLength * itsWidth; }
    long GetPerim() { return 2*itsLength + 2*itsWidth; }
    virtual int GetLength() { return itsLength; }
    virtual int GetWidth() { return itsWidth; }
    void Draw();
private:
    int itsWidth;
    int itsLength;
}
void Rectangle::Draw() {
    for (int i = 0; i<itsLength; i++) {
        for (int j = 0; j<itsWidth; j++)
            std::cout << "x ";
        std::cout << "\n";
    }
    Shape::Draw();
}
class Square : public Rectangle {
public:
    Square(int len);
    Square(int len, int width);
    ~Square(){}
    long GetPerim() {return 4 * GetLength();}
};
Square::Square(int len):Rectangle(len,len)
{}
Square::Square(int len, int width):Rectangle(len,width){
    if (GetLength() != GetWidth())
        std::cout << "Error, not a square... a Rectangle???\n";
}
int main() {
    Shape * sp;
    sp = new Circle(5);
    sp->Draw();
    sp = new Rectangle(4,6);
    sp->Draw();
    sp = new Square (5);
    sp->Draw();
    return 0;
}
```

9.0 Polymorphism

## Abstract base classes

➤ **Abstract Base Class:**

- Class with pure virtual function.
- No instances may be created.
- Represents abstract concepts.
  - Inherited classes can implement those functionalities.
- Class derived from an Abstract class must provide an implementation for the pure virtual function.

January 05, 2016 Proprietary and Confidential - 15 -

 Capgemini  
CONSULTING TECHNOLOGY SERVICES

Abstract base classes are classes used to derive other classes. Hence, no base class objects can be defined. This class cannot be instantiated. (A class, which can be instantiated, is called as a concrete class).

Designers of abstract base classes allow derivations to occur via protected data members and virtual functions. However, they want to prevent the user from actually creating an instance of the base class.

To create an abstract base class, specify at least one pure virtual function. A pure virtual function is a virtual function with no body.

```
class Base {  
public:  
    virtual void show( ) = 0;  
};
```

The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The “= 0” syntax simply tells the compiler that the function is pure and has no body.

A destructor function, even though it can be virtual, can never be made a pure virtual function.

9.0 Polymorphism  
**Lab**

➤ **Lab 6**



January 05, 2016 | Proprietary and Confidential | + 10 +

 Capgemini

## Summary

- **Compile-time polymorphism:**
  - Implemented in C++.
  - Overloaded functions and operators are used.
- **Run-time polymorphism:**
  - Select appropriate member function while the program runs.
- **Virtual function:**
  - Dynamically bound on references and pointers, not on objects
- **VTable: Resolves appropriate function calls.**
- **Abstract class: Class that contains a pure virtual function.**



## Review Question

- **Question1:** Polymorphism makes the system less extensible.
  - Option 1: True
  - Option 2: False
- **Question 2:** Classes from which we would never want to create objects are called as:
  - Option 1: Virtual class
  - Option 2: Abstract class
  - Option 3: Base Class
- **Question3:** There is only one VTABLE per class.
  - Option 1: True
  - Option 2: False



C++

**Lesson 10: String Class**

January 15, 2016

Proprietary and Confidential

+ 1 +



## Lesson Objectives

➤ **To understand:**

- String class
- Constant member functions in String class
- Operators in String function
- Member function



January 15, 2016 | Proprietary and Confidential | - 3 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Lesson Objectives

This lesson covers about String class and function. The lesson contents are:

#### Lesson 10: String Class

- 10.1. String Class
- 10.2. String Class – const Member functions
- 10.3. Operators Defined for String
- 10.4. Member Functions

10.0 String class

## String class

- **String class is part of the C++ Standard Library.**
- **To use the string class, include the header file with the #include directive the header file:**
  - `#include <string>`
- **Constructors:**
  - `string()`
  - `string( other_string )`
  - `string( other_string, position, count )`
  - `string( count, character )`

January 15, 2016 | Proprietary and Confidential | +3 =

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

One of the biggest time-wasters in C is to use character arrays for string processing and to track the difference between static quoted strings and arrays created on the stack and the heap. Sometimes you are passing around a `char*` and sometimes you must copy the whole array.

Especially because string manipulation is so common, character arrays are a great source of misunderstandings and bugs. Despite this, to create string classes remains a common exercise for beginning C++ programmers for many years. Standard C++ library string classes solves the problem of character array manipulation once and for all and keep track of memory even during assignments and copy-constructions. You simply do not need to think about it.

No matter which programming idiom you choose, there are really only about three things you would want to do with a string:

Create or modify the sequence of characters stored in the string.

Detect the presence or absence of elements within the string.

Translate between various schemes for representing string characters.

String class is part of the C++ standard library. A string represents a sequence of characters.

10.0 String class

## String class—constant Member functions

- **const char \* data ()**
  - Returns the contents of the string.
- **unsigned int length ()**
  - Returns the length of the string.
- **unsigned int size ()**
  - Returns the length of the string (same as the length function).
- **bool empty ()**
  - Returns true if the string is empty and false otherwise.

January 15, 2016 | Proprietary and Confidential | + 8 +



#### Constant Member Functions:

These functions do not modify the string.

`const char * data ()` Returns a C-style null-terminated string of characters representing the contents of the string.

`unsigned int length ()` Returns the length of the string .

`unsigned int size ()` Returns the length of the string (i.e., same as the length function).

`bool empty ()` Returns true if the string is empty, false otherwise.

10.0 String class

## Operators defined for String

➤ **Assign =**

- E.g.

```
string s1;
string s2;.....
s1 = s2; // the contents of s2 is copied to s1
```

➤ **Append +=**

- E.g.

```
string s1( "abc" );
string s2( "def" );
...
s1 += s2; // s1 = "abcdef" now
```

January 15, 2016 | Proprietary and Confidential | +5+

 Capgemini  
CONSULTING TECHNOLOGY SERVICES

```
Assign =
string s1;
string s2;
...
s1 = s2; // the contents of s2 is copied to s1

Append +=
string s1( "abc" );
string s2( "def" );
...
s1 += s2; // s1 = "abcdef" now
```

10.0 String class

## Operators defined for String(Contd...)

- **Indexing [ ]**
  - E.g.    string s( "def" );  
              char c = s[2]; // c = 'f' now  
              s[0] = s[1]; // s = "eef" now
- **Concatenate +**
  - E.g.    string s1( "abc" );  
              string s2( "def" );  
              string s3;  
              s3 = s1 + s2; // s3 = "abcdef" now

January 15, 2016 | Proprietary and Confidential | + 5 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

Indexing []  
string s( "def" );  
char c = s[2]; // c = 'f' now  
s[0] = s[1]; // s = "eef" now

Concatenate +  
string s1( "abc" );  
string s2( "def" );  
string s3;  
...  
s3 = s1 + s2; // s3 = "abcdef" now

10.0 String class  
**Operators defined for String(Contd..)**

➤ **Equality ==**

— E.g.

```
string s1( "abc" );
string s2( "def" );
string s3( "abc" );
...
bool flag1 = ( s1 == s2 ); // flag1 = false now
bool flag2 = ( s1 == s3 ); // flag2 = true now
```

January 15, 2016 | Proprietary and Confidential | - 7 -

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

```
Equality ==
string s1( "abc" );
string s2( "def" );
string s3( "abc" );
...
bool flag1 = ( s1 == s2 ); // flag1 = false now
bool flag2 = ( s1 == s3 ); // flag2 = true now
```

## 10.0 String class Operators defined for String(Contd..)

### ➤ Inequality != the inverse of equality

### ➤ Comparison <, >, <=, >=

— performs case-insensitive comparison

— E.g.

```
string s1 = "abc";
string s2 = "ABC";
string s3 = "abcdef";
...
bool flag1 = ( s1 < s2 ); // flag1 = false now
bool flag2 = ( s2 < s3 ); // flag2 = true now
```

January 15, 2016 | Proprietary and Confidential | - 8 -



Inequality !=  
Inverse of equality

Comparison <, >, <=, >=  
Performs case-insensitive comparison  

```
string s1 = "abc";
string s2 = "ABC";
string s3 = "abcdef";
...
bool flag1 = ( s1 < s2 ); // flag1 = false now
bool flag2 = ( s2 < s3 ); // flag2 = true now
```

10.0 String class

## Member functions

- **void swap ( other\_string )**
  - swaps the contents of this string with the contents of other\_string.
  - E.g.

```
string s1( "abc" );
string s2( "def" );
s1.swap( s2 ); // s1 = "def", s2 = "abc" now
```
- **string & append ( other\_string )**
  - appends other\_string to this string, and returns a reference to the result string.

January 15, 2016 | Proprietary and Confidential | + 9 +



```
void swap ( other_string )
Swaps the contents of this string with the contents of other_string.
string s1( "abc" );
string s2( "def" );
s1.swap( s2 ); // s1 = "def", s2 = "abc" now
string & append ( other_string )
Appends other_string to this string, and returns a reference to the result string.
```

10.0 String class

## Member functions (Contd.)

- **string & insert ( position, other\_string )**
  - Inserts other\_string into this string at the given position.
- **string & erase ( position, count )**
  - Removes count characters from this string, starting with the character at the given position.
- **unsigned int find ( other\_string, position )**
  - Finds other\_string inside this string and returns its position.
- **string substr ( position, count )**
  - Returns the substring starting at position and of length count from this string.

January 15, 2016 | Proprietary and Confidential | + 10 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

`string & insert ( position, other_string )`

Inserts other\_string into this string at the given position, and returns a reference to the result string.

`string & erase ( position, count )`

Removes count characters from this string, starting with the character at the given position. If count is omitted (only one argument is given), the characters up to the end of the string are removed. If both position and count are omitted (no arguments are given), the string is cleared (it becomes the empty string). A reference to the result string is returned.

`unsigned int find ( other_string, position )`

Finds other\_string inside this string and returns its position. If position is given, the search starts there in this string, otherwise it starts at the beginning of this string.

`string substr ( position, count )`

Returns the substring starting at position and of length count from this string.

10.0 String class

# Lab

➤ Lab 7



January 15, 2016 | Proprietary and Confidential | - 11 -

 Capgemini  
CONSULTING. INNOVATION. DIFFERENCING.

## Summary

- **String class is a part of the standard C++ library.**
- **It has some constant functions such as:**
  - length(), size()
  - These do not modify the string.
- **Operators overloaded in String class:**
  - ==, !=, <, >, <=, >=
- **Some member functions are:**
  - swap(), append(), find(), substring()



## Review Question

➤ **String constant member functions are:**

- option 1: length
- option 2: data
- option 3: erase
- option 4: empty



➤ **Comparison Operators are overloaded in string class.**

- True/False

➤ **Manipulating the string with the use of string class, increases the occurrence of bug.**

- True/False

# C++

## Lesson 11: Exception Handling

January 18, 2016 | Proprietary and Confidential | 45 •



## Lesson Objectives

➤ In this lesson, you will learn:

- Exception Handling
- How do you handle an exception?
  - Throw an Exception
  - Catch an Exception
- Uncaught Exceptions
- Exception Specifications
  - Unexpected Exception



January 18, 2016 | Proprietary and Confidential | + 2 +

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS SERVICES

### Lesson Objectives

This lesson covers about exception handling. The lesson contents are:

#### Lesson 11: Exception Handling

- 11.1. Exception handling
- 11.2. How to handle an Exception
- 11.3. Uncaught Exceptions
- 11.4. Exception Specification

11.1: Introduction to Exception Handling

## Introduction

- **Exception refers to an unusual condition in a program.**
  - These can be errors that cause the program to fail or certain conditions that can lead to an error.
- **You can follow the following broad level steps while handling an Exception:**
  - Find the problem (Hit the exception).
  - Inform that an error has occurred (Throw the exception).
  - Receive the error information (Catch the exception).
  - Take corrective measures (Handle the exception).

January 18, 2016 | Proprietary and Confidential | + 3 +

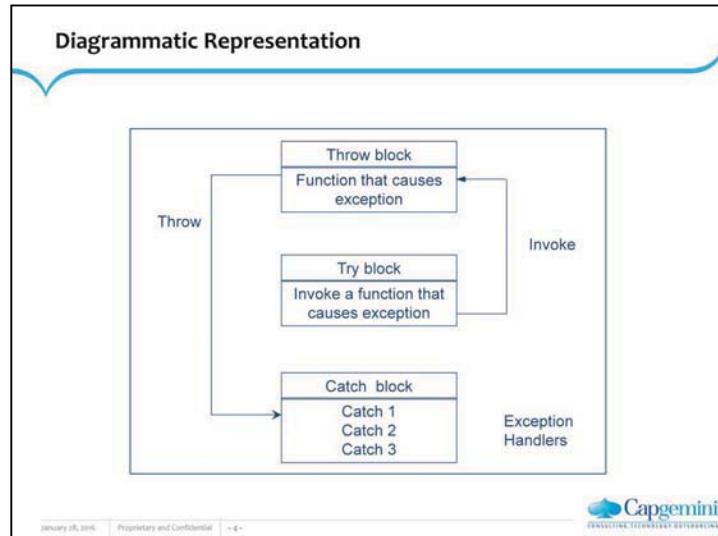
 Capgemini  
CONSULTING TECHNOLOGY BUSINESS SERVICES

### Introduction to Exception Handling:

Exception handling allows you to manage run-time errors in an orderly fashion. Using C++, exception handling your program can automatically invoke an error-handling routine when an error occurs.

Exception refers to unusual condition in a program. These can be errors that cause the program to fail or certain conditions that can lead to error.

Without exception handling there will be inherent problems in how a function can inform the caller that some error condition has occurred.



```
#include<iostream>
using namespace std;
double divide(long dividend, long divisor){
    return (double) dividend/divisor;
}
void main( ){
    cout << "Enter a dividend and a divisor : ";
    long dividend, divisor;
    while( !(cin >> dividend >> divisor).eof() ){
        cout << dividend << '/' << divisor << '='
            << divide( dividend, divisor ) << endl;
        cout << "Next dividend and divisor : ";
    }
}
```

### How to Handle an Exception?

The program in the above slide works just fine unless the function `divide()` gets called with a divisor equal to zero. Exception handling solves this problem.

11.1: How to Throw an Exception?  
**Explanation**

- While executing a program, if an error occurs, the exception handling provides a more elegant way to exit from the function or program.
- This exit happens regardless of the function's return type, and lets the caller retain control by having the function or program throw an exception.
- Syntax:

```
throw expression;
```

### Example

➤ Let us see an example on how to throw an exception:

```
double divide(long dividend, long divisor)
{
    if (divisor == 0L)
        throw divisor;
    return (double) dividend / divisor;
}
```

Below:

We can also throw a message instead as shown in the snippet given

```
double divide(long dividend, long divisor)
{
    if (divisor == 0L)
        throw "Division by zero";
    return (double) dividend / divisor;
}
```

11.1: How to Catch an Exception?

## Explanation

- The function causing exception should be invoked from the context of a try block.
- Immediately following the try block, you must write one or more handlers that can catch the exception being thrown.
- Even if there is only one statement that comprises the try block, the braces are still needed. Immediately following the try block, you must write one or more exception handlers that can catch the exception being thrown.

January 18, 2016 | Proprietary and Confidential | +2+

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS SERVICES

### How to Catch an Exception?

In order for the calling routine to recognize the various types of exceptions that are being thrown, it must first invoke the function within the context of a try block or any function called within a try block.

The exception handlers (also known as a catch block) can appear only after a try block or another exception handler. There can be no intervening statements.

The process of throwing and catching an exception entails searching the exception handlers after the try block for the first one that can be invoked. After the handler has finished its processing, all the remaining handlers are automatically skipped and execution of the program resumes normally with the first statement after the last handler.

## Syntax

- Following is the syntax for catching an exception:

```
try
{
    // call the function
}
catch( formal arguments )
{
    // process the exception
}
```

January 18, 2016 | Proprietary and Confidential | - 8 -



### Demo: catch\_exception.cpp

```
double divide (long dividend, long divisor)
{
    if ( divisor == 0L )
        throw "Division by zero ";
    return (double) dividend / divisor;
}
void main()
{
    cout << "Enter a dividend and a divisor : ";
    long dividend, divisor ;
    while( !(cin >> dividend >> divisor).eof() ){
        try{
            cout << dividend << ' / ' << divisor << '=' << divide( dividend, divisor);
        }
        catch( const char* message){
            cerr << message << endl ;
        }
        cout << "Next dividend and divisor : ";
    }
}
```

11.3: Uncaught Exceptions  
**Explanation**

- If there is no handler to catch the exception being thrown, then a special function called terminate() will be called and a runtime error message will be displayed. Then the program will automatically terminate.
- Rather, let an abort occur, you can provide a function that can serve as a default when an exception is thrown but not caught.

January 18, 2016 | Proprietary and Confidential | +9 +

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS SERVICES

#### Uncaught Exceptions:

What if there are no handlers to catch the exception being thrown?

In this case, a special function called terminate() will be called and a runtime error message will be displayed. Then the program will automatically be terminated.

The runtime error message is:

“No handler for thrown object”

Rather let an abort occur, you can provide a function that can serve as a default when an exception is thrown but not caught.

To do this, you must call the set\_terminate( ) function (prototyped in the file except.h) and provide an argument that is a pointer to your own terminate function. The latter function must take no arguments and return nothing.

### Example

➤ Let us see an example using `set_terminate()` function:

```
#include<iostream.h>
#include<except.h>
double divide(long dividend, long divisor) {
    if ( divisor == 0L )
        throw "Division by zero ";
    return (double) dividend / divisor;
}
void my_terminate( ){
    cerr << "Uncaught exception ";
}
void main( ){
    set_terminate(my_terminate);
```



**Example**

```
cout << "Enter a dividend and a divisor : ";
long dividend, divisor ;
while( !(cin >> dividend >> divisor).eof() ){
    try {
        cout << dividend << ' / ' << divisor << '=' << divide(dividend,
divisor);
    }
    catch( const char* message) {
        cerr << message << endl ;
    }
    cout << "Next dividend and divisor : ";
}
```

11.4: Exception Specification  
**Explanation**

➤ Exception specifications may be given along with the prototype of the function indicating the exceptions that can be thrown out of the function.

➤ Example:

```
double divide(long a, long b) throw(const char*);
```

— The function can throw exception of type const char\*.

January 18, 2016 | Proprietary and Confidential | +12 =

 Capgemini  
CONSULTING TECHNOLOGY BUSINESS SERVICES

#### Exception Specification:

Typically, the users of a function are only aware of the functions prototype, that is the number and types of the arguments that the function expects to receive, and the type of arguments (if any) that will be returned.

However, if the user is supposed to write exception handlers to accommodate all the exceptions that the function may throw, then obviously the types of the thrown expressions must also be known.

```
throw (type1, type2,...) ;  
double divide (long dividend, long divisor) throw (const char*);
```

## Unexpected Exception

- When a function with an exception specification throws an exception that is not listed in its exception specification, then the C++ run time does the following:
  - The unexpected() function is called.
  - The unexpected() function calls the function pointed to by unexpected\_handler. By default, unexpected\_handler points to the function terminate().
- The set\_unexpected() function replaces the default value of unexpected\_handler.

January 26, 2016

Proprietary and Confidential

&lt;13&gt;



### Exception Specification – Unexpected Exception:

- If the function throws an exception that has not been specified, then the function **unexpected()** will automatically get executed. This function will cause the program to terminate.
- The runtime error is:  
“violation of function exception specification”
- If you wish to trap this condition, you may call the **set\_unexpected()** function with an argument that is the address of your own function that takes no arguments and returns nothing. The function **set\_unexpected()** is prototyped in the file **except.h**.
- **Note:** Although **unexpected()** function cannot return, it may throw (or rethrow) an exception.

**Example**

```
#include<iostream.h>
#include<except.h>
double divide(long dividend, long divisor)
throw(const char*) {
    if ( divisor == 0L )
        throw divisor ;
    return (double) dividend / divisor ;
}
void my_unexpected( ){
    cerr << "Unexpected exception " << endl ;
}
```



**Example (contd.)**

```
void main( ){  
    set_unexpected(my_unexpected);  
    cout << "Enter a dividend and a divisor : ";  
    long dividend, divisor;  
    while( !(cin >> dividend >> divisor).eof() ){  
        try {  
            cout<<dividend<<'/'<<divisor,<<'='<<divide(dividend,  
divisor);  
        }  
        catch( const char* message){  
            cerr << message << endl;    }  
        cout << "Next dividend and divisor:";  
    }}}
```

January 18, 2016 | Proprietary and Confidential | +15+



Exception Specification:

The output is:

Enter a dividend and a divisor : 7 4

7 / 4 = 1.75

Next dividend and divisor : 7 0

Unexpected exception

## Demo

### ➤ Demo on:

- Handle an exception
- Throw and Catch an exception
- Uncaught Exception and use of unexpected()
- Exception Specification



**Lab**➤ **Lab 8: Exception Handling**

## Summary

➤ **In this lesson, you have learnt:**

- Exception handling allows you to manage run-time errors in an orderly fashion.
- Try catch blocks are used for exception handling.
- Terminate() handles the uncaught exceptions.
- Unexpected Exception handling is performed in case an exception, which is not specified in the Exception specification given at the time of function creation, occurs.



### Review Question

- **Question 1:** While handling exceptions in an application, we have to create a separate section of code to tackle the error called as:
  - Option 1: Try block
  - Option 2: Exception handler or catch block
  - Option 3: Option 1 and Option 2 both
- **Question 2:** We can have nested try blocks.
  - True/False



# C++

## Lesson 12: Templates

January 28, 2016 | Proprietary and Confidential | +1+



## Lesson Objectives

➤ In this lesson, you will learn:

- Templates
- Template Types
  - Function Templates
- Overloading Function Template
- Class Templates



January 28, 2016 | Proprietary and Confidential | + 2 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Lesson Objectives

This lesson covers Templates. The lesson contents are:

#### Lesson 12: Templates

- 12.1. Template
- 12.2. Template Types
- 12.3. Class Templates

12.1: Templates

## Concept of Templates

- Templates enable us to create skeleton for functions and for classes so that one can generate similar functions or classes.
- It is possible to create generic functions and generic classes.
- In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.

January 28, 2016 | Proprietary and Confidential | - 3 -



### Templates:

Templates enable us to create skeleton for functions and for classes so that one can generate similar function or classes.

Function templates provide you with the capability to write a single function that is a skeleton, or template for a family of similar functions.

In this function, at least one formal argument is generic.

This function template becomes a real function with real types when and only when it needs to be invoked, or its address taken.

With a template, it is possible to create generic functions and generic classes.  
Consider the following example:

```
#include<iostream.h>
int max(int x, int y)
{
    return (x > y) ? x : y ;
}
long max(long x, long y)
{
    return (x > y) ? x : y ;
}
```

## Templates (contd.):

```
double max(double x, double y)
{
    return (x > y) ? x : y ;
}
char max(char x, char y)
{
    return (x > y) ? x : y ;
}
void main( )
{
    cout << max(1, 2) << endl ;
    cout << max(4L, 3L) << endl ;
    cout << max(5.62, 3.48) << endl ;
    cout << max('A', 'a') << endl ;
}
```

Although function overloading has been used, the problem with the above example is that there is still too much repetitious coding. Each function is doing essentially the same thing – returning the greater of its two input arguments. Now, instead of you having to write many similar max( ) functions, it would be nice to be able to write just one max() function that returns the greater of its two input arguments and can accommodate almost any type of input argument.

A function template solves the problem by allowing you to write just one function. This function template does not specify the actual types of arguments that the function accepts. Instead, it uses a generic or parameterized type as a “place holder” until you determine the specific type. In generic function or class, the type of data upon which the function or class operates is specified as a parameter. This choice occurs implicitly when you make a call to the function. At this time, the compiler infers the type of actual arguments and then proceeds to instantiate the function with the actual type by replacing the generic type with the type you specify. For each new type that you code, a different function, called a template function, will be instantiated.

Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for different data types.

12.2: Template Types

## Classification of Templates

➤ **Templates can be categorized in two types:**

- Function Templates
- Class Templates

January 28, 2016 | Proprietary and Confidential | +5+



### Template Types:

There are two types of templates in C++, namely:

- function templates
- class templates

#### Function Templates:

Use function templates to perform identical operations for each type of data compactly and conveniently.,

You can write a single function template definition.

Based on the argument types provided in calls to the function, the compiler automatically instantiates separate object code functions to handle each type of call appropriately.

The STL algorithms are implemented as function templates.

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one variable type or class without repeating the code for each type.

#### Class Templates:

C++ class templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, then they become good candidates for being written as Templates.

One good area where this C++ Class Templates are suited can be container classes. Very famous examples for these container classes will be the STL classes like vector, list, and so on. Once code is written as a C++ class template, it can support all data types.

12.2: Function Templates

## Use of Function Templates

- A function template allows us to write just one function that serves as a skeleton, or template, for a family of functions whose all tasks are similar.
  - They do not specify the actual types of arguments that the function accepts
  - it uses a generic, or parameterized type as a “place holder” until a call to the function is made using the actual type.
  - For each new type, a different function, called a template function, will be instantiated.

January 28, 2016 | Proprietary and Confidential | + 8 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

### Function Templates:

There are lot of occasions, where we might need to write the same functions for different data types.

A favorite example can be addition of two variables. The variable can be integer, float, or double.

The requirement will be to return the corresponding return type based on the input type.

If we start writing one function for each of the data type, then we will end up with 4 to 5 different functions, which can be a nightmare for maintenance.

C++ templates come to our rescue in such situations. When we use C++ function templates, only one function signature needs to be created. The C++ compiler will automatically generate the required functions for handling the individual data types. This is how a programmer's life is made a lot easier.

## Use of Function Templates

➤ **Functions without C++ Template:**

- int Add(int a,int b) { return a+b;}
- float Add(float a, float b) { return a+b;}

➤ **Syntax:**

```
template <class identifier> function_declaration  
template <typename identifier> function_declaration
```

January 28, 2016 | Proprietary and Confidential | +2+



### Function Templates (contd.):

Let us assume a small example for Add function. If the requirement is to use this Add function for both integer and float, then two functions are to be created for each of the data type (overloading).

```
int Add(int a,int b) { return a+b;} // function Without C++ template  
float Add(float a, float b) { return a+b;} // function Without C++ template
```

If there are some more data types to be handled, more functions should be added.

However, if we use a C++ function template, then the whole process is reduced to a single C++ function template. The following will be the code fragment for Add function.

Syntax:

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;  
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword “class” or the keyword “typename”. Its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way.

```
template <class T>  
T Add(T a, T b) //C++ function template sample  
{  
    return a+b;  
}
```

## Example:Function Template

➤ Let us see an example using function template:

```
template <class T>
T Add(T a, T b)           //C++ function template sample
{
    return a+b;
}

template<class T>
const T& max(const T& x,const T& y)
{
    return (x>y)?x:y;
}
```

January 28, 2016 | Proprietary and Confidential | + 8 +



### Function Templates:

Function templates are implemented like regular functions, except they are prefixed with the keyword template.

Here T is the typename. This is dynamically determined by the compiler according to the parameter passed. The keyword “class” means, the parameter can be of any type. It can even be a class.

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max (int, int). Similarly the compiler generates definitions for max (char, char) and max (float, float) in this case.

### Example: Function Template

```
#include<iostream>
using namespace std;
template <class T>
const T& max(const T& x, const T& y) {
    return (x > y) ? x : y;
}
void main( ) {
    cout << max(1, 2) << endl ;
    cout << max(4L, 3L) << endl ;
    cout << max(5.62, 3.48) << endl ;
    cout << max('A', 'a') << endl ;
    cout << max(5, 6) << endl ;
}
```

January 28, 2016 | Proprietary and Confidential | + 9 +



#### Function Templates:

A function template should be written at the start of your program in the global area, or you may place it into a header file. All function templates start with a template declaration.

The output is as shown below:

```
2
4
5.62
a
6
```

Each of the first four max( ) function calls causes a template function to be instantiated, first with an int , then with a long, then with a double, and then with a char. The last call using two integers does not cause another instantiation to occur because this call uses the instantiated function generated by the first call that also used integers.

You cannot instantiate the function template with different types. The following call to the max( ) function will not compile because the first argument is an int and the second is a double.

```
cout << max(4, 2.6) << endl ;
```

## Demo

- Demo on:
  - Function Template



12.2: Overloading a Function Template

## Example

Let us see an example on overloading a Function template:

```
#include<iostream>
#include<string>
using namespace std;
const char* max(const char* x, const char* y)
{
    return strcmp(x, y) > 0 ? x : y;
}
template <class T>
const T& max(const T& x, const T& y)
{
    return (x > y) ? x : y;
}
void main()
{
    cout << max(1, 2) << endl;
    cout << max('A', 'B') << endl;
    cout << max("A", "B") << endl;
}
```

January 28, 2016 | Proprietary and Confidential | + 11 +



### Overloading a Function Template:

Sometimes a function template just cannot handle all the possible instantiations that you might want to do. The program either will not compile, or if it does the wrong answers may be obtained.

In this case, you must overload the function template by declaring the function, and specifying fixed argument types, just as you would for any other overloaded function. Pointers are a good example of where you need to overload a function template.

The output is as shown below:

2

B

B

## Demo

- Demo on:
  - Overloading Function Template



## 12.2: Class Templates Use of Class Templates

- A class template defines all algorithms used by that class.
- However, the actual type of data being manipulated will be specified as a parameter when objects of that class are created.
- Syntax:

```
template <class T>
class test
{
    // data & functions
}
```

January 28, 2016 | Proprietary and Confidential | + 13 +



### Class Templates:

A class template defines all algorithms used by that class. However, the actual type of data being manipulated will be specified as a parameter when objects of that class are created.

Syntax:

```
template <class T>
class test
{
    // data & functions
}
```

## Use of Class Templates

- We must tell the compiler that you are referring to a template class, not a non-template class by writing the “class name” followed by the “parameterized type name” between angle brackets.
- Syntax:

Test <T>



January 28, 2016 | Proprietary and Confidential | + 14 +

### Class Templates:

The name of the non-parameterized class and the parameterized class is not the same. In other words, whenever you need to write the class name, for example, in front of the scope resolution operator, you must tell the compiler that you are referring to a template class, not a non-template class. You do this by writing the “class name” followed by the “parameterized type name” between angle brackets. Assuming that the parameterized type name is T, then whenever you need to refer to the parameterized class name test within or outside the class definition of test, you must write:

Test <T>

## Example

➤ Let us see an example on Class Templates:

```
#include<iostream.h>
template <class T>
class test {
    T number;
public:
    test( ):number(0) // Default constructor
    {}
    test( T n ):number(n) // 1 argument constructor
    {}
    test(const test<T>& t ):number( t.number )
    {}      // Copy constructor
```

### Example (contd.)

```
-test( ) //destructor
{
    test<T>& operator =( const test<T>& t ) {
        number = t.number;
        return *this;
    }
    friend ostream& operator << (ostream& stream, const test<T>& t)
    {
        return stream << t.number;
    }
};
```

January 28, 2016 | Proprietary and Confidential | +16+



### Class Templates (contd.):

Suppose the member functions (and friend functions) are defined outside the class definition. Then all the class member functions are automatically template functions, and the template declaration must be repeated for each function.

```
#include<iostream.h>
template <class T>
class test
{
    T number;
public:
    test();
    test( T );
    test (const test<T>& );
    ~test();
    test<T>& operator =( const test<T>& );
    friend ostream& operator << (ostream& stream, const test<T>& );
};
```

Class Templates (contd.):

```
template <class T>
test <T> :: test( ) : number( 0 ) {}
template <class T>
test <T> :: test( T n ) : number( n ) {}
template <class T>
test <T> :: test( const test<T>& t ):number( t.number ) {}
template <class T>
test <T> :: ~test( ) {}
template <class T>
test <T>& test<T>::operator =(const test<T>& t )
{
    number = t.number ;
    return *this ;
}
template <class T>
ostream& operator<<(ostream& stream, const test<T>& t )
{
    return stream << t.number ;
}
```

### Example (contd.)

```
void main( )
{
    test<int> t1(10);
    cout << t1 << endl;
    test<long> t2 (123456789L);
    cout << t2 << endl;
}
```

## Demo

- Demo on:
  - Class Templates



Lab

➤ Lab 9



January 28, 2016 | Proprietary and Confidential | + 20 +

 Capgemini  
CONSULTING TECHNOLOGY OUTSOURCING

## Summary

➤ In this lesson, you have learnt:

- Templates are helpful for the creation of generic functions and classes that can work with different types of data.



## Review Question

- Question 1: Class templates are normally used for \_\_\_\_\_ classes.
- Question 2: We can override a function template to behave in one way for all the data types except one.
  - True/False
- Question 3: Does using templates save memory ?
  - Yes / No



## C++

### Lesson 13: I/O in C++

January 26, 2016 | Proprietary and Confidential | - 1 -



## Lesson Objectives

➤ In this lesson, you will learn:

- Input/output in C++
- File Handling:
  - File Opening Modes
  - Opening and Closing a File
  - Writing to a file
  - Reading From a file
- Some useful functions



January 26, 2016 | Proprietary and Confidential | -2 -



### Lesson Objectives

This lesson covers about file handling in C++. The lesson contents are:

Lesson 13: Input Output in C++

- 13.1. Input Output in C++
- 13.2. File Handling
- 13.3. Some Useful Functions

13.1: Input / Output in C++

## Introduction

- C++ provides a device-independent way to work with I/O devices known as Streams (sequence of bytes).
- A program extracts the bytes from an input stream and inserts byte into an output stream.

January 08, 2016 | Proprietary and Confidential | - 3 -

 Capgemini  
INNOVATION. INSPIRATION. PERFORMANCE.

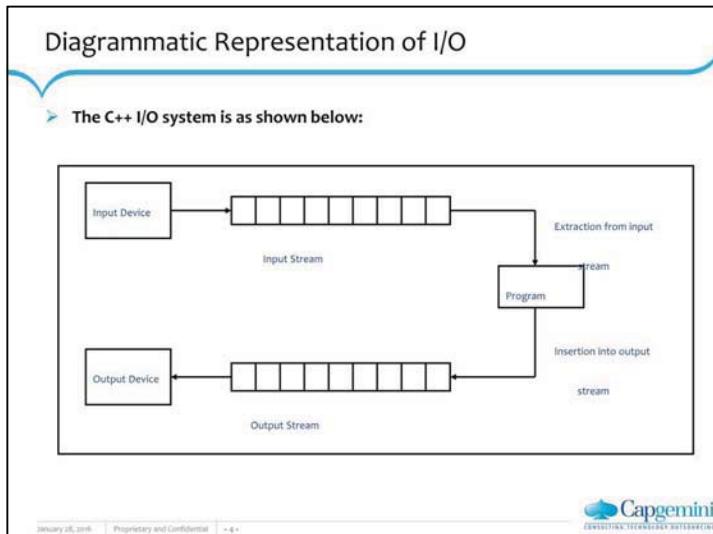
### Input / Output in C++:

The I/O system in C++ provides a device independent interface to work with a variety of devices. This interface, known as a stream, is a sequence of bytes. A program extracts the bytes from an input stream and inserts byte into an output stream.

Streams: I/O uses the concept of streams - a flow of characters. These characters may be 1 byte or 2 byte ('wide'), the latter necessary for languages with many characters. Streams may flow into or out of files or strings. C++ tries to offer the same set of commands whatever the nature of the source and destination.

The following streams are predefined:

cin: the standard source of input (often the keyboard)  
cout: the standard destination for output (often the terminal window)  
cerr: the standard destination for error messages (often the terminal window). Output through this stream is unit-buffered, which means that characters are flushed after each block of characters is sent.  
clog: like cerr, but its output is buffered.



#### Input / Output in C++ (contd.):

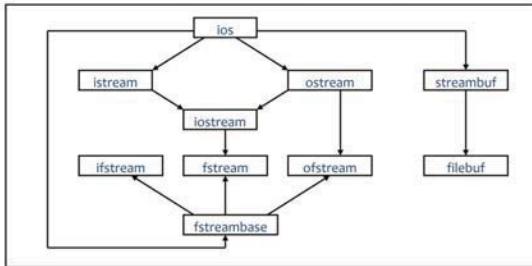
**buffers:** It is inefficient to update a file on disc each time a character is added. The character is written into a buffer, and when enough characters are in the buffer to make updating worthwhile, the buffer is flushed and the file on disc is updated. C++ offers ways to control the behavior of buffers, but often the default is ok.

**state:** Each stream has state information indicating whether an error has occurred, and so on.

Note: The C++ I/O system contains a hierarchy of stream classes used for input and output operations. These classes are included in the header file 'iostream.h'.

### Diagrammatic Representation of I/O

- The I/o subsystem in C++ is implemented using a hierarchy of classes as shown below:



January 08, 2016 | Proprietary and Confidential | - 5 -



#### Input / Output in C++ (contd.):

The ios class is a virtual base class that contains basic facilities that are used by all other i/o classes. It also contains a pointer to a streambuf object.

The istream class declares input functions such as get(), getline(), read(), and so on. It contains overloaded extraction operator >> for all the basic data types.

The ostream class declares output functions put(), write(). It contains overloaded insertion operator for all the basic data types.

cin is an object of istream class which represents standard input stream. cout is an object of ostream class, which represents the standard output stream.

The streambuf provides an interface to physical devices through buffers. It acts as a base for filebuf class used by ios files.

13.2: File Handling

## Introduction

- For file handling, the I/O system of C++ uses the following classes :
  - ifstream
  - ofstream
  - Fstream
- To use them, include the header file fstream.h.
- The class ifstream handles file input (reading from files).
- The class ofstream handles file output (writing to files).

January 26, 2016 | Proprietary and Confidential | + 6 \*

 Capgemini  
INNOVATION. INSPIRATION. PERFORMANCE.

### File Handling:

The constructor for both classes (ifstream and ofstream) will actually open the file if you pass the name as an argument. Also, both classes have an open command (`a_file.open()`) and a close command (`a_file.close()`). It is generally a good idea to clean up after yourself and close files once you are finished. Filebuf is used to set the file buffers to read and write.

## File Opening Modes

Let us see some of the file opening modes in C++:

Mode	Description
ios::app	It appends output to the end of the file. The pointer cannot be repositioned.
ios::ate	It opens the file and moves to the end. Data may be written anywhere in the file by repositioning the pointer.
ios::in	It opens the file for input.
ios::out	It opens the file for output. If the file exists, it will be overwritten.
ios::trunc	It truncates the files contents if it exists.
ios::nocreate	It only opens the file if it exists.

### File Handling:

#### File Opening Modes:

Before processing a file it needs to be opened in one or more modes listed below.

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 at the current position.

ios::in : If FileName is a new file, then the operation fails. 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 as an empty file. Once it gets created as empty file, you can write data to it. If FileName already exists, then it is opened, its content is destroyed, and the file becomes like a new file. Therefore you can create new data to write to it.

ios::trunc : If FileName already exists, its content is destroyed and the file becomes as new.ios::nocreatelf 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 will fail because it cannot create a file of the same name in the same location.

## Opening and Closing a File

➤ **File can be opened in two ways:**

- Using Constructor
- Using open()

➤ **Example 1: Opening a file using Constructor:**

```
ofstream outfile("samplefile"); //default mode is ios::out  
ofstream a_file("test.txt", ios::nocreate);  
ofstream outFile("junk.txt", ios::out);  
char FileName[20] = "Test.txt";  
ifstream Students(FileName);
```

### File Handling:

#### Opening and Closing a File:

To open a file we must create a file stream, and then link it to the filename.

The file stream can be defined using the classes ifstream, ofstream, and fstream depending upon the purpose (that is whether we want to read data from file or write data to it or both).

A file can be opened in two ways:

- By using the constructor function of the class
- By using the member function open() of the class

## Opening and Closing a File

### ➤ Example 2: Opening a file using open

```
ifstream infile;  
infile.open("samplefile");  
ofstream file;  
file.open ("example.bin", ios::out | ios::app | ios::binary);
```

### ➤ Example 3: Closing a file

```
Infile.close();  
Outfile.close();
```

## Writing to a File

### ➤ Example 1: Writing to a File

```
#include <iostream.h>
#include <fstream.h>
int main() {
    char name[80];
    int age;
    ofstream outFile("junk.txt", ios::out);
    // Check if there was an error opening file
    if (!outFile) {
        cout << "Unable to open the file\n";
        return 1;
    }
```

## Writing to a File

### ➤ Example 1: Writing to a File (contd.)

```
cout << "Enter your name: ";
cin.getline( name);

cout <<"Enter your age :";
cin >> age;

outFile << name << "\t" << age << endl;
outFile.close(); // close file & release stream
return 0;
}
```

January 08, 2016 | Proprietary and Confidential | + 11 +



## Reading from a File

### ➤ Example 1: Reading from a file

```
#include <fstream>
#include <iostream>
int main() {
    char FirstName[30], LastName[30];
    int Age;
    char FileName[20];
    cout << "Enter the filename you want to open: ";
    cin >> fileName;

    ifstream Students(FileName);
```

January 26, 2016 | Proprietary and Confidential | + 12 +



## Reading from a File

### ➤ Example 1: Reading from a file (contd.)

```
// Check if there was an error opening the file
if (!Students)      {
    cout << "Unable to open the file\n";
    return 1;
}
Students >> FirstName >> LastName >> Age;
cout << "\nFirst Name: " << FirstName;
cout << "\nLast Name: " << LastName;
cout << "\nAge:      " << Age;
return 0;
}
```

January 26, 2016 | Proprietary and Confidential | +13 +



## Demo

➤ **Demo on:**

- Writing to a file
- Reading from a file



13.3: Some More Useful Functions

## List of Functions

➤ Let us see some more useful functions in C++ :

- seekg() : It puts the pointer at required location while reading from a file.
- seekp() : It puts the pointer at required location while writing to a file.
- tellg() : It tells the current pointer position while reading from a file.
- tellp() : It tells the current pointer position while writing to a file.

January 08, 2016 | Proprietary and Confidential | + 95 +

 Capgemini  
INNOVATION TECHNOLOGY INSPIRATION

### Some More Useful Functions:

Let us see some more useful functions in C++ :

seekg() : It puts the pointer at required location while reading from a file.

For example:

```
File.seekg(-5,ios::end);
```

It will put the pointer 5 characters before end

```
File.seekg(ios::beg);
```

It will put the pointer to the beginning of the file.

```
File.seekg(10,ios::cur)
```

It will put the pointer 10 characters ahead from current position

seekp() : It is similar to seekg(), but it is used when you write in a file.

tellg() : It returns an int type that shows the current position of the inside-pointer. This works only when we read a file.

tellp() : It is the same as tellg() but is used when we write in a file

flush () : It is used to save the data from the output buffer, even if the buffer is still not full.

## List of Functions

➤ Let us see some more useful functions in C++ (contd.):

- flush(): It is used to save the data from the output buffer, even if the buffer is still not full.

## Summary

➤ In this lesson, you have learnt:

- Streams are used to interact with devices in C++.
- C++ uses ifstream, ofstream, and fstream classes for file handling.
- Reading and writing to file requires opening the file using open() or constructor function in the appropriate mode. At the end the files need to be closed using close().



January 06, 2016 | Proprietary and Confidential | + 17 +

Add the notes here.

## Review Question

➤ **Question 1: The output sent to which of the following stream objects is not buffered:**

- Option 1: clog
- Option 2: cout
- Option 3: cerr



➤ **Question 2: Which of the following is the default file opening mode?**

- Option 1: ios::in
- Option 2: ios::out
- Option 3: ios::app

# C++

## Lesson 14: Best Practices

January 18, 2018 | Proprietary and Confidential | 14 / 14



## Lesson Objectives

➤ **In this lesson, you will learn about:**

- The best practices to be followed with respect to the following:
  - Variable Declaration
  - Class Definition
  - General



January 08, 2008 | Proprietary and Confidential | - 2 -

 Capgemini  
CONSULTING TECHNOLOGY INTEGRATION

### Lesson Objectives

This lesson covers about the practices to be followed. The lesson contents are:

#### Lesson 14: Good Programming Practices

- 14.1. Variable Declaration
- 14.2. Class Definition
- 14.3. General

14.1: Best Practices

## Introduction

➤ **Goal of a Programmer:**

- To develop consistent source code and robust product

➤ **To meet this goal, the code should fulfill the following criteria:**

- Correct
- Maintainable
- Fast / Small
- Reusable
- Portable

January 08, 2008 | Proprietary and Confidential | 34

 Capgemini  
CONSULTING TECHNOLOGY INTEGRATION

### Best Practices:

A set of guidelines to develop a consistent source code and robust products are available. For using these guidelines, the code should be correct, maintainable, fast/small, reusable, and portable.

Correct code is the one which will solve the problem and is bug free.

Maintainable code is understandable code if it needs to be enhanced or extended.

Make your code fast and small as much as possible.

Reusable code is desirable if it can be used in some other places in the same or different projects.

Ensure that the code is portable.

Note: Some projects have their own coding conventions and guidelines. Such guidelines will over rule all the other guidelines. You have to follow only the guidelines given by the customer.

## 14.2: Variable Declaration Explanation

➤ **Ensure the following while declaring a variable:**

- Declare and initialize variable closer to where they are used.
- Initialization is better than assignment.
- Variable should have most restricted scope.

14.3: Class Definition  
**Explanation**

➤ Ensure the following while defining a class:

- Class should have public, protected, and private members in that order.
- Public function must never return pointer to a local variable.
- All the resources allocated in the constructor must be deallocated in the destructor.
- All member functions that do not modify class data should be declared as constant member functions.
- Constructor and destructors should not be inline.

January 08, 2006 | Proprietary and Confidential | 14.3 - 1

 Capgemini  
CENTRAL TECHNOLOGY SERVICES

**Class Definition:**

Returning a pointer or reference to a local variable is always wrong since it gives the user a pointer or reference to an object that no longer exists. Such pointer or reference cannot be used without the risk of overwriting the caller's stack space. Most compilers warn about this, but mistakes are still possible.

Example: Returning dangling pointers and references

```
int& dangerous() {  
    int i = 5; return i; // NO: Reference to local returned  
}  
  
int& j = dangerous(); // NO: j is dangerous to use  
// much later: cout << j; // Crash, program dies
```

14.4: General  
**Explanation**

➤ In general, ensure the following while developing your code:

- Code should be well-structured, consistent in style, and consistently formatted.
- All variables should have proper type consistency.
- All variables and pointers should be initialized before its use.
- Loop terminations conditions should be achievable and obvious.
- Array declaration should not have hard coded constants.

January 08, 2008 | Proprietary and Confidential | + 6 +

 Capgemini  
CENTRAL & EAST ASIA

General Best Practices:

Example 1: Consider the following array declaration:

```
int iarray[13];
```

Here, the code should not have any uncalled procedure or unreachable/dead code.  
Instead of the above method, declare the array as shown below:

```
int iarray[TOT_MONTHS+1];
```

Example 2:

```
char szIniFileName[_MAX_PATH];
```

## Summary

➤ In this lesson, you have learnt:

- Better methods of declaring variables
- Points to be taken care of while defining a class



## Review Question

- **Question 1:** What is the order in which class members are defined?
  - Option 1: private
  - Option 2: public
  - Option 3: protected
- **Question 2:** \_\_\_\_\_ is understandable code if it needs to be enhanced or extended.



## C++

**Lesson 15: Implementations of common Design Patterns in C++with relevant examples**



## Lesson Objectives

➤ In this lesson, you will learn implementations and some examples of following design patterns:

- Fundamental Patterns: Delegation Pattern,
- Creational Patterns: Abstract Factory Method, Singleton
- Structural Patterns: Façade
- Behavioral Patterns: State, Strategy



January 18, 2016 | Proprietary and Confidential | - 2 -

**Capgemini**  
CONSULTING IT SERVICES

### Lesson Objectives:

This lesson provides some examples for design patterns.

## Concept of Design Pattern

- Design Pattern is a solution to a problem in a context.
- Pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”
- Design Patterns are “reusable solutions to recurring problems that we encounter during software development.”
- Patterns enable programmers to “... recognize a problem and immediately determine the solution without having to stop and analyze the problem first.”
- They provide reusable solutions.
- They enhance productivity.



January 18, 2016 | Proprietary and Confidential | - 3 -

### What is a Design Pattern?

“A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever using it the same way twice.”

Patterns can be applied to many areas of human endeavor, including software development.

## Design Patterns are NOT

- They are not data structures that can be included in classes and reused as is (i.e. linked lists, hash tables, etc)
- They are not complex domain-specific design solutions for the entire application
- Instead, they are:
  - Proven solutions to a general design problem in a particular context which can be customized to suit our needs.



January 18, 2016 | Proprietary and Confidential | + 4 +

### Features of JSP:

Do not ever think that the class structure given by Design Patterns can be used as is unlike Data structures (namely, link lists, hash tables, etc).

They do not provide a domain specific design solution for an entire application.

Instead they provide a proven solution to a general design problem in a particular context (not domain) which can be customized to suit our needs.

## Broad level Categories of Design Patterns

➤ **Design Patterns can be broadly classified as:**

- Fundamental patterns
- Creational patterns
- Structural patterns
- Behavioral Patterns



January 18, 2016 | Proprietary and Confidential | - 5 -

### Classification of GOF Design Patterns:

The Design Patterns can be broadly classified as :

Fundamental Patterns: They are the building blocks for the other three categories of Design Patterns.

Creational Patterns: They deal with creation, initializing, and configuring classes and objects.

Structural Patterns: They facilitate decoupling interface and implementation of classes and objects.

Behavioral Patterns: They take care of dynamic interactions among societies of classes and objects. They also give guidelines on how to distribute responsibilities amongst the classes.

## What is a Fundamental Design Pattern

- These are widely used by other patterns or are frequently used in a large number of programs.
- Fundamental patterns are further classified as:
  - Delegation Pattern
  - Interface Pattern
  - Abstract Superclass
  - Interface and abstract class
  - Immutable Pattern
  - Marker Interface Pattern



January 18, 2016 | Proprietary and Confidential | + 6 +

### Note:

The fundamental design patterns are the building blocks of the other design patterns.

#### Fundamental Design Patterns:

**Delegation Pattern:** It is a way of extending and reusing a class using composition rather than inheritance.

**Interface Pattern:** This pattern facilitates the design principle – “Program to Interface rather than Implementation”.

**Abstract Superclass:** It ensures consistent behavior of conceptually related classes by giving them a common abstract superclass.

**Interface and abstract class:** It is a combination of Interface and Abstract superclass Patterns.

**Immutable Pattern:** This pattern prevents the object from changing its state information after it is constructed thereby eliminating the issues related to concurrent access of the object.

**Marker Interface:** It is used to mark an object to be part of a particular group thereby allowing other objects to treat them in a uniform way.

We will see Delegation, Interface, and Abstract Superclass in some more detail.

4.1: Examples of Fundamental Patterns

## Delegation Pattern

- A class outwardly expresses certain behavior, but in reality delegates responsibility for implementing that behavior to another class
- Used for extending a class behavior
  - As against inheritance where operations are inherited, delegation involves a class calling another class's operation
  - Suitable for "Is a role played by" relationship
- Helps in reducing coupling in the System

January 18, 2016 | Proprietary and Confidential | - 2 -

 Capgemini  
CONSULTING IT SERVICES

### Delegation Pattern:

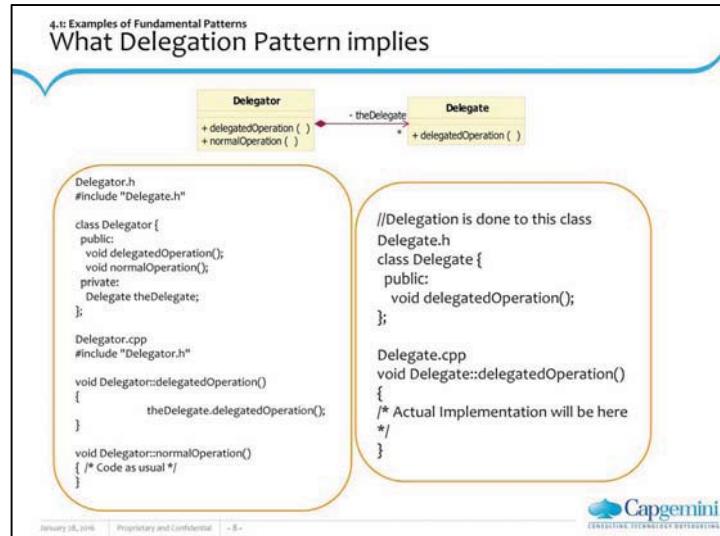
The delegation pattern is a technique where an object outwardly expresses certain behavior but in reality delegates responsibility for implementing that behavior to an associated object in an Inversion of Responsibility.

Inheritance is a common way to extend and reuse the functionality of a class.

Delegation is a more general way for extending a class's behavior that involves a class calling another class's methods rather than inheriting them.

Inheritance is useful for capturing "is-a-kind-of" relationships because of their static nature. Delegation is suitable for "is-a-role-played by" relationship.

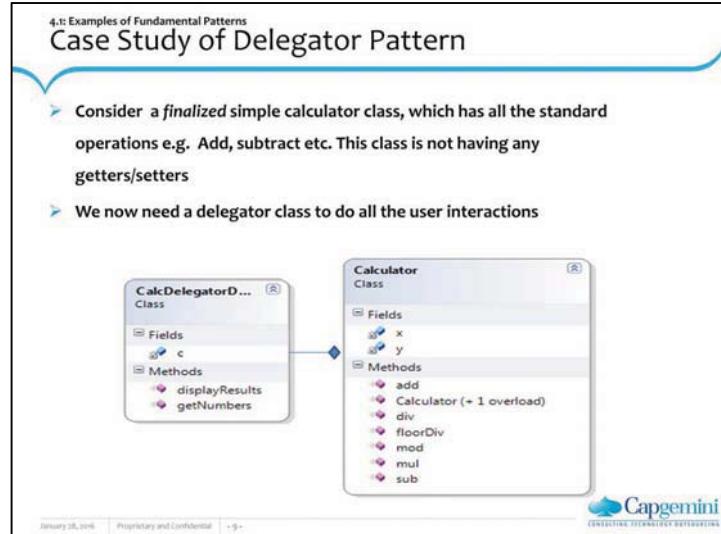
Suppose it is found that a class attempts to hide a method or variable inherited from a super class or from other classes. Then that class should not inherit from the super class. There is no effective way to hide methods or variables inherited from a super class. However, it is possible for an object to use another object's methods and variables while ensuring that it is the only object with access to the other object. This accomplishes the same thing as inheritance but uses dynamic relationships that can change over time.



### Structure of Delegation Pattern:

You can use delegation to reuse and extend the behavior of a class. You do this by writing a new class (the delegator) that incorporates the functionality of the original class by using an instance of the original class (the delegatee) and calling its methods.

Delegation is more general purpose than inheritance. Any extension to a class that can be accomplished by “inheritance” can also be accomplished by “delegation”. The implementation of delegation is very straightforward, for it simply involves the acquisition of a reference to an instance of the class to which you want to delegate and call its methods. The best way to ensure that a delegation is easy to maintain is to make its structure and purpose explicit.



```

#include <iostream>
using namespace std;
class Calculator {
public:
    int add() { return x+y; }
    Calculator(unsigned a,unsigned b) { x=a;y=b; }
    unsigned add() const { return x+y; }
    unsigned sub() const { return x-y; }
    unsigned mul() const { return x*y; }
    int mod() const {
        if(y==0) return x%y;
        else return -1;
    }
    float floorDiv() const {
        if(y==0) return x/y;
        else return -1;
    }
    float div() const {
        if(y==0) return float(x)/float(y);
        else return -1;
    }
private: unsigned x,y;
}
//We need a delegator since the calculator does not have getter/setter methods to interact with outside world
class CalcDelegatorDemo {
public:
    void getNumbers() {
        int val1,val2;
        cout<<"Enter two values:"<<endl;
        cin>>val1>>val2;
        c=new Calculator(val1,val2);
        return ;
    }
    void displayResults() const {
        cout<<"The result of all operations:"<<endl;
        cout<<>>add()<<>>sub()<<>>mul()<<>>mod()<<>>div()<<>>floorDiv()<<endl;
    }
private: Calculator *c;
};

int main() {
    CalcDelegatorDemo d;
    d.getNumbers();
    d.displayResults();
    d.getNumbers();
    d.displayResults();
    d.getNumbers();
}

```

4.2: Examples of Creational Patterns

## What is a Creational Design Pattern

- Creational Design Patterns are design patterns that deal with object creation mechanisms.
- They help to create objects in a manner suitable to the situation.
- They provide guidance on how to create objects when their creation requires making decisions.
- Creational Design Patterns are further classified as:
  - Factory method
  - Abstract Factory
  - Prototype
  - Builder
  - Singleton

January 18, 2016 | Proprietary and Confidential | + 10 +

 Capgemini  
CONSULTING IT SERVICES

### Introduction to Creational Patterns:

Creational Patterns provide guidelines on creation, configuration, and initialization for objects.

“Decisions typically involve dynamically deciding which class to instantiate or which objects an object will delegate responsibility to.”

3.2: Examples of Creational Patterns

## Factory Method Pattern

- Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to subclasses.
- Use of Factory pattern promotes loose coupling.

January 18, 2016 | Proprietary and Confidential | - 11 -

 Capgemini  
CONSULTING IT SERVICES

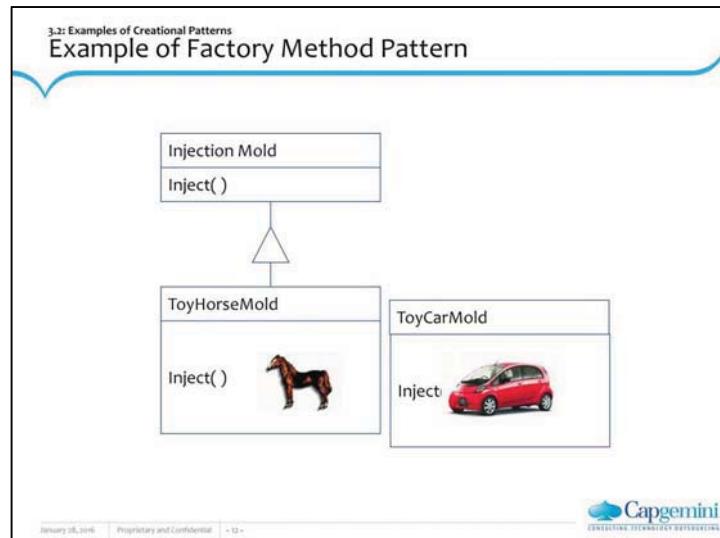
### Factory Method:

You can define an interface for creating an object, however, let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

It is also called as a Factory Pattern since it is responsible for “manufacturing” an Object.

The Factory Method pattern is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which can then be overridden by subclasses to specify the derived type of the product that will be created.

They promote loose coupling by eliminating the need to bind application specific classes into the code.



#### Example of Factory Method Pattern:

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate.

Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

## 4.2: Examples of Creational Patterns

## Cons of Factory Method

➤ **Cons:**

- Two major varieties: The creator superclass may or may not implement the factory method. In any case, the superclass needs to be subclassed to create a concrete product.
- Parameterized factory methods: Multiple kinds of products can be created by passing parameters to factory methods. There is no standardization on this aspect.

4.3: Abstract Factory Pattern

## Usage of Abstract Factory Pattern

- The Abstract Factory pattern is one higher level of abstraction than the factory method pattern.
- The Abstract factory is a factory object that returns one of several factories.

January 18, 2016 | Proprietary and Confidential | + 14 +



### Abstract Factory Pattern:

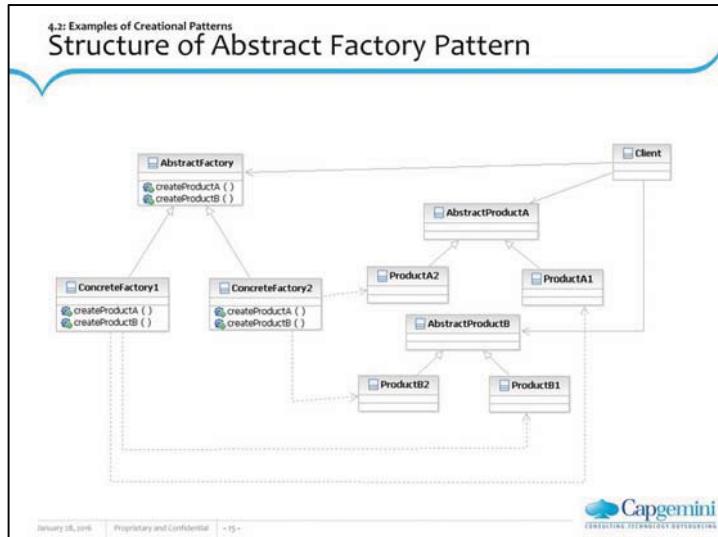
Being a software design pattern, the Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme.

It provides an interface for creating families of related or dependent objects without specifying their concrete classes.

In normal usage, the client software creates a concrete implementation of the abstract factory, and then uses the generic interfaces to create the concrete objects that are part of the theme.

The client does not know (nor care) about the concrete objects that it gets from each of these internal factories. This is because it uses only the generic interfaces of their products.

This pattern separates the details of implementation of a set of objects from its general usage.



### Structure of Abstract Factory Pattern:

#### Structure Elements:

**AbstractFactory:** It declares an interface for operations that create abstract product objects.

**ConcreteFactory:** It implements the operations to create concrete product objects.

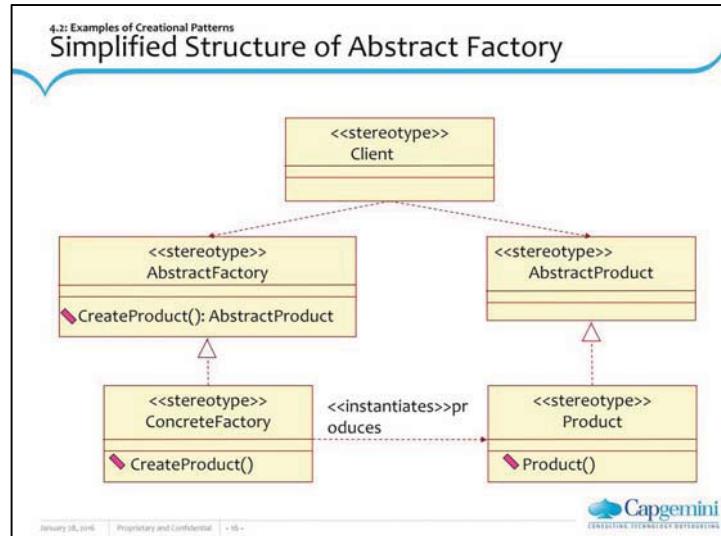
**AbstractProduct:** It declares an interface for a type of product object.

#### ConcreteProduct:

It defines a product object to be created by the corresponding concrete factory.

It implements the **AbstractProduct** interface.

**Client:** It uses only interfaces declared by **AbstractFactory** and **AbstractProduct** classes.



**4.2: Examples of Creational Patterns**

## Case Study of Abstract Factory Pattern

- Consider the iPhone phone, which comes in different models iPhone2, iPhone3 etc with each version having upgraded hardware and operating system features
- Build a factory that produces different models of iPhone depending on the demand in market.

```

classDiagram
    class iPhoneFactory {
        Methods: GetPhoneType()
    }
    class iPhone {
        Methods: CameraType(), OsType(), PrintDetails()
    }
    class iPhone4 {
        Methods: CameraType(), OsType()
    }
    class iPhone5 {
        Methods: CameraType(), OsType()
    }

    iPhoneFactory --> iPhone
    iPhone4 --> iPhone
    iPhone5 --> iPhone
  
```

January 18, 2016 | Proprietary and Confidential - 17 -

**Capgemini**  
CONSULTING IT SERVICES

```

#include<iostream>
#include<string>
#include<memory>
#include<ios>
//Abstract Base class
class iPhone {
public:
    virtual string CameraType()=0;
    virtual string OsType()=0;
    void PrintDetails() {
        cout<<"Camera Type is "<<CameraType()<<endl;
        cout<<"Os Type is "<<OsType()<<endl;
    }
};

//Concrete Class
class iPhone4:public iPhone {
public:
    string CameraType() {
        return "5 Mega Pixel";
    }
    string OsType() {
        return "ios 4";
    }
};

class iPhone5:public iPhone {
public:
    string CameraType() {
        return "8 Mega Pixel";
    }
    string OsType() {
        return "ios 6";
    }
};

//Abstract Factory returning type of mobile.
class iPhoneFactory {
public:
    iPhone *GetiPhoneType(string type);
};

iPhone *iPhoneFactory::GetiPhoneType(string type) {
if(type == "iPhone 4") {
    return new iPhone4();
} if(type == "iPhone 5") {
    return new iPhone5();
} return NULL;
}

void main() {
iPhoneFactory *myFactory = new iPhoneFactory();
iPhone *myiPhone1 = myFactory->GetiPhoneType("iPhone 4");
myiPhone1->PrintDetails();

iPhone *myiPhone2 = myFactory->GetiPhoneType("iPhone 5");
myiPhone2->PrintDetails();
int i;
for(i=0;i<5;i++)
}
  
```

4.2: Examples of Creational Patterns

## Pros and Cons of Abstract Factory

**➤ Pros**

- It isolates concrete classes. Hence clients manipulate instances through their abstract interfaces.
- It makes exchanging product families easy by choosing the right factory, which in turn creates the products of the respective family.
- It promotes consistency among products by grouping family of products under a particular factory.

**➤ Cons**

- Supporting new kinds of products requires changing the AbstractFactory class and all of its subclasses.

January 18, 2016 | Proprietary and Confidential | - 18 -

 Capgemini  
CONSULTING IT SERVICES

### Pros and Cons of Abstract Factory:

#### Pros:

It isolates concrete classes: The Abstract Factory pattern helps you to control the classes of objects that an application creates. Since a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory. They are not displayed in client code.

It makes exchanging product families easy: The class of a concrete factory appears only once in an application — that is, where it is instantiated. This makes it easy to change the concrete factory that an application uses. It can use different product configurations simply by changing the concrete factory. Since an abstract factory creates a complete family of products, the whole product family changes at once. In our user interface example, we can switch from Motif widgets to Presentation Manager widgets simply by switching the corresponding factory objects and recreating the interface.

It promotes consistency among products: When product objects in a family are designed to work together, it is important that an application uses objects from only one family at a time. AbstractFactory makes this easy to enforce.

4.2: Examples of Creational Patterns

## Singleton Pattern

- The Singleton pattern ensures that only one instance of a class is created.
- All objects that use an instance of that class use the same instance.

January 18, 2016 Proprietary and Confidential • 19 •

 Capgemini  
CONSULTING IT SERVICES

### Singleton Pattern:

There are cases in programming where you need to ensure that there can be one and only one instance of a class which is used by the application.

The Singleton pattern ensures that a class has only one instance, and provides a global point of access to it.

It is important for some classes to have exactly one instance.

Although there can be many printers in a system, there should be only one printer spooler.

There should be only one File system and one Window manager.

A digital filter will have one A/D converter.

An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible?

A global variable makes an object accessible. However, it does not keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

**4.2: Examples of Creational Patterns**

## Case study of Singleton Pattern

- Consider a class which creates Log Book for recording all transactions in a bank application.
- Only one instance of this class has to be created.

```

LogFile
- uniqueInstance : LogFile* = 0
- LogFile( )
+ GetUniqueInstance( )

```



January 18, 2016 | Proprietary and Confidential | +10 +

Note the following  
A private constructor is used to restrict the application from creating multiple instances of the class  
A static instance variable makes the single instance globally available.  
A getUniqueInstance method is used to ensure that an instance is created the first time round and the same instance is used each time

```

LogFile.h
class LogFile {
public:
    /* STATIC operation implements the logic for returning the unique instance of the Singleton. It creates
    the unique instance if it does not already exist. */
    static LogFile* GetUniqueInstance();

private:
    /* Notice that the default constructor of the Singleton class is private, so that it CANNOT be accessed
    outside the Singleton class. */
    LogFile();

    /* This attribute stores the Singleton's unique instance. */
    static LogFile* uniqueInstance;
};

LogFile.cpp
#include "LogFile.h"

LogFile* LogFile::GetUniqueInstance() {
    if (uniqueInstance == 0)
    {
        uniqueInstance = new LogFile();
    }
    return uniqueInstance;
}

LogFile::LogFile() {}

LogFile* LogFile::uniqueInstance = 0;

```

**4.2: Examples of Creational Patterns**

## Case study of Singleton Pattern (Cont.)

- Consider extended scenario, where a class which creates Log Book for recording all transactions in a bank application.
- Now only three instance of this class has to be created, for each kind of account: Savings, Current and Recurring accounts
- What we have now is called a Multiton pattern in contrast to Singleton

The diagram shows a UML Class named LogBook. It has three fields: fp, lineNumber, and numofinstance. It has three methods: ~LogBook (destructor), LogBook (constructor with one overload), and writeData.

January 18, 2016 | Proprietary and Confidential | - 21 -

**Capgemini**  
CONSULTING IT SERVICES

```
#include <iostream>
#include <fstream>
#define COUNT 3 //update this number to convert into singleton/multiton pattern
using namespace std;

class LogBook {
public:
    LogBook() {
        if(numOfInstance==COUNT) {throw "Cannot create furthermore instances";}
        fp.open("c:\temp\log.txt",fstream::in||fstream::out);
        lineNumber=0;
        numOfInstance++;
    }
    LogBook(char path[],int val) {
        if(numOfInstance==COUNT) {throw "Cannot create furthermore instances";}
        lineNumber=val;
        fp.open(path,fstream::in||fstream::out);
        numOfInstance++;
    }
    void writeData(char data[]) {
        fp<<"Line."<<lineNum<<" "<<data<<endl;
        lineNumber++;
    }
    ~LogBook() {
        fp.close();
        numOfInstance--;
    }
private:
    static unsigned numOfInstance;
    fstream fp;
    int lineNumber;
};

unsigned LogBook::numOfInstance=0;
int main()
{
    LogBook l1,l2,l3;
    l1.writeData("message 1");
    l1.writeData("message 2");
    l1.writeData("message 3");

    try {
        LogBook l4("d:\temp\log.txt",20);
        l4.writeData("this is temp message");
    }
    catch(char *msg) {cout<<msg}
}
```

4.3: Examples of Structural Patterns

## What are Structural Design Patterns

- **Structural patterns describe how objects and classes can be combined to form larger structures**
  - Structural class patterns use inheritance to compose interfaces or implementations
  - Structural object patterns describe how objects can be organized to work with each other to form a larger structure
- **Structural Design patterns are further classified as:**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

January 18, 2016 | Proprietary and Confidential | + 12 +

 Capgemini  
CONSULTING IT SERVICES

### Structural Patterns:

Structural Patterns describe how objects and classes can be combined to form larger structures.

Class patterns describe abstraction with the help of “inheritance” and how it can be used to provide more useful program interface.

Object patterns, on other hand, describe how objects can be associated and composed to form larger, more complex structures.

Structural Patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

4.3: Examples of Structural Patterns

## Facade Pattern

- Facade Pattern provides a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- You can wrap a complicated subsystem with a simpler interface.
- They are normally Singleton as only one Facade object is required.

January 18, 2016 | Proprietary and Confidential | - 23 -

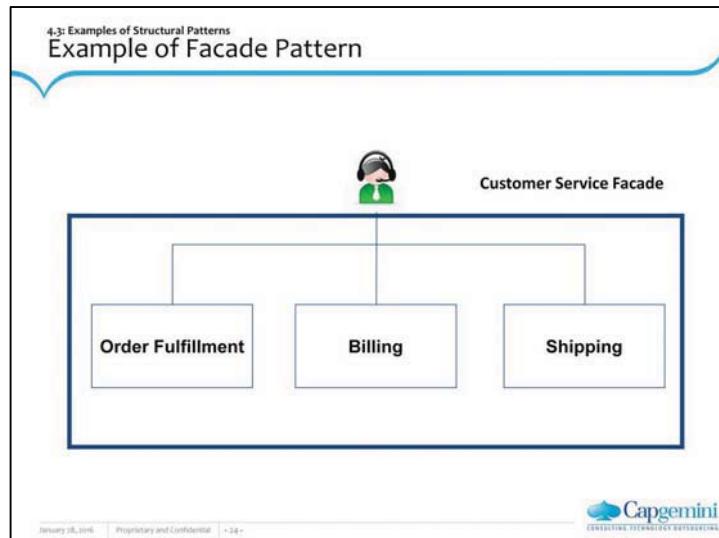
 Capgemini  
CONSULTING IT SERVICES

### Facade Pattern:

Facade encapsulates a complex subsystem within a single interface object. This reduces the learning curve that is necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it limits the features and flexibility that are needed by “power users”.

The Facade Design Pattern is used to provide a high-level interface that makes the subsystem easier to use. It helps create a unified interface to a set of interfaces in the subsystem.

Facade objects are often Singletons because only one Facade object is required.

**Example of Façade Pattern:**

Consumers encounter a Facade while ordering from a catalog. The consumer calls one telephone number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.

**4.3: Examples of Structural Patterns**

### Case study of Façade pattern

- Consider a class to be constructed which has functions to compute Maximum of three factorials, and product of two cubes
- Assume you are given three types of calculator class, as shown below
- We now need a Façade to solve our problem

```

classDiagram
    class CalcFacade {
        Fields: a, b, c
        Methods: CalcFacade(+), maxOfThreeFact, prodOfCubes
    }
    class TernaryCalculator {
        Fields: x, y, z
        Methods: maxOfThree, minOfThree, setX, setY, setZ, TernaryCalculat...
    }
    class BinaryCalculator {
        Fields: x, y
        Methods: add, BinaryCalculato..., div, floorDiv, mod, mul, setX, setY, sub
    }
    class UnaryCalculator {
        Fields: x
        Methods: absolute, cube, factorial, setX, square, UnaryCalculato...
    }
    CalcFacade "1" --> "1" TernaryCalculator
    CalcFacade "1" --> "1" BinaryCalculator
    CalcFacade "1" --> "1" UnaryCalculator
  
```

January 18, 2016 | Proprietary and Confidential | - 25 -

**Capgemini**  
CONSULTING IT SERVICES

```

#include <iostream>
using namespace std;
class UnaryCalculator{
public:
    UnaryCalculator() {x=0;}
    UnaryCalculator(unsigned val) {x=val;}
    void setX(unsigned val) {x=val;}
    unsigned abs() const {if(x<0) return 0; //Code for factorial}
    int absolute() {if(x<0) return -x;}
    unsigned square() {return x*x;}
    unsigned cube() {return x*x*x;}
private:
    unsigned x;
};
class BinaryCalculator {
public:
    BinaryCalculator() {x=y=z=1;}
    BinaryCalculator(unsigned a,unsigned b) {x=a;y=b;z=1;}
    void setX(unsigned val) {x=val;}
    void setY(unsigned val) {y=val;}
    unsigned add() const {return x+y;}
    int sub() const {return x-y;}
    unsigned mult() const {return x*y;}
    int mod() const {if(y==0) return x; //Code for factorial}
    if(y>0) return x%y;
    else return -x;
}
    float div() const {
        if(y==0) return x/y;
        else return 1;
    }
private:
    unsigned x,y,z;
};
class TernaryCalculator{
public:
    TernaryCalculator(unsigned val1,unsigned val2,unsigned val3){x=(val1)y=(val2)z=(val3);}
    void setX(unsigned val) {x=val;}
    void setY(unsigned val) {y=val;}
    void setZ(unsigned val) {z=val;}
    unsigned maxOfThree() {
        if(x>y&&y>z) return x;
        if(y>z&&z>x) return y;
        if(z>x&&x>y) return z;
    }
}
  
```



January 18, 2016 Proprietary and Confidential • 16 •



```
unsigned minOfThree() {
    if(x<=y&&y<=z) return x;
    if(y<=z&&z<=x) return y;
    if(z<=x&&x<=y) return z;
}
private:
unsigned x,y,z;
}
/*We need a fascade to simplify and present commonly used max of three factorials, and product of
cubes functions*/
class CalcFascade {
public:
CalcFascade(){a=b=c=1;}
CalcFascade(unsigned val1,unsigned val2,unsigned val3):a(val1),b(val2),c(val3){}
unsigned maxOfThreeFact() {
    UnaryCalculator uc1(a),uc2(b),uc3(c);
    TernaryCalculator tc(uc1.factorial(),uc2.factorial(),uc3.factorial());
    return tc.maxOfThree();
}
unsigned prodOfCubes(){
    UnaryCalculator uc1(a),uc2(b);
    BinaryCalculator bc(uc1.cube(),uc2.cube());
    return bc.mul();
}
private:
unsigned a,b,c;
}
int main()
{
CalcFascade cf1(5,4,10),cf2(22,44,1);
cout<<"Max of factorial of three values:"<<cf1.maxOfThreeFact()<<endl;
cout<<"Product of cubes!"<<cf2.prodOfCubes();
int i;cin>>i;//screen wait
}
```

4.4: Introduction to Behavioral Patterns

## What is a Behavioral Design Pattern

➤ Behavioral patterns are those design patterns which deal with interactions between the objects.

- Behavioral patterns are concerned with the assignment of responsibilities between objects, or encapsulating behavior in an object and delegating requests to it.

➤ Here is a list of the different behavioral design patterns:

<ul style="list-style-type: none"><li>— Chain of Responsibility (COR)</li><li>— Command</li><li>— Interpreter</li><li>— Iterator</li><li>— Mediator</li><li>— Memento</li></ul>	<ul style="list-style-type: none"><li>— Observer</li><li>— State</li><li>— Strategy</li><li>— Template Method</li><li>— Visitor</li></ul>
---	---

January 18, 2016 | Proprietary and Confidential | 22 | Capgemini CONSULTING IT SERVICES

### Behavioral Patterns:

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not only patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow, which is difficult to follow at run-time. These patterns shift your focus away from flow of control, and let you concentrate just on the way objects are interconnected.

Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use “object composition” rather than “inheritance”. There are 11 different behavioral design patterns. They are Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

3.4: Introduction to Behavioral Patterns

## State Pattern

- State Pattern allows an object to alter its behavior when its internal state changes. The object appears to change its class.
- It uses polymorphism to define different behaviors for different states of an object.

January 18, 2016 | Proprietary and Confidential | - 18 -

 Capgemini  
CONSULTING IT SERVICES

### State Pattern:

State pattern is a behavioral type design pattern which is widely used in different applications. It is especially used in 3D-Graphics applications and applications for devices. In object-oriented design, object can change its behavior based on its current state. A state pattern design implements state and behavior of an object. By using state pattern, we can reduce the complexity in handling different states of an object. This helps to easily maintain code in future.

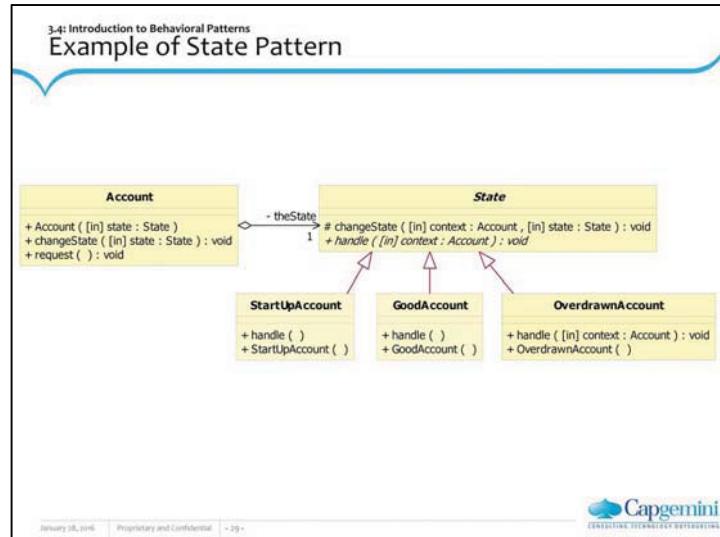
A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Alternatively, an application is characterized by large and numerous case statements that vector flow of control, based on the state of the application.

The State pattern does not specify the location where the state transitions will be defined. There are two choices:

The “context” object

Each individual State derived class

The advantage of the latter option is ease of adding new State derived classes. The disadvantage is that each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.



An account holder has an account with a bank and the account behaves differently depending on its balance state.  
The difference in behavior is delegated to State objects.

**4.4: Introduction to Behavioral Patterns**

### Case Study of State Pattern

➤ Consider three states for a (fixed capacity) Stack of integers as depicted below

➤ Transitions between the states are triggered by the actions indicated on the arrows

```

graph LR
    EMPTY((EMPTY STACK)) -- "First push" --> OPERATIONAL((OPERATIONAL STACK))
    OPERATIONAL -- "Last push" --> FULL((FULL STACK))
    FULL -- "push" --> OPERATIONAL
    OPERATIONAL -- "First pop" --> FULL
    FULL -- "Last pop" --> EMPTY
    EMPTY -- "pop" --> EMPTY
    OPERATIONAL -- "Push/Pop" --> OPERATIONAL
  
```

January 18, 2016 | Proprietary and Confidential | + 30 +

Capgemini CONSULTING IT SERVICES

```

StatePatternDemo.h
#define MAXSIZE 4
class baseStack{
public:
    virtual char* state() const = 0;
    virtual baseStack* push(int val) = 0;
    virtual baseStack* pop() = 0;
protected:
    int tos;
    int stk[MAXSIZE];
};

class EmptyStack: public baseStack {
public:
    EmptyStack();
    char* state() const;
    baseStack* push(int val);
    baseStack* pop();
    ~EmptyStack();
};

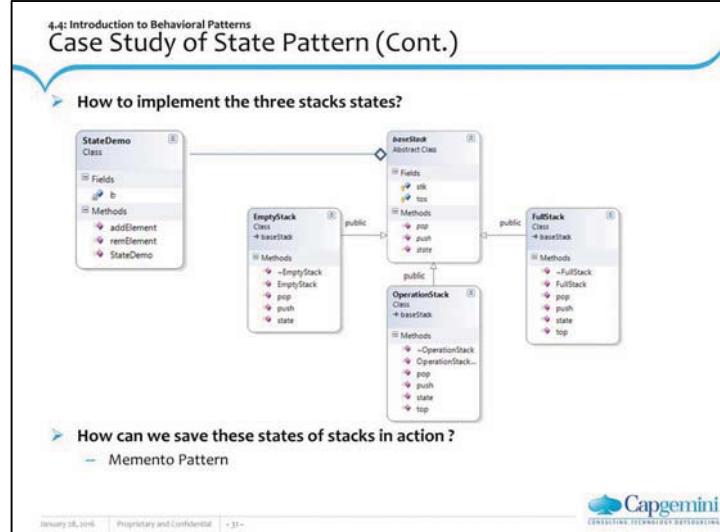
class OperationStack:public baseStack {
public:
    OperationStack(int val);
    OperationStack(int val,int size);
    char* state() const;
    baseStack* push(int val);
    baseStack* pop();
    int top();
    ~OperationStack();
};

class FullStack: public baseStack{
public:
    FullStack(int al[],int size);
    char* state() const;
    int top();
    baseStack* pop();
    baseStack* push(int val);
    ~FullStack();
};

StatePattern.cpp
#include <iostream>
#include "StatePatternDemo.h"
using namespace std;

/*Code related to actions on empty Stack*/
EmptyStack::EmptyStack() {tos=-1;cout<<"created empty stack\n";}

  
```



```

char* EmptyStack::state() const {return " Empty Stack ";}
baseStack* EmptyStack::push(int val) {delete this; return new OperationStack(val);} //Once a element is inserted it goes to operational state
baseStack* EmptyStack::pop() {return this;} //notice no action change in state because of pop
EmptyStack::~EmptyStack() {cout<<" destroyed empty stack\n" << endl;}

/*Code related to actions on operational Stack*/
OperationStack::OperationStack(int val){
    tos=0;
    stk[tos]=val;
    cout<<"created op stack\n";
}
OperationStack::OperationStack(int a[],int size) {
    for(int i=0,tos=size;i<size;i++) stk[i]=a[i];
    cout<<" created op stack\n";
}
char* OperationStack::state() const {return " Op Stack ";}
baseStack* OperationStack::push(int val) {
    ++tos;
    if(tos>MAXSIZE) {stk[tos]=val;return this;}
    else {
        delete this;
        return new FullStack(stk,-tos);
    } //once the capacity has reached it is now in full state
}
baseStack* OperationStack::pop() {
    --tos;
    if(tos<=0) return this;
    else {
        delete this;
        return new EmptyStack();
    } //once all elements are removed it is now in empty state
}
int OperationStack::top() { return stk[tos];}
OperationStack::~OperationStack(){
    cout<<" destroyed op stack\n" << endl;
}
  
```



January 18, 2016 | Proprietary and Confidential | 32 •



```
/*Code related to actions on Full Stack*/
FullStack::FullStack(int a[],int size) {
    for(int i=0;tos<size;i<=tos;i++) stk[i]=a[i];
    cout<<" created full stack\n";
}
char* FullStack::state() const { return " Full Stack ";}
int FullStack::top() { return stk[tos];}
baseStack* FullStack::pop() {
    delete this;
    return new OperationStack(stk,-tos);
} //one element removed, back to operational state
baseStack* FullStack::push(int val) {return this;} //notice no action change in state because of push
FullStack::~FullStack(){cout<<" destroyed full stack\n";}
class StateDemo {
public:
StateDemo() { b = new EmptyStack();}
void addElement(int val) {
    cout<<"Before: "<<b->state();
    b->push(val);
    cout<<"After: "<<b->state()<<endl;
}
void remElement() {
    cout<<"Before: "<<b->state();
    b->pop();
    cout<<"After: "<<b->state()<<endl;
}
private:
baseStack *b;
};
int main() {
StateDemo s; //empty stack state
s.remElement(); //empty stack state
s.remElement(); //empty stack state
s.addElement(10); //op stack state
s.addElement(20); //op stack state
s.addElement(30); //op stack state
s.addElement(40); //op stack state
s.addElement(50); //full stack state
s.addElement(60); //full stack state
s.addElement(70); //full stack state
s.remElement(); // op stack state
int i;cin>>i;//screen wait
}
```

4.4: Introduction to Behavioral Patterns

## Strategy Pattern

- Strategy Pattern defines a family of algorithms, encapsulate each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- It looks similar to state pattern but intent is to encapsulate interchangeable behavior which is not state-based.

January 18, 2016 Proprietary and Confidential • 33 •

 Capgemini  
CONSULTING IT SERVICES

### Strategy Pattern:

The Strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to:

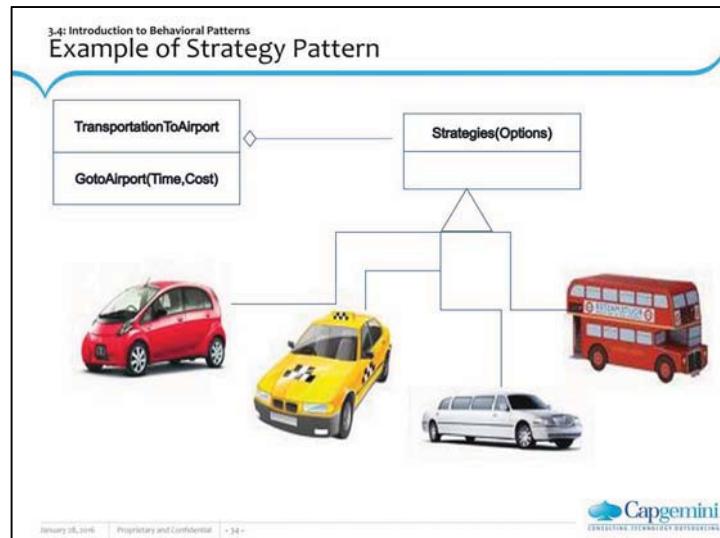
Provide a means to define a family of algorithms

Encapsulate each one as an object, and make them interchangeable

The strategy pattern lets the algorithms vary independently from clients that use them.

The Strategy pattern uses “composition” instead of “inheritance”. It is a good idea to use the Strategy Pattern when you have several objects that are basically the same, and differ only in their behavior. By using Strategies, you can reduce these “several objects” to “one class” that uses several Strategies. The use of strategies also provides a nice alternative to sub-classing an object to achieve different behaviors. When you subclass an object to change its behavior, the behavior that it executes is static.

Strategy is a bind-once pattern, whereas State is more dynamic.



#### Example of Strategy Pattern:

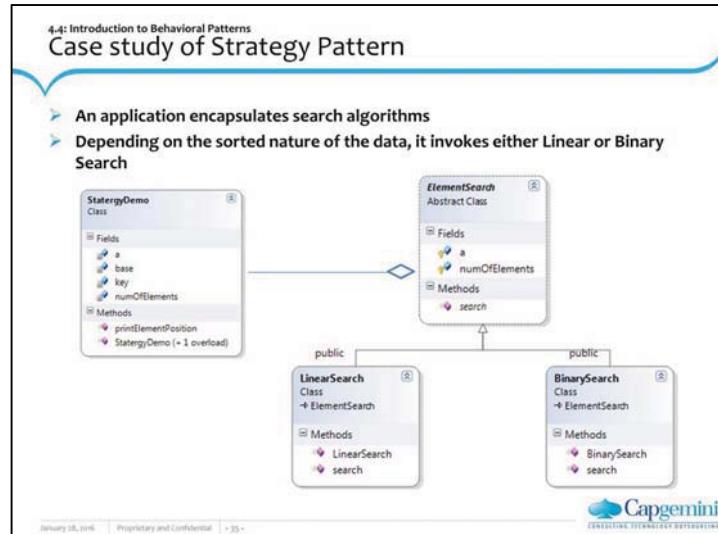
A Strategy defines a set of algorithms that can be used interchangeably.

Modes of transportation to an airport is an example of a Strategy.

Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.

For some travelers airports, subways, and helicopters are also available as a mode of transportation to the airport.

Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on tradeoffs between cost, convenience, and time.



```

#include <iostream>
#define ARRAYSIZE 10
using namespace std;
class ElementSearch {
public: virtual int search(int key)=0;
};

class LinearSearch:public ElementSearch {
public: LinearSearch(int arr[],int size) {
    for(int i=0;i<size;i++) a[i]=arr[i];
    numElements=size;
}
int search(int key) { //code for linear search
private: int a[ARRAYSIZE],numOfElements;
};

class BinarySearch:public ElementSearch{
public: BinarySearch(int arr[],int size) {
    for(int i=0;i<size;i++) a[i]=arr[i];
    numElements=size;
}
int search(int key) { //code for binary search
private: int a[ARRAYSIZE],numOfElements;
};

class StrategyDemo {
public: StrategyDemo() {
    numElements=5; key=-1;
}
StrategyDemo(int arr[],int size,bool isSorted,int val) {
    for(int i=0;i<size;i++) a[i]=arr[i];
    numElements=size;
    key=val;
    /*Calling binary OR linear search depending on sorted status of the array*/
    if(isSorted) base=new BinarySearch(a,numOfElements);
    else base=new LinearSearch(a,numOfElements);
}
int printElementPosition() {
/*statically common function*/
int pos=base->search(key);
if(pos>-1) cout<<"Key found at "<<pos<<endl;
else cout<<"Key not found"<<endl;
}
private: ElementSearch *base;
int a[ARRAYSIZE],numOfElements,key;
};

int main() {
int arr[]={10,20,30,40,50,60};
int val=10,20,30,40,50;
StrategyDemo e1(a,6,true,60);
StrategyDemo e2(a,6,false,60);
e1.printElementPosition();
e2.printElementPosition();
}
  
```

## Summary

➤ In this lesson, you have learnt implementations of :

- Delegation Pattern,
- Abstract Factory Method, Singleton
- Façade
- State, Strategy



## Review Question

### Fill in Blanks

1. Behavioral patterns are those design patterns which deal with \_\_\_\_\_ between the objects.
2. State Pattern allows an object to alter its behavior when its \_\_\_\_\_ changes. The object appears to change \_\_\_\_\_.
3. Strategy Pattern defines a \_\_\_\_\_, encapsulate each one, and makes them interchangeable.





# C++ Programming

## Lab Book

---

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

## Document Revision History

---

Date	Revision No.	Author	Summary of Changes
June 2005	1.0		
Sept 2009	2.0D	Liji Shyuu	Content Upgradation
10-oct-2009	2.0D	CLS Team	Review
22-Jun-2011	3.0D	Rathnajothi Perumalsamy	Revamp/Refinement

## Table of Contents

<i>Document Revision History</i> .....	2
<i>Table of Contents</i> .....	1
<i>Getting Started</i> .....	3
Overview.....	3
Setup Checklist for C++ Programming.....	3
Instructions .....	3
Learning More (Bibliography).....	3
<i>Lab 1. Introduction</i> .....	4
1-1: Introduction to Visual Studio Environment .....	4
1-2: Simple Program .....	11
1-3: Enumerated Data Types and Scope Resolution Operator .....	13
1-4: Reference Variable .....	14
1-5: Dynamic Memory.....	16
<<TODO>> .....	18
<i>Lab 2. Functions</i> .....	19
2.1: Inline and Overloaded Functions and Default Arguments.....	19
<<TODO>> .....	21
<i>Lab 3. Classes</i> .....	22
3.1 Introduction to Classes .....	22
<<TODO>> .....	25
<i>Lab 4. Constructors and Destructors</i> .....	28
4.1 Constructor and Destructor Functions.....	28
4.3 Friend Class .....	30
<<TODO>> .....	32
<i>Lab 5. Operator Overloading</i> .....	35
5.1 Overloading + Operator .....	35
<<TODO>> .....	36
<i>Lab 6. Inheritance and Runtime Polymorphism</i> .....	39
6.1 Inheritance .....	39
6.2 Runtime Polymorphism .....	41
6.3 Program using virtual function .....	43
<<TODO>> .....	44

---

<i>Lab 7. String Class</i> .....	47
<<TODO>> .....	47
<i>Lab 8. Exception Handling</i> .....	48
8.1 Exception Handling .....	48
<<TODO>> .....	48
<i>Lab 9. Templates</i> .....	49
9.1 Function Template .....	49
9.2 Class Template .....	49
<<TODO>> .....	50
<i>Lab 10. Input Output in C++</i> .....	51
10.1 File Handling .....	51
<<TODO>> .....	52
<i>Appendices</i> .....	53
APPENDIX A: C++ Programming Standards .....	53
APPENDIX B: Debugging Examples .....	61
<i>Table of Figures</i> .....	70
<i>Table of Examples</i> .....	71

## Getting Started

---

### Overview

This lab book is a guided tour for learning C++ Programming. It comprises solved examples and 'To Do' assignments. Follow the steps provided in the solved examples and work out the 'To Do' assignments that are given.

### Setup Checklist for C++ Programming

Here is what is expected on your machine for the lab to work.

#### Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 512MB of RAM (1GB or more recommended)

#### Please ensure that the following is done:

- Visual studio 2005 is installed on your machine.

### Instructions

- Create a directory by your name in drive <drive>. In this directory, create a subdirectory cpp\_assgn. For each lab exercise create a directory as lab <lab number>.
- For all coding standards refer Appendix A. All lab assignments should refer coding standards

### Learning More (Bibliography)

- Programming with C++; by John R. Hubbard
- Let us C++; by Yashwant Kanetkar.

## Lab 1. Introduction

---

Goals	<ul style="list-style-type: none"><li>Get familiar with Visual Studio Environment and learn basics of CPP programming from simple programs.</li></ul>
Time	180 minutes

### 1-1: Introduction to Visual Studio Environment

#### Introduction

This brief contains information on creating, compiling, and executing a simple program in the Visual studio 2005 environment. The best approach for getting familiar with the steps mentioned below is to work through a simple program.

What is Developer Studio?

#### Developer Studio

- Developer Studio is a completely self-contained environment for creating, compiling, linking, and testing windows programs.
- It is the Integrated Development Environment (IDE) that comes with Visual C++ and is a great environment in which to learn.
- The Developer Studio incorporates a range of fully integrated tools designed to make the whole process of writing programs easy.

The fundamental parts of Visual C++, provided as part of the Developer Studio, are the editor, compiler, linker, and the libraries.

The program development and execution will be performed from within the developer studio. When you start Visual C++, assuming no project was active when you shut it down last, you will see the following screens listed in the Solution section.

#### Solution:

**Step 1:** From the **Start** menu, navigate to **Programs → Microsoft Visual Studio 2005**.

---

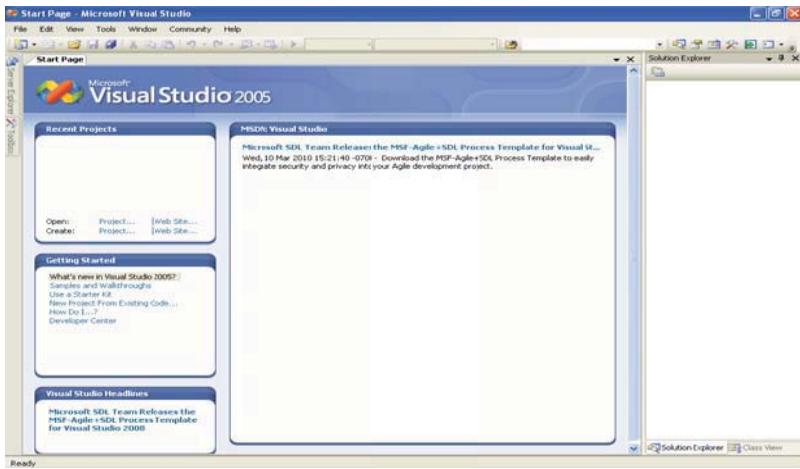


Figure-1: Visual Studio – start Page

**Step 2:** Go to **File → New Menu**, select **Project**.

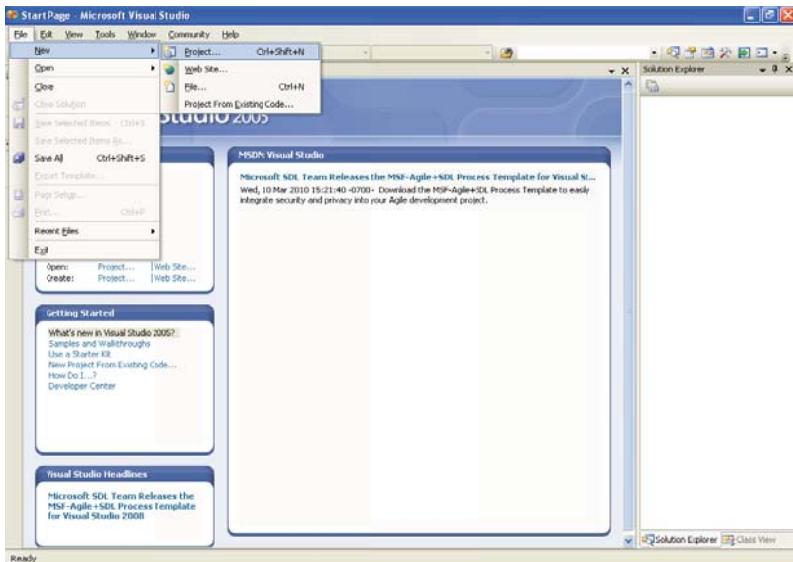


Figure 2: Creating New Project in Visual Studio

**Step3:** Under Project Types, select Visual C++ → Win32 → Win32 Project.

1. In the **Name** text box, type **sample**.
2. In the **Location** text box, type **C:\CPP\_Programming** and then click **OK**.

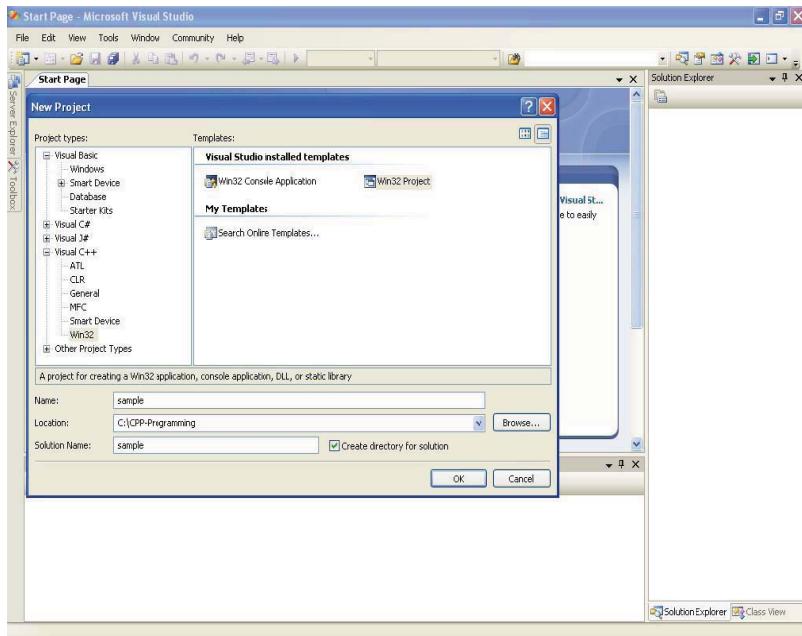
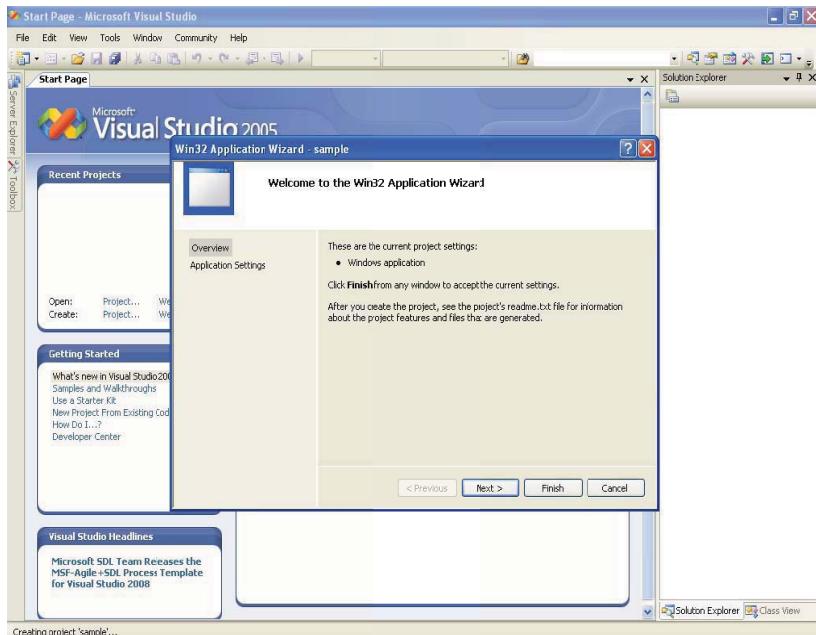


Figure 3: Selecting the project type

**Step 4:** When the following screen gets displayed, click **Next**.



**Figure-4: Accepting Current Setting of the project**

**Step 5:** In the **Win32 Application Wizard** dialog box, click **Application Settings** in the left pane.

1. Under **Application Type**, click to select any of the following options:
  - a. **Console application:** Creates a simple console application. The application files include a .cpp file that contains an empty **main** function.
  - b. **Windows application:** Creates a simple Microsoft Windows-based application. The application files include a **ProjectName.cpp** file that contains a **\_tWinMain** function. You can use this type of application to perform graphical user interface (GUI) based programming.
  - c. **DLL:** Creates a 32-bit Windows-based DLL application project.
  - d. **Static library:** Creates a 32-bit Windows-based DLL application project.

- e. To create a simple "Hello World!" program, click to select **Console application**.

2. Under **Additional options**, click to select the **Empty project** check box.

**Note:** To create a simple "Hello World!" program, do not select this check box.

Instead, let the wizard generate the code for you.

3. Click **Finish**.

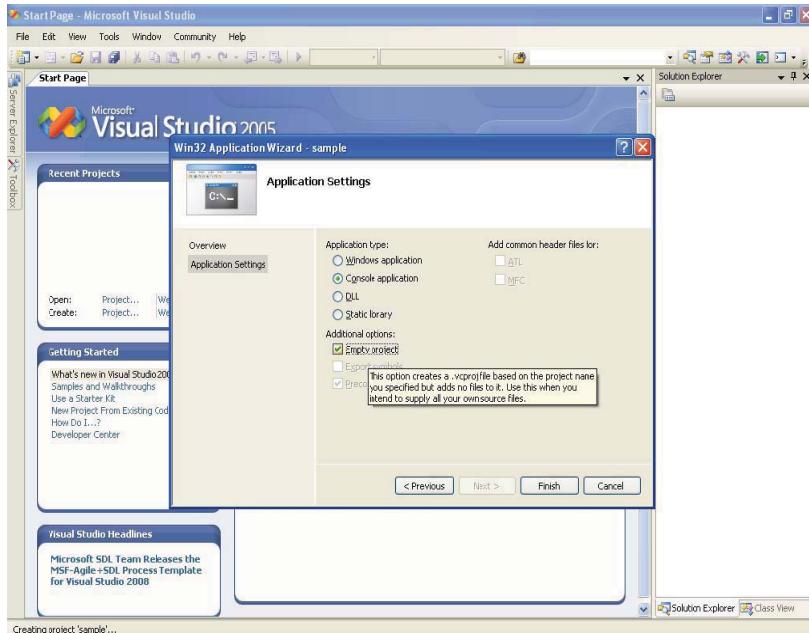


Figure 5: Selecting Application Type

**Step 6:** In Solution Explorer, right-click the **Source Files** folder, point to **Add**, and then click **Add New Item**.

- a) In **Add New Item - Sample** dialog box, click on **code** under **Visual C++** in the left pane and click **C++ File (.cpp)** under **Templates**.

- b) In the **Name** text box, type **sample-example.cpp**, and then click **Add**.
- c) In the **sample-example.cpp** code window, write the following code:

The screenshot shows the Microsoft Visual Studio IDE interface. The main window displays the code for 'sample-example.cpp' in the 'Global Scope'. The code is as follows:

```
#include<iostream>
using namespace std;
void main()
{
    cout<<"Welcome";
}
```

The Solution Explorer on the right shows a project named 'sample' containing 'Header Files', 'Resource Files', and 'Source Files' with 'sample-example.cpp' listed. The Properties window is also visible.

**Figure 6: Welcome Program**

**Step 7:** On the **Build** menu, click **Build Solution**. On the **Debug** menu, Click **Start without Debugging** to run the application or Press **CTRL+F5**. You will receive the following message in a Command Prompt window:

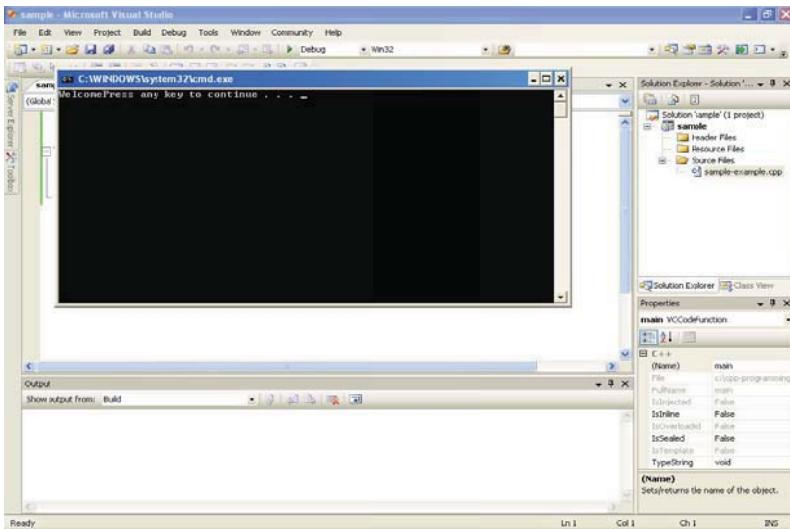


Figure 7: Output Screen

### 1-2: Simple Program

Write a simple program to accept name and age of a person. Display the details of the person's age in the next year.

#### Solution:

**Step 1:** Write the following code in simple.cpp.

Example 1: Simple.cpp

---

### Step 2: Analyze the output of simple.cpp.

The screenshot shows the Microsoft Visual Studio interface. In the center, there's a code editor window for 'simple.cpp' containing C++ code. To the right of the code editor is a 'Command Prompt' window titled 'C:\WINDOWS\system32\cmd.exe' showing the execution of the program. Below the code editor is an 'Output' window displaying the build log. On the right side of the interface, there's a 'Properties' manager window for a 'main' object, showing details like file name, full name, and type. The status bar at the bottom indicates a successful build.

```
#include <iostream>
using namespace std;
void main()
{
    char cName[10];
    cout<<"Please enter your name";
    cin>>cName;
    int iAge;//I can declare
    cout<<"Please enter your age";
    cin>>iAge;
    cout<<"Hi "<<cName<<" , r
}
```

Figure 8: Output of simple.cpp file

We have learnt that:

1. C++ supports C style as well as single line comments.
2. The main keyword is followed by void, if main() is not returning any value.
3. C++ let us declare a variable, after executable statements (It is unlike C).
4. cin and cout are used for reading the data from keyboard, and writing to the screen respectively.
5. <iostream> is the C++ label for a standard header file for input and output streams
6. C++ provides a namespace for including code from its standard library
7. We can have cascaded cout to display more than one message or variable  
(Similarly we can have cascaded cin).

8. Manipulators can be used with cout for formatted output.

#### **1-3: Enumerated Data Types and Scope Resolution Operator**

Write a program to display the working days and a holiday using enumerated data type and a scope resolution operator.

**Solution:**

**Step 1:** Write the following code in enumerated.cpp.

**Example 2: enumerated.cpp**

**Step 2:** Analyze the output of enumerated.cpp.

```

#include <iostream>
using namespace std;
int iDay = 7;
void main()
{
    enum days_of_week {Sun,Mo,Tu,Wed,Th,Fri,Sat};
    //iDay is local var
    for(int iday=0;iday<::id
    {
        if (iday==Sun) // same
            cout<<"Holiday!!!!"<<endl
        else
            cout<<"Oh Not Working Day!"<<endl;
    }
}

```

Figure 9: Output of enumerated.cpp

We have learnt that:

1. We can have a same variable name in two different scopes. Local variable has more priority over the global variables. To be able to access global variable, we can use a scope resolution operator.
2. Enumerated data type would give integer values (starting with 0 by default) to a set of variables. You can set this initial value to some other value (can be negative), next variable in the set would take next value and so on.

#### 1-4: Reference Variable

Write a program to swap value of two numbers using references.

**Solution:**

**Step 1:** Write the following code in reference.cpp.

**Example 3: reference.cpp**

**Step 2:** Analyze the output of reference.cpp.

```

#include <iostream>
using namespace std;
void swap_values(float& a,
{
    float temp;
    temp = a;
    a = b;
    b = temp;
}
void main(void)
{
    float big = 2.5;
    float small = 1.5;
    float& big_alias = big;
    float& small_alias = small;
    cout<<"Before swapping"
        << endl;
    cout<<Big<<endl;
    cout<<Small<<endl;
    cout<<"After swapping"
        << endl;
    cout<<Big<<endl;
    cout<<Small<<endl;
}

```

Output:

```

Show output from: Build
1>reference.cpp
1>Generating Code...
1>Linking...
1>Adding manifest...
1>Build log was saved at "file:///c:/CPP-Programming\sample\sample\Debug\BuildLog.htm"
1>sample - 0 error(s), 0 warning(s)
=====
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped

```

Figure 10: Output of reference.cpp

We have learnt that:

- 1) A reference is another name (reference) given to our variable.
- 2) No separate memory is allocated for a reference variable.
- 3) Any change made to original variable is reflected in the referenced variable and vice a versa.
- 4) A reference can also be constant. A constant reference cannot be changed or updated. The variable being referred can be changed or updated and the reference would get the updated value of the variable it is referring to.

## 1-5: Dynamic Memory

Write a program to create an array dynamically and display it.

- 1) An array size is accepted as a character and then converted to the integer using atoi.
- 2) An array element is accepted as character and then converted to long using atoll.
- 3) You can directly accept array size as integer, and array elements as long.

**Solution:**

**Step 1:** Write the following code in dynamic\_memory.cpp.

**Example 4: dynamic\_memory.cpp**

**Step 2: Analyze the output of dynamic\_memory.cpp.**

```
#include <iostream>
using namespace std;
int main ()
{
    char input [100];
    int i,n;
    long * l;
    cout << "How many numbers
    cin.getline (input,100);
    i=atoi (input);
    l= new long[i];
    if (l == NULL)
        exit (1);
    for (n=0; n<i; n++)
    {
        cout << "Enter number:
    }
```

Output:

```
1>Compiling...
1>dynamic_memory.cpp
1>Linking...
1>Embedding manifest...
1>Build log was saved at "file:///c:/CPP-Programming/sample/sample/Debug/BuildLog.htm"
1>sample - 0 error(s), 0 warning(s)
=====
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

**Figure 11: dynamic\_memory.cpp**

We have learnt that:

- 1) Dynamic memory is allocated at run time using new operator.
- 2) You need appropriate pointer for allocating dynamic memory.
- 3) Dynamic memory should be deallocated, when not required. So that it can be reused by same or other programs.

<<TODO>>

**Assignment-1:** Write a program that declares a string pointer initialized to some value, and then proceed to put the characters in reverse order. Do not create temporary string.

**Assignment-2:** Write a program to get array of values and perform four arithmetic calculations based on enumerated values like PLUS, MINUS, MULTIPLY and DIVIDE.

**Note:** Allocate array dynamically and deallocate it.

## Lab 2. Functions

---

Goals	<ul style="list-style-type: none"><li>• Write a program using Inline function.</li><li>• Write a program using Function Overloading.</li></ul> <p>Write a program passing default arguments to functions.</p>
Time	60 minutes

### 2.1: Inline and Overloaded Functions and Default Arguments

Write a program to find out maximum of two integers and two characters.

**Solution:**

Step 1: Write the following code, and save it as Function.cpp.

**Example 5: Function.cpp****Step 2: Analyze the output of Function.cpp.**

```
#include <iostream>
using namespace std;
int max(int ,int=10);
char max(char,char);
void main()
{
    int a=21, b=5;
    cout<<"Maximum of "<<a <
    cout<<"\n";
    int c=5;
    cout<<"Maximum of "<<c <
    cout<<"\n";
    char ch1='A', ch2='Z';
    cout<<"Maximum of "<<ch1
}
inline int max(int a,int b)
```

Output

```
1>Compiling...
1>Function.cpp
1>Linking...
1>Adding manifest...
1>Build log was saved at "file:///c:/CPP-Programming/sample/sample\Debug\BuildLoc.htm"
1>sample - 0 error(s), 0 warning(s)
***** Build: 1 succeeded, 0 failed, 3 up-to-date, 0 skipped *****
```

Build succeeded

**Figure 12: Output of Function.cpp**

We have learnt that:

- 1) You should declare small functions as inline.
- 2) It is up to compiler, whether to treat a function as inline or not.
- 3) C++ allows functions with same name, but different argument list.
- 4) The argument list may vary in the number of parameters, data types of the parameters, or sequence of different data types appearing in the signature of the function.
- 5) Return type of overloaded function may or may not be different.
- 6) A function may take some default values as arguments.
- 7) In absence of a parameter (in the function call) at respective position, the default value is considered, else ignored.

- 8) If we are having default arguments present in our functions, then they must be at the trailing positions in the argument list.

<<TODO>>

**Assignment-1:** Write a C++ program which will have two inline functions `sqr()` and `cube()`. These functions will return the square and cube value of the given input number.

**Assignment-2:** Write a C++ program with a function `repchar()` which accepts a given char and a numerical value. The function when called will print the given character so many number of times. If no arguments are passed to the function it will print \*, 5 times.

Invoke the function with the following:

- i. no arguments
- ii. Single char argument
- iii. Single int argument
- iv. a char and an int argument

## Lab 3. Classes

<b>Goals</b>	<ul style="list-style-type: none"> <li>Understand and create classes.</li> </ul>
<b>Time</b>	90 minutes

### 3.1 Introduction to Classes

Create a header file and instantiate an object of the class.

#### Solution:

Step 1: In Solution Explorer, right-click the Source Files folder, point to Add, and then click Class.

- In Add Class - Solution\_Lab3 dialog box, click on C++ under Visual C++ in the left pane and click C++ Class under Visual Studio installed Templates.

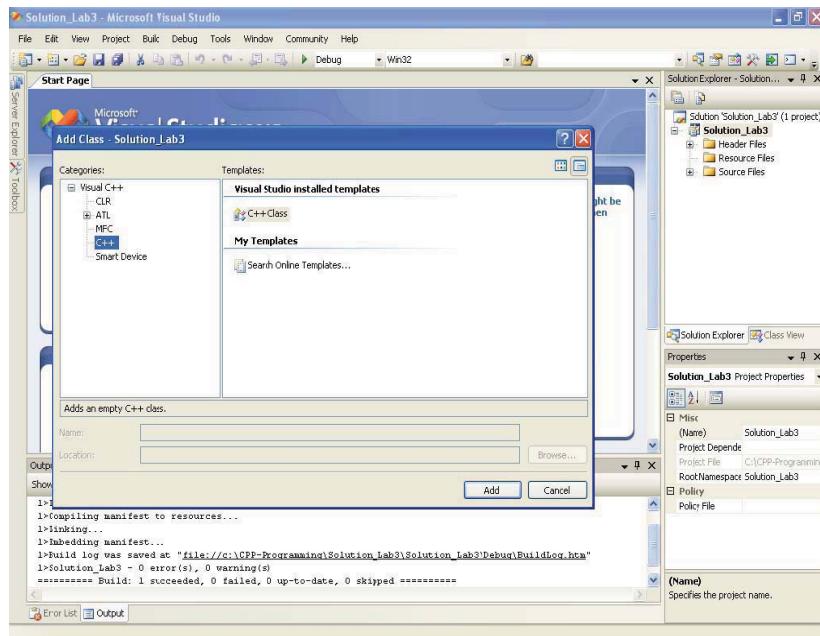


Figure 13: Selecting C++ class file.

- b) Click on Add.
- c) In the Generic C++ Class Wizard - Solution\_Lab3 Dialog Box, type MyClass in the class name text box and then click on finish.

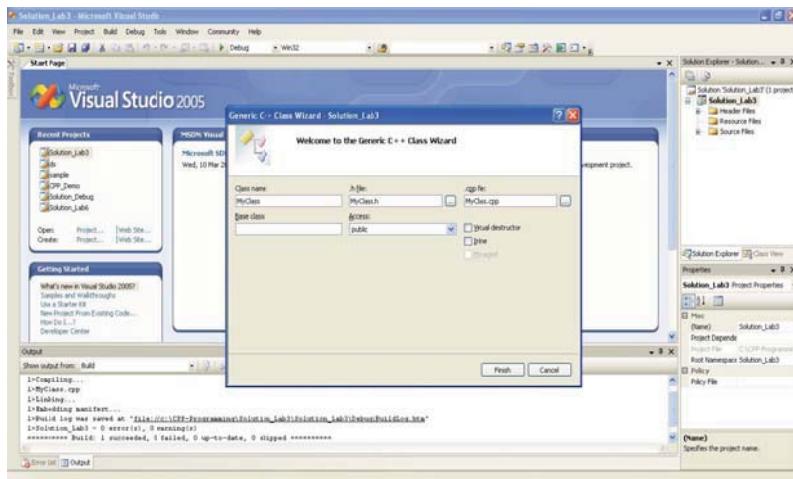


Figure 14: Creating .h and .cpp file

- d) In the MyClass.h code window, write the following code:

Example 6: Sample Code

- e) In the MyClass.cpp code window, Write the following code:

**Step 2: Analyze the output.**

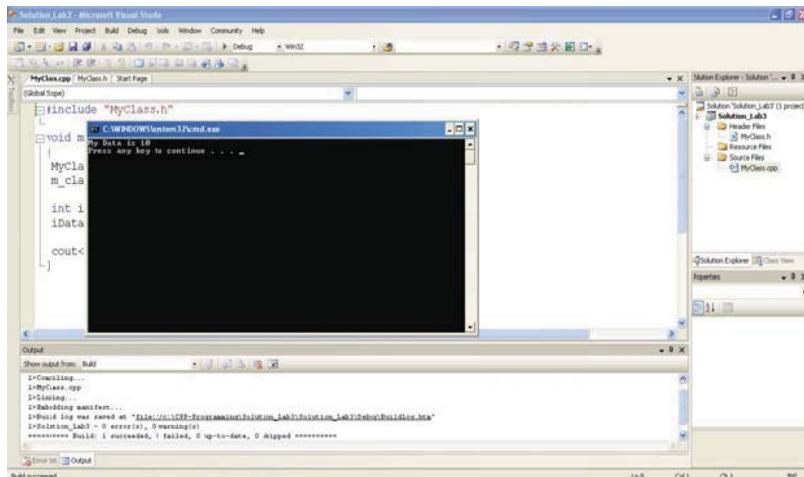


Figure 15: Output Screen

We have learnt that:

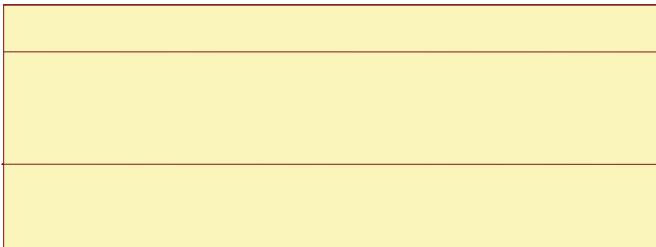
1. In simple words, a class is a user-defined data type. We need to create instances of the class to use it.
2. Instances of the class are called as objects. Different objects have different data values but same behavior.
3. A class contains data members and function members.

4. A class member function can be accessor (getMyData in our example, the one that only gets the value of the data member), mutator (setMyData in our example, the one that set the value to data member), or manager (constructors and destructors).

<<TODO>>

**Assignment 1:** Assume the following class diagram and write the skeleton of the program.

Class Diagram:



**Figure 16: Class Diagram for Student Class**

Solution:

**Example 8: Sample Code**

**Assignment 2:** Consider the following class diagram and Create a class for time consisting hours, minutes, and seconds as data members. Methods should be used to initialize and display them. Develop a method to increment time by one second.

Class Diagram:

---

Time
-iHours: int
-iMinutes: int
-iSeconds: int
+setTime(int, int, int): void
+getTime(void): void
+incrementTime(): void

Figure 17: Class Diagram for Time Class

## Lab 4. Constructors and Destructors

---

Goals	<ul style="list-style-type: none"><li>• Use Constructors and Destructors.</li><li>• Use Friend Functions and Friend Classes.</li></ul>
Time	120 minutes

### 4.1 Constructor and Destructor Functions

Consider the following class diagram and Write a program to calculate area of the rectangle by constructing and destroying a rectangle object.

Class Diagram:

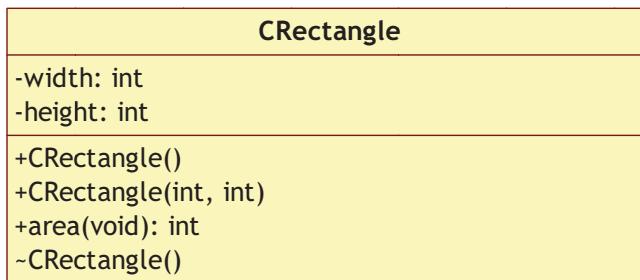


Figure 18: Class Diagram for CRectangle Class

**Solution:**

**Step 1:** Write the code based on the class diagram, and save it as **Construct.h**.

**Step 2:** Write the following code save it as Construct.cpp.

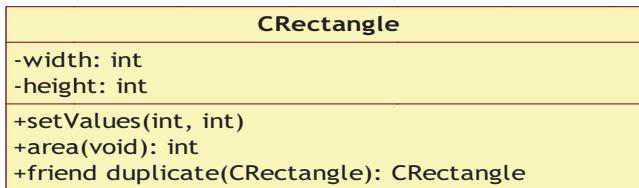

**Example 9: Construct.cpp**

**Step 3:** Analyze the output of Construct.cpp.

#### 4.2 Friend Functions

Consider the following class diagram and Write a program to change width and height (private members of class) of a rectangle object and find area of rectangle with new values.

Class Diagram:



**Figure 19:** Class Diagram for CRectangle with friend function

---

**Solution:**

**Step 1:** Write the code based on class diagram for CRectangle and save it as Rectangle.h.

**Step 2:** Write the following code save it as Rectangle.cpp.

Consider the following class diagram and Write a program to convert a square object into a rectangle object using friend class.

Class Diagram:

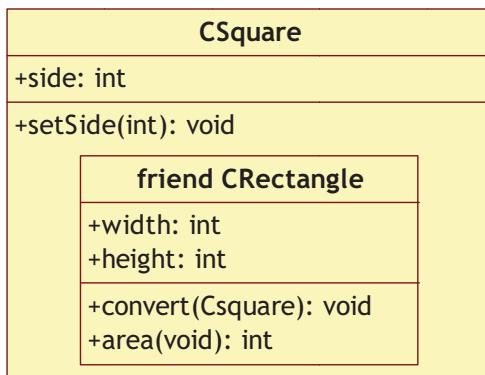


Figure 20: Class Diagram for friend Class

**Solution:**

**Step 1:** Write the following code, and save it as Lab4\_3.h.

**Step 2:** Write the following code, and save it as Lab4\_3.cpp.

Example 12: Lab4\_3.cpp

---

Step 3: Analyze the output of Lab4\_3.cpp.  
<<TODO>>

Assignment-1: Define a class employee by referring the below class diagram and display the employee details.

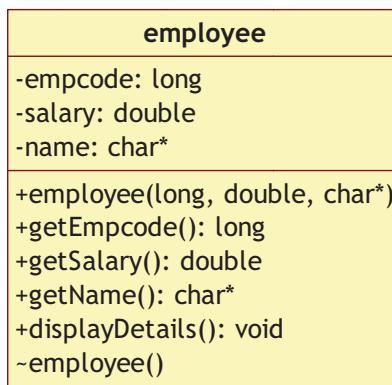


Figure 21: Class Diagram for Employee

Assignment-2: Define a class complex with reference to the below class diagram and implement a function to add two complex numbers.

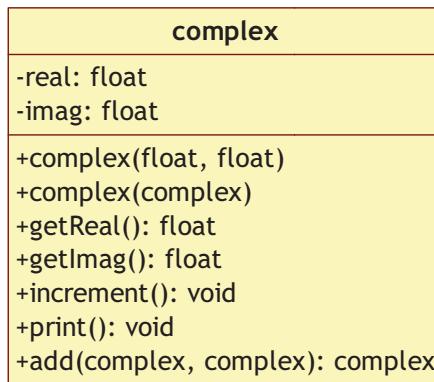


Figure 22: Class Diagram for complex

---

Assignment-3: Write a class called Point with three member variables x, y and z. The class should have a default constructor, a parameterized constructor with default arguments for z, and a copy constructor. The class should also have a method called negate to negate the member variables, and it should have a function called print to display the variables.

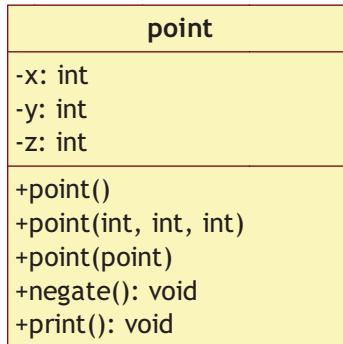


Figure 23: Class Diagram for point

Assignment-4: Refer to the below class diagram and Implement a String class. Each object of this class will represent a character string. Ensure that the access functions do not modify the member variables.

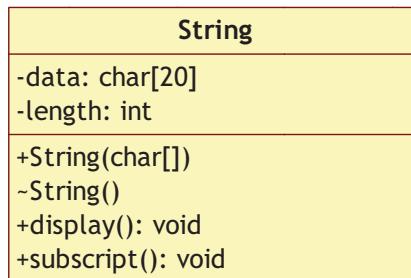


Figure 24: Class Diagram for String

Assignment-5: Refer the below class diagram and Write a friend function to exchange data of two classes.

---



Figure 25: Class Diagram for swapping data's in classes.

Assignment-6: Refer the below class diagram and Create a class that emulates a fractional number. Implement a friend function inc( ) that adds 1 to the object. Ensure to use add( ) function.

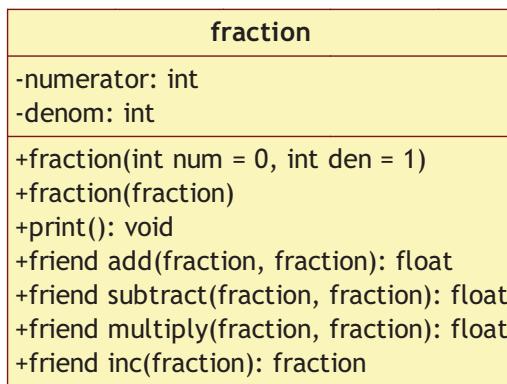


Figure 26: Class Diagram for fraction

---

## Lab 5. Operator Overloading

---

Goals	<ul style="list-style-type: none"><li>Understand and use Operator Overloading to write a program.</li></ul>
Time	90 minutes

**5.1 Overloading + Operator**

Refer the below class diagram and Write a program to overload the + operator to add two objects.

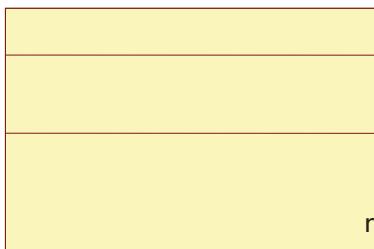


Figure 27: Class Diagram for + operator overloading two objects.

**Solution:**

**Step 1:** Create a class structure as per of class diagram with operator overloading function declaration and save it as Vector.h.

**Step 2:** Write the following code save it as Vector.cpp.

---

---

**Example 13: Vector.cpp****Step 3: Analyze the output of Vector.cpp.**

We have learnt that:

- 1) An overloaded operator is a member function of a class.
- 2) We can use operator overloading for the following:
  - a. To treat user defined data type as built in data type

In our example, vector is a class and + operator is overloaded to add two vectors. If we see only the main program statement `c = a + b;`, it appears as if two built in data types are added. However, actually a, b, and c are objects of the class vector.

<<TODO>>

**Assignment-1:** Refer the class diagram and Write a ComplexNumber class with overloaded arithmetic, relational, assignment, and I/O operators. The main function should read in two complex numbers, and use the overloaded operators to compute and display the results.

ComplexNumber
<pre>-real: float -imag: float</pre>
<pre>+ComplexNumber(float, float) +operator +(ComplexNumber): void +operator -(ComplexNumber): void +operator *(ComplexNumber): void +operator ==(ComplexNumber): void +operator !=(ComplexNumber): void +operator +=(ComplexNumber): void +operator -=(ComplexNumber): void +operator *=(ComplexNumber): void</pre>

**Figure 28: Class Diagram for ComplexNumber****Sample output:**

```
Enter a complex number C1: 4.2 + 8.3i
```

```
Enter a complex number C2: 3.1 - 9.2i
```

```
For C1 = 4.2 + 8.3i and C2 = 3.1 - 9.2i:
```

```
C1 + C2 = 7.3 - 0.9i
```

```
C1 - C2 = 1.1 + 17.5i
```

```
C1 * C2 = 89.38 - 12.91i
```

```
C1 == C2 is false
```

```
C1 != C2 is true
```

```
After C1 += C2, C1 = 7.3 - 0.9i
```

```
After C1 *= C2, C1 = 89.38 - 12.91i
```

**Formula used:**

Complex Addition:  $(a+bi)+(c+di)=(a+c)+i(b+d)$

Complex Subtraction:  $(a+bi)-(c+di)=(a-c)+i(b-d)$

Complex Multiplication:  $(a+bi)*(c+di)=(ac-bd)+i(ad+bc)$

Complex Division:  $(a+bi)/(c+di)=((ac+bd)+i(bc-ad))/(c^2+d^2)$

**Assignment-2:** Refer the below class diagram and Create a class CString. Use constructor to allocate free memory. The constructor should be overloaded for the following inputs.

- CString str; // Should initialize with single null character.
- CString str(10); // Should initialize with 10 spaces.
- CString str('\*', 10); // should initialize with 10 '\*' characters.
- CString str("Computer"); // should initialize with given string.
- Overload the following operators. Develop friend functions wherever applicable.

= += == <= [ ]

**For example:**

```
CString str1("Com"), str2("Prn");
str1 = "New String";
str1 += 's';
cout << str1[2]; // Should print the char 'w'
if (str1 == str2){...}
if (str1 <= str2){...}
```

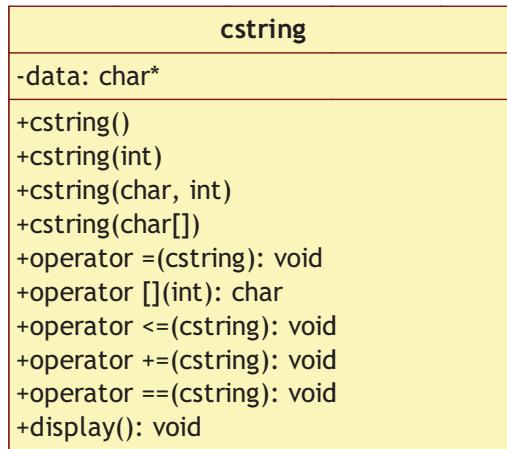


Figure 29: Class Diagram for CString

---

## Lab 6. Inheritance and Runtime Polymorphism

---

Goals	<ul style="list-style-type: none"><li>Understand and use inheritance to write a program.</li><li>Use run time Polymorphism to write a program.</li></ul>
Time	150 minutes

### 6.1 Inheritance

Refer the below class diagram and write a simple program to create two classes rectangle and triangle that derive from the polygon class. Both the classes should use width and height variables from polygon class for finding their area.

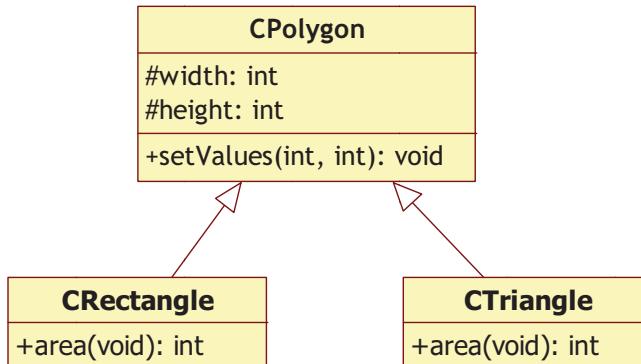


Figure 30: Lab 6-1 Class Diagram

**Solution:**

**Step 1:** Write the following code, and save it as Polygon.h.

**Step 2:** Write the following code, and save it as Lab6\_1.cpp.

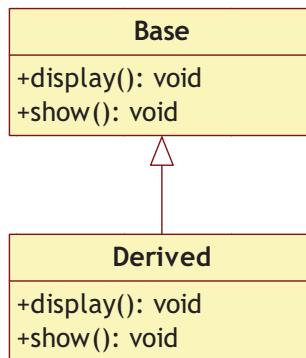
---

**Example 15: Polygon.cpp**

**Step 3:** Analyze the output of Polygon.cpp.

**6.2 Runtime Polymorphism**

Refer the below class diagram and write a program to create a derived class which inherits from base class. Derived class should override all the functions in base class without achieving runtime polymorphism.



**Figure 31: Lab 6-2 Class Diagram**

**Solution:**

Step 1: Write the following code, and save it as Lab6\_2.h.

**Example 16: Lab 6\_2.h**

**Step 2:** Write the following code, and save it as Lab6\_2.cpp.

**Example 17: Lab 6\_2.cpp**

**Step 3:** Analyze the output of Lab6\_1.cpp.

We have learnt that:

- 1) In the inheritance, we can have a pointer of base class type, which can point to a base class object as well as a derived type object. A derived class type pointer cannot point to object of base class type.
- 2) If we have an overridden function in base and derived class, then the base class pointer pointing to derived class object would always call base class version of the function. In the above mentioned example, the derived class object “D” is been pointed by a base class pointer “bptr”. The “bptr” would always call display() and show() functions from the Base class and never from the derived class.
- 3) Virtual keyword would solve this problem. Let us make change in the above program just by adding a keyword virtual before show() function of the Base class.

### 6.3 Program using virtual function

Refer the below class diagram for writing a program to create a base class with show method as virtual and derived class which inherits from base class. Derived class should override all the functions including virtual function in base class with achieving runtime polymorphism.

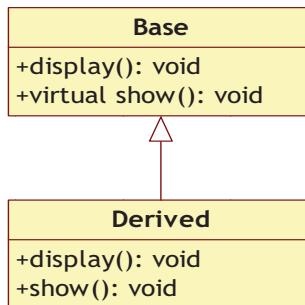


Figure 32: Lab 6-3 Class Diagram

#### Solution:

Step 1: Write the following code, and save it as Lab6\_3.h.

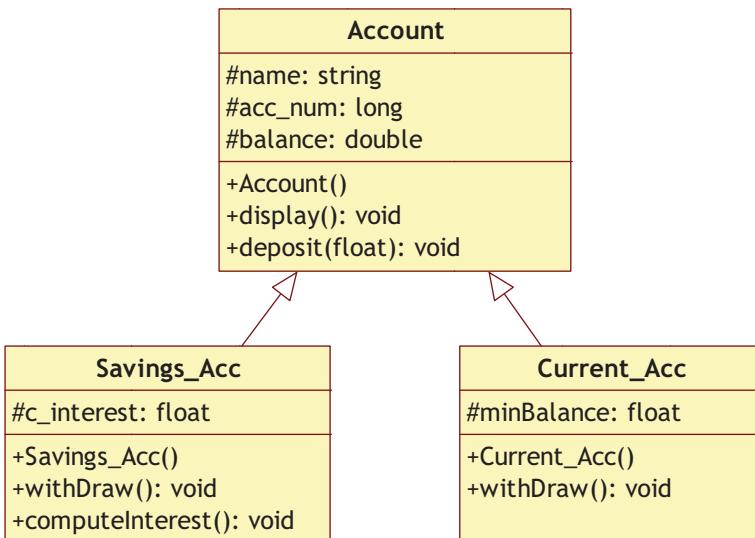
Step 2: Write the main code for Lab6\_3.cpp as similar to Lab6\_2.cpp and analyze the output of Lab6\_3.cpp.

&lt;&lt;TODO&gt;&gt;

**Assignment-1:** A bank maintains two kinds of accounts for customers, one called savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book but no interest. Current account holders should have a minimum balance else they should pay service charges.

Refer the below class diagram and write a program to create a class account that stores account number, customer name, and type of account. Create two classes cur\_acct and sav\_acct that derive from the account class to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks.

- 1) Accept deposit from customer and update the balance.
- 2) Display the account number, customer name, balance
- 3) Compute and deposit interest
- 4) Permit withdrawal and update balance
- 5) Use constructors it initializes the class members.



**Figure 33: Lab 6-Assignment 1 Class Diagram**

**Assignment-2:** Refer the below class diagram and write a program to create master class which inherits from account and admin class which in turn derive from the person class. In master class, use the attributes from base classes and write functions to perform operations like Create, update and display.

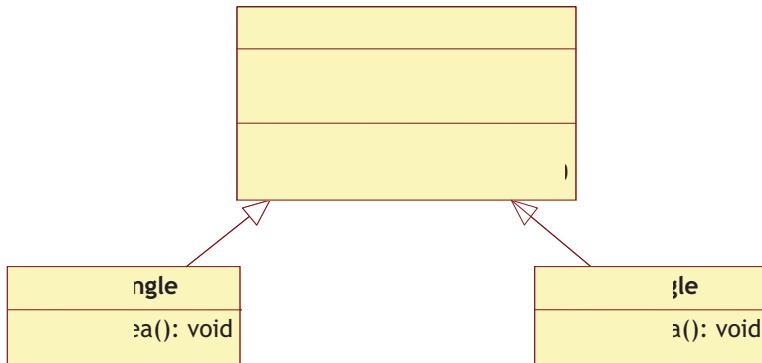
**Figure 34: Lab 6-Assignment 2 Class Diagram**

**Assignment-3:** Refer the below class diagram and write a simple program to create two classes rectangle and triangle that derive from the shape class. Both the classes should use x and y variables from shape class for finding their area by defining the virtual function which declares in shape class.

**Formula:**

Area of rectangle =  $x * y$ ;

Area of triangle =  $\frac{1}{2} * x * y$ ;

**Figure 35: Lab 6-Assignment 3 Class Diagram**

**Assignment-4:** Refer the below class diagram and Extend the above program to display the area of circles. This requires addition of a new derived class circle, which computes the area of circle. However, circle should accept only one value.

**Hint: Overload the get\_data( ) functions in base class.**

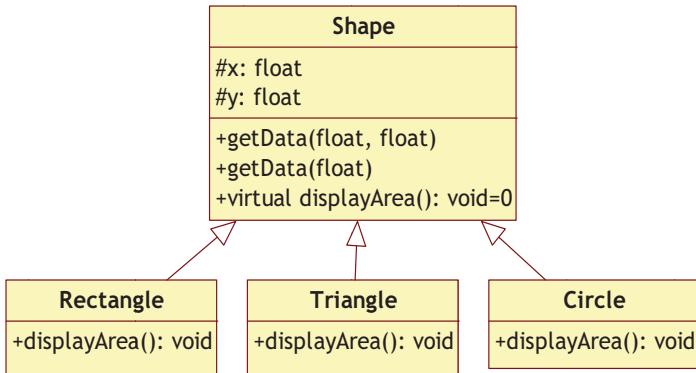


Figure 36: Lab 6-Assignment 4 Class Diagram

---

## Lab 7. String Class

---

<b>Goals</b>	<ul style="list-style-type: none"> <li>Understand and use string functions.</li> </ul>
<b>Time</b>	60 minutes

<<TODO>>

**Assignment-1:** Write a program to create a class named as string with a function as given below.

<b>String</b>
+replace_all(string&, const string&, const string&): void

Check whether passing different values to the function gives the corresponding output as given below.

- a) string text = "A man, a plan, a canal, Panama!";  
 replace\_all(text, "an", "XXX");

**produces the following o/p**

text = "A mXXX, a pIXXX, a cXXXal, PXXXama!";

- b) string text = "ananan";  
 replace\_all(text, "an", "XXX");

**produces the following o/p**

text == "XXXXXXXXXX";

- c) string text = "ananan";  
 replace\_all(text, "an", "anan");

**produces the following o/p**

text == "anananananan";

## Lab 8. Exception Handling

---

Goals	<ul style="list-style-type: none"><li>• Understand and use Exception Handling in writing a program.</li></ul>
Time	30 minutes

### 8.1 Exception Handling

Write a program to handle exceptional situation of accessing memory locations of an array beyond its limit.

#### Solution:

Step 1: Write the following code, and save it as Lab8\_1.cpp.

Step 2: Analyze the output of Lab8\_1.cpp.

<<TODO>>

**Assignment-1:** Write a global function that takes an integer as its argument and does a table lookup on an array of elements, which consists of numbers as string. For example: “One” for 1, “Five” for 5, and so on. If the number cannot be found, throw an exception.

---

## Lab 9. Templates

---

Goals	<ul style="list-style-type: none"><li>• Understand and use Templates.</li><li>• Understand and use Function Templates.</li><li>• Understand and use Class Templates.</li></ul>
Time	90 minutes

### 9.1 Function Template

Write a program to find minimum of two integers, character, and float numbers using function template.

**Solution:**

**Step 1:** Write the following code, and save it as `Function_template.cpp`.

**Example 20:** `Function_template.cpp`

**Step 2:** Analyze the output of `Function_template.cpp`.

### 9.2 Class Template

Write a program to add two float or integer members of a class using template class.

**Solution:**

---

**Step 1:** Write the following code save it as `class_template.cpp`.

<<TODO>>

**Assignment-1:** Write a function template called `swap()` that swaps the contents of its two input arguments.

**Assignment-2:** Write a template array class that emulates an array of whatever the parameterized type is. Test the template with int and float array.

- Include member function to add two arrays together
- Sort the array

## Lab 10. Input Output in C++

---

Goals	<ul style="list-style-type: none"><li>• Understand and use File handling</li></ul>
Time	90 minutes

### 10.1 File Handling

Write a program to read and write a file.

**Solution:**

**Step 1:** Write the following code save it as item.cpp.

#### Example 22: item.cpp

##### Step 2: Analyze the output of item.cpp.

We have learnt that:

- 1) If we want to use disk file, then we need to decide a suitable name for the file, purpose, and opening method for the file.
- 2) A file can be opened in two ways:
  - a. By using the constructor function of the class.

For example:

```
ofstream outfile(("STUDENTS") //output only or Istream
```

```
infile("STUDENTS") //for input only
```

- b. By using the member function open() of the class.

For example:

```
ofstream outfile;
```

```
outfile.open("STUDENTS");
```

```
outfile.close();
```

In both the examples, infile or outfile are called as stream objects. The function open() can be used to open multiple files that use the same stream object.

<<TODO>>

**Assignment-1:** Copy a text file into another by toggling the case of each letter from the source file. (All capital letters from source file should become small case in the target file and vice versa).

**Assignment-2:** Write a program to display the contents of a text file in reverse order.

## Appendices

---

### APPENDIX A: C++ Programming Standards

#### Standards:

The purpose of this document is to serve as a guiding standard for C++ programmers. Any client specific standards should override the standards given here in the event of contradiction between the two.

#### Naming Conventions:

The names of all variables, functions, and so on, should be informative and meaningful. It is worth the time to devise and use informative names because good mnemonics lead to clear documentation, easy verification, maintenance and improved readability. The following rules should be followed to achieve this target.

#### Filenames

- Use file name in the format : <basename>.<ext>
- Ensure that all the characters forming the basename and the extension must be in lowercase.
- Ensure that basename length is not longer than 10 characters always, out of which first 8 characters are unique and extension is not any longer than 3 characters .
- Always start the basename with an alphabet, and not with a digit or an underscore. Ensure that the basename does not end with an underscore.
- Avoid digits as far as possible. If used, ensure that the digits are at the end of the basename.
- Standardize the use of extensions as per the project requirements.
- Ensure that name of the file is always related to the purpose of that file.
- Ensure that the file name never conflict with any system file names.

#### Source Code File Names

- Ensure that files belonging to a module have a common prefix added to the basename, which indicates the module to which the file belongs.
- If a file contains only one function, then you can name the file with the same name as the name of the function.

#### Identifier Names

##### General:

- Ensure that the names are not longer than 31 characters.
  - Avoid similar looking names, for example: systst and sysstst
  - Ensure that names for similar but distinct entities are distinct.  
For example:  
'goat' and 'tiger' instead of 'animal\_1' and 'animal\_2'
-

- Consider limitations such as allowable length and special characters of compiler / linker on local machines.
- Ensure that names do not have an underscore at their beginning or end.
- Ensure that names are related to the purpose of the identifiers.
- Ensure that Underscore ' \_ ' is used to separate only significant words in names.
- Avoid names that differ only in presence or absence of underscore ' \_ '
- Avoid names that differ only in case, for example: foo, Foo.
- Avoid names conflicting with standard library names, variables and keywords.
- Standardize use of abbreviations.
- Avoid use of obscure abbreviations.
- Do not abbreviate InputCharacter as inch.
- Abbreviate common part of mnemonics

For example:

'next\_char' is preferred over 'n\_character'

#### **Local variables:**

- Ensure that local variables are in lower case only.
- Use standard short names when the scope of the variable is limited to a few lines (about 22 lines).

#### **Constants (#defines and enums):**

- Strictly avoid use of constants as they convey little meaning about their use. Instead use symbolic constants.
- There can be exceptions to this usage where 0 and 1 may appear as themselves.  
For example:  
`for (i = 0 ; i < ARREBOUND ; i++)`
- Ensure that all the symbolic constant names are in capitals only.  
For example:  
`#define ARREBOUND 25 /* comment */`
- Use the enumeration data type to declare variables that take discrete set of values.  
For example:  
`enum { BLACK = 0, WHITE = 1 } colors;`

#### **Tags:**

- Tags include typedefs, classes, structs, enums, and unions.
- Ensure that tags are in lower case only with \_t as a suffix.  
For example:



Example 23: Sample Code

---

**Class names:**

- Ensure that class names are in lower case only.

**Macros:**

- Append '\_M' to the macro names.
- Ensure that macro name is in all capitals.

For example:  
`#define MOD_M(a,b) \  
 <(b) ? ((b) - (a)) : ((a) - (b))`
- Ensure that macro has a pair of parenthesis if it calls a function inside it, even if the macro does not have parameters.

For example:  
`#define CALLFUNC_M( ) CallOtherFunction( )`

**Functions / Methods:**

- Relate the name of the function to the name of the module / class it belongs to. Convention for the abbreviation should be the same as that for the file names.

For example: function belonging to obj module will have Objxxxx.
- Use capitalization to separate significant words.

For example:  
'displayroutine' should be written as  
'DispRoutine'

**Pointers:**

- Append a '\_p' to pointer names.

For example: char \*name\_p ;

**Programming style:**

**General Rules:**

- Ensure that line length is less than or equal to 80 characters.
- Ensure that file length is less than or equal to 1000 lines.
- Ensure that each function within a file should be limited to 200 lines.
- Ensure that storage class, type, and variables are vertically aligned. Each item should start from the same column.
- Add the following rules for union/enum/struct/class:



**Example 24: Sample Code**

- Ensure that function prototypes and definitions are as per the following style:

**Example 25: Sample Code**

- Ensure that indentation is in multiples of 4 spaces. Do not use tabs for indentation.
- Ensure that any start and end brace are alone on separate lines, and are vertically aligned with the control keyword.

For example:

**Example 26: Sample Code**

- If a group of functions have alike parameters or local variables, then ensure that they have same names.

For example:

**Example 27: Sample Code**

- Avoid hiding identifiers across blocks.

For example:

**Example 28: Sample code**

- For pointer definitions, attach "\*" to variable names and not to the types.

For example:

**Example 29: Sample Code**

- Put unrelated variables on the separate lines.

For example:

**Example 30: Sample code**

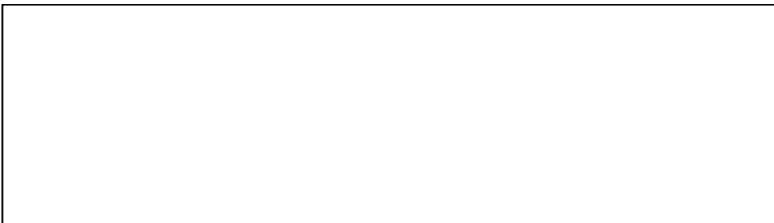
- Separate all the logical blocks by appropriate blank lines. Use three blank lines between the definitions of any two functions and at least one blank line between other major blocks.
- Demarcate static, automatic, and register variable declarations by blank lines.

- Declare the variables declared with the register storage class in decreasing order of importance. This is to ensure that the compiler allocates a register to the most important variables.
- Avoid simple blocks (that is blocks without any control statements).
- Do not depend upon any implicit types.  
For example:



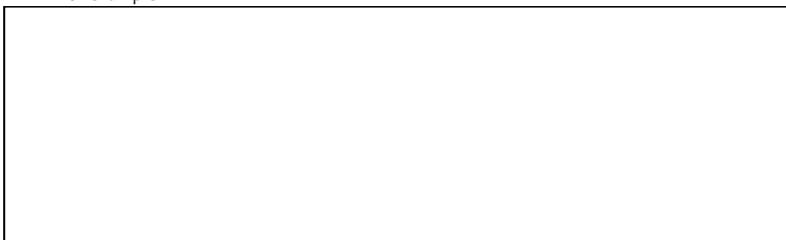
**Example 31: Sample Code**

- Ensure that Null body of the control statement is always on a separate line and explicitly commented as /\* Do nothing \*/.



**Example 32: Sample Code**

- Ensure that one line has at the most one statement except when the multiple statements are closely related.
- Use return value of functions.
- Write if statement with else-if clauses with the else conditions left justified.  
The format then looks like a generalized switch statement.  
For example:

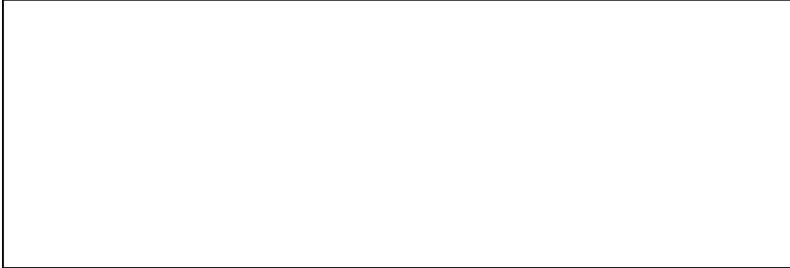


**Example 33: Sample Code**

---

- Ensure that the body of every control statement is a compound statement (enclosed in braces even if there is only one statement).
- Comment the Fall through in "case" statement.
- Ensure that "default" is present at the end of "case" statement.
- Do condition testing on logical expressions only.

For example: with 'count' having the declaration:



- Use "goto"s only under error/exception conditions and that too only in forward direction. However, it is to be generally avoided.
- Use labels only with "goto"s. Place labels at the first indentation.
- If some piece of code is deleted or commented while modifying the file, then delete or comment the identifiers, which become unused respectively.
- Check the side effects of ++ and --.
- Use "," as a statement separator only when necessary.
- Use ternary operator only in simple cases.
- Repeat array size in the array declarations if the array is defined with size.

For example:



---

**Example 35: Sample Code**

- Initialize all the global variable definitions.
- Do not assume the value of uninitialized variables.
- Declare the functions and variables used only in a particular file as “static” in that file.
- Prototype a function before it is used.
- Declare a Boolean type "bool" in a global include file.

For example:

**Example 36: Sample Code**

- Instead of checking equality with TRUE, check inequality with FALSE. This is because FALSE is always guaranteed to have a unique value, zero. However, this is not the case with TRUE.
- For example:

**Example 37: Sample Code**

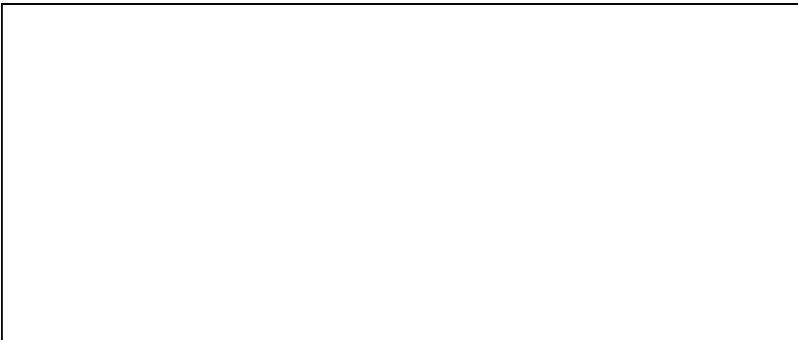
- Do not save tab characters in the file.
- Write the data declarations as vertically aligned with respect to both – the data types and identifier names.

For example:

**Example 38: Sample Code**

- Add space before and after all the binary operators except “.” and “->” for structure members.
- Do not separate Unary operators from their single operand.
- Note that horizontal and vertical spacing is equally important.
- Mandatorily use one space after each keyword.
- Break long statements on different lines, splitting out logically separate parts, and
- continue on the next line with an indentation of 4 spaces from the first line.

For example:



**Example 39: Sample Code**

#### **APPENDIX B: Debugging Examples**

##### **Example: Debugging in Visual Studio environment**

Suppose that following is the program that has to be debugged.

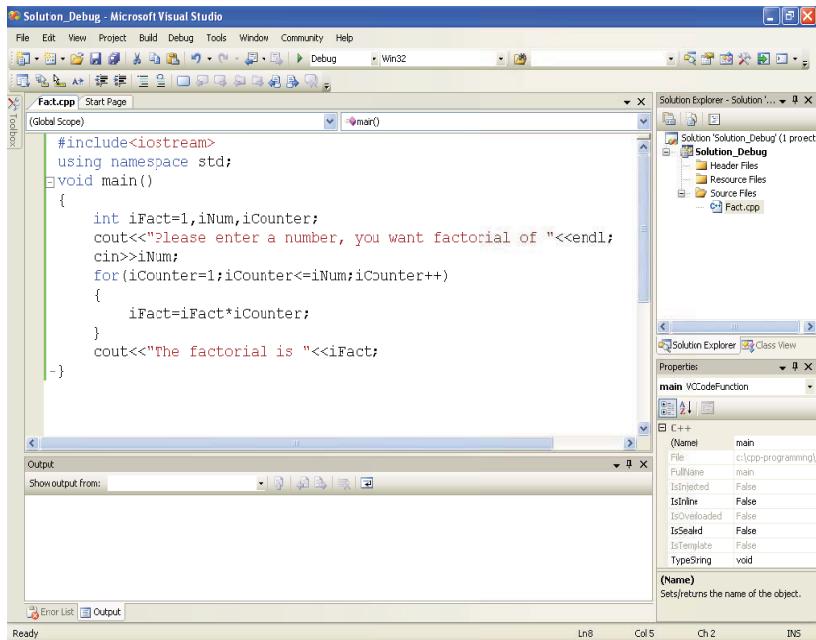


Figure 37: Program for Debugging

To use debug tool, we need to be sure that the build configurations for the given example is set to Win32 Debug, rather than Win32 release. The configuration is shown in the Configuration Manager Dialog Box. To display it, select Build -> Configuration Manager.

---

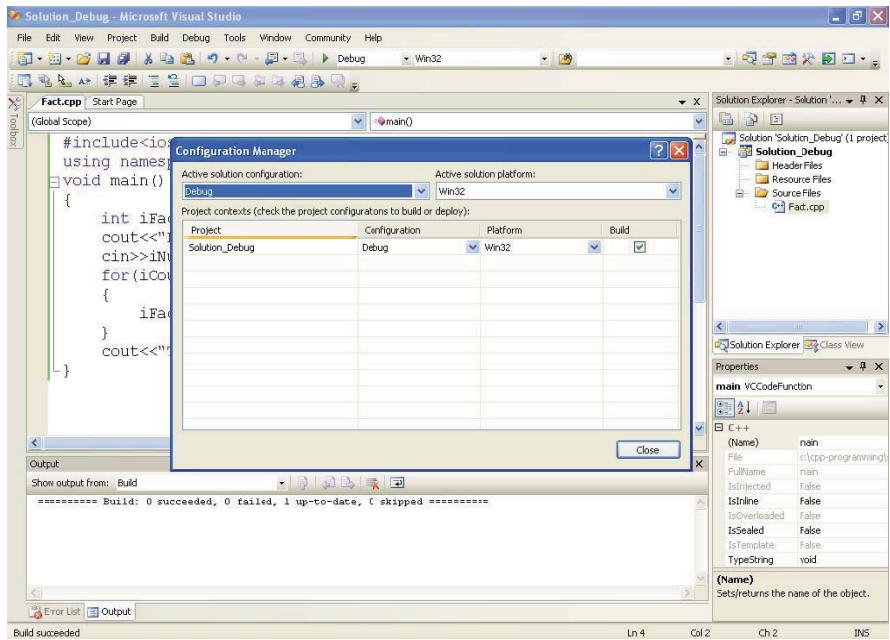


Figure 38: Checking the debugging Configuration

### Setting Breakpoints:

This enables us to define where in the program we want the execution to pause so that we can look at the variables in the program and change them if they don't have the values they should. For a large program we only want to look at a particular area of the program where there is a greater chance for error(s).

To set a breakpoint, simply place the cursor in the statement where we want execution to stop and right click select breakpoint -> Insert Breakpoint. To delete a breakpoint again place the cursor in the statement consists of breakpoint and right click select breakpoint -> Delete Breakpoint. Alternatively, we can remove all the Breakpoints in the active project by selecting the Debug Menu -> Delete All breakpoints.

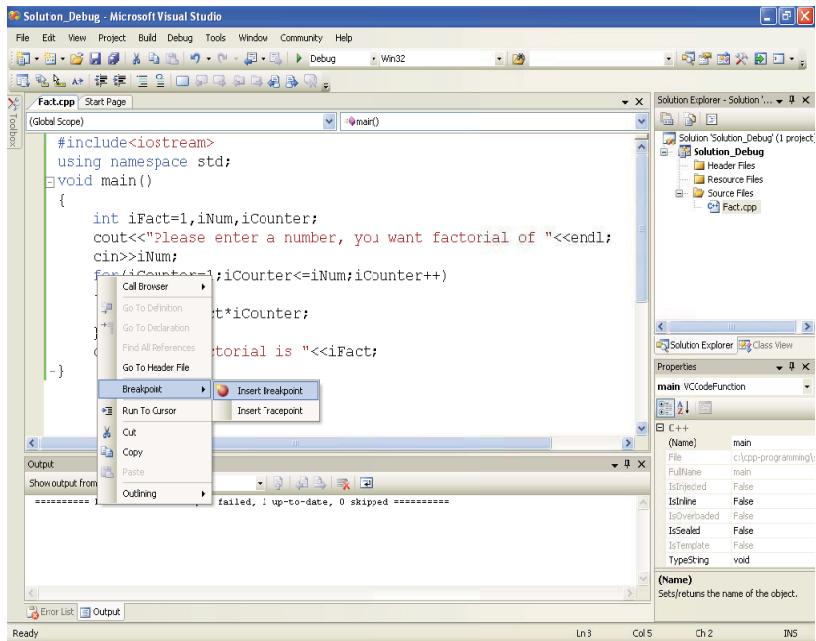


Figure 39: Setting Breakpoint

Observe the red round, indicating the breakpoint at some instruction.

The screenshot shows the Microsoft Visual Studio interface for a C++ project named "Solution\_Debug". The code editor displays a file named "Fact.cpp" containing the following code:

```
#include<iostream>
using namespace std;
void main()
{
    int iFact=1,iNum,iCounter;
    cout<<"Please enter a number, you want factorial of "<<endl;
    cin>>iNum;
    for(iCounter=1;iCounter<=iNum;iCounter++)
    {
        iFact=iFact*iCounter;
    }
    cout<<"The factorial is "<<iFact;
}
```

The "Output" window shows a successful build message:

```
===== Build: 0 succeeded, 0 failed, 1 up-to-date, 0 skipped =====
```

A red circular breakpoint icon is visible in the margin of the code editor at the line where the loop begins. The "Properties" window on the right shows the properties for the "main" function.

Figure 40: Break point is set

## Starting Debugging

In the development stage, you need to debug your code a number of times. A step-by-step execution of the code helps you monitor the execution path that gets followed. Monitoring/watching the values of the variables helps you catch the bugs (logical errors) in the code.

Options available in debugging:

- The Attach to Process option enables us to debug a program that is already running.
- The Step into option simply executes one statement at a time. This will step through all the statements in the #include files.
- The Step over facility can avoid having to step through all that code.

1. To start debugging the code, choose Debug -> Start Debugging or press F5.

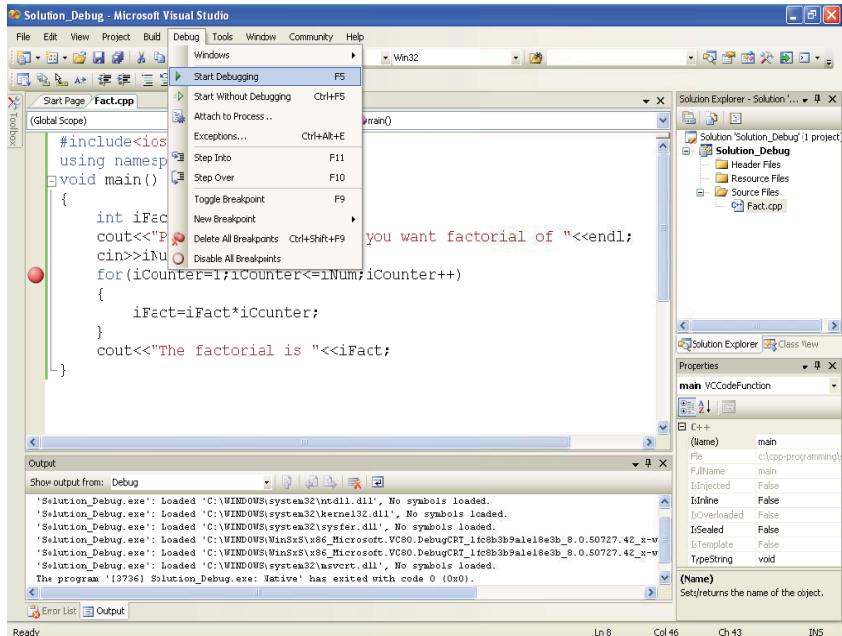


Figure 41: Starting Debugging

2. When debugging starts, the program prompts for input and stops at the breakpoint (indicated with yellow arrow as shown below). Further debugging of the loop is done, by selecting step into from Debug menu or you can continue the execution by pressing the key, F11 every time.

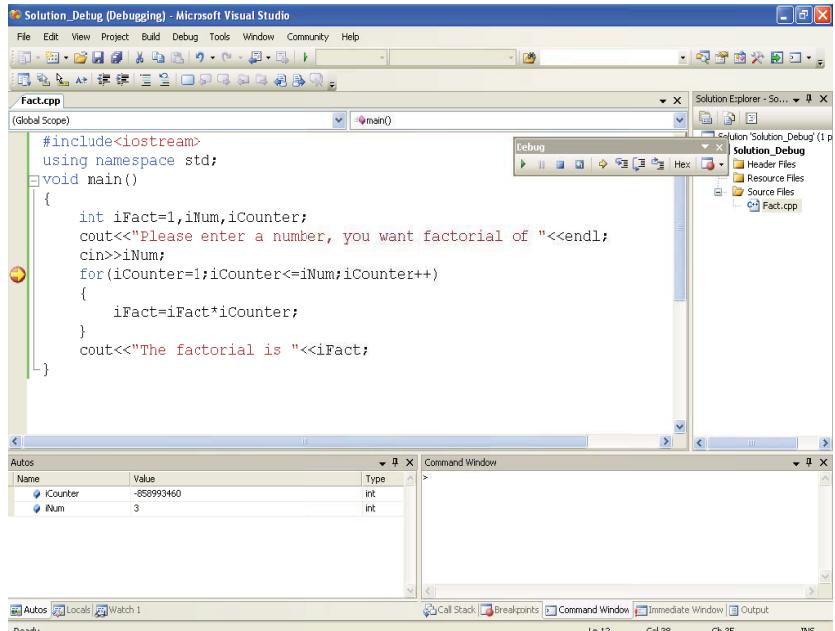
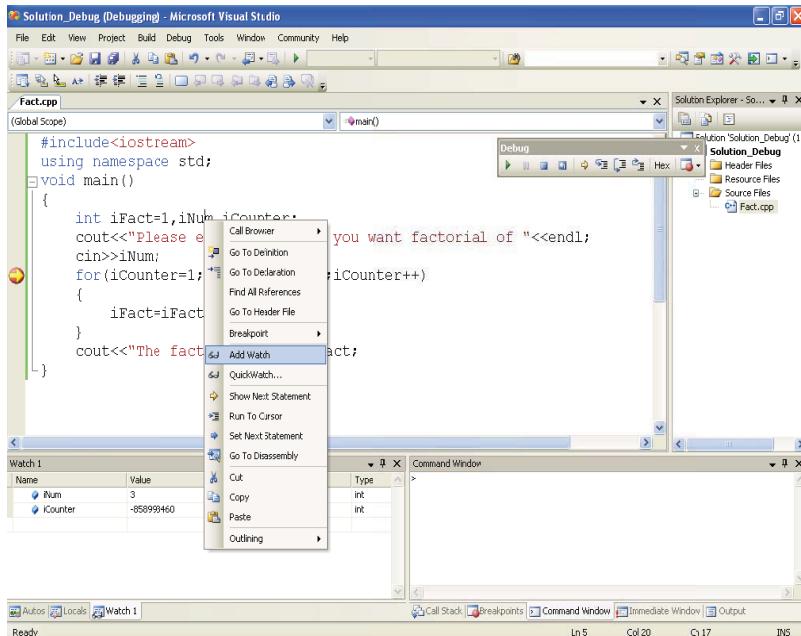


Figure 42: Debugging Started

3. To monitor/display the current value of a variable throughout the program execution, you can add a watch on that variable. To add a watch, right click on the variable in debug mode and select Add Watch from popup menu.



**Figure 43: Adding variable in the watch**

4. Observe iFact and iCounter variable, selected in the watch.

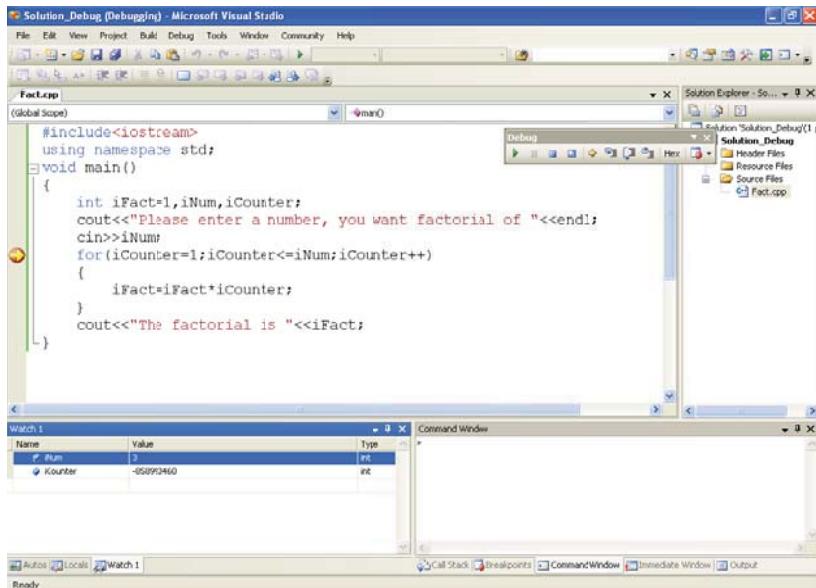


Figure 44: Variables in the watch Window.

5. Variable from the watch can be deleted by right clicking the variable in the watch window and selecting Delete Watch.

## Table of Figures

---

Figure-1: Visual Studio – start Page.....	5
Figure 2: Creating New Project in Visual Studio.....	6
Figure 3: Selecting the project type .....	7
Figure-4: Accepting Current Setting of the project .....	8
Figure 5: Selecting Application Type .....	9
Figure 6: Welcome Program .....	10
Figure 7: Output Screen .....	11
Figure 8: Output of simple.cpp file .....	12
Figure 9: Output of enumerated.cpp.....	14
Figure 10: Output of reference.cpp.....	16
Figure 11: dynamic_memory.cpp .....	18
Figure 12: Output of Function.cpp.....	20
Figure 13: Selecting C++ class file.....	22
Figure 14: Creating .h and .cpp file.....	23
Figure 15: Output Screen.....	24
Figure 16: Class Diagram for Student Class .....	25
Figure 17: Class Diagram for Time Class.....	27
Figure 18: Class Diagram for CRectangle Class.....	28
Figure 19: Class Diagram for CRectangle with friend function.....	29
Figure 20: Class Diagram for friend Class .....	30
Figure 21: Class Diagram for Employee.....	32
Figure 22: Class Diagram for complex .....	32
Figure 23: Class Diagram for point.....	33
Figure 24: Class Diagram for String .....	33
Figure 25: Class Diagram for swapping data's in classes.....	34
Figure 26: Class Diagram for fraction .....	34
Figure 27: Class Diagram for + operator overloading two objects.....	35
Figure 28: Class Diagram for ComplexNumber.....	37
Figure 29: Class Diagram for CString .....	38
Figure 30: Lab 6-1 Class Diagram.....	39
Figure 31: Lab 6-2 Class Diagram .....	41
Figure 32: Lab 6-3 Class Diagram .....	43
Figure 33: Lab 6-Assignment 1 Class Diagram .....	45
Figure 34: Lab 6-Assignment 2 Class Diagram.....	45
Figure 35: Lab 6-Assignment 3 Class Diagram.....	45
Figure 36: Lab 6-Assignment 4 Class Diagram .....	46
Figure 37: Program for Debugging .....	62
Figure 38: Checking the debugging Configuration.....	63
Figure 39: Setting Breakpoint .....	64
Figure 40: Break point is set .....	65
Figure 41: Starting Debugging .....	66
Figure 42: Debugging Started .....	67
Figure 43: Adding variable in the watch.....	68
Figure 44: Variables in the watch Window.....	69

---

## Table of Examples

---

Example 1: Simple.cpp .....	11
Example 2: enumerated.cpp.....	13
Example 3: reference.cpp .....	15
Example 4: dynamic_memory.cpp.....	17
Example 5: Function.cpp .....	20
Example 6: Sample Code .....	23
Example 7: MyClass.cpp.....	24
Example 8: Sample Code .....	26
Example 9: Construct.cpp.....	29
Example 10: Rectangle.cpp.....	30
Example 11: Lab4_3.h .....	31
Example 12: Lab4_3.cpp.....	31
Example 13: Vector.cpp .....	36
Example 14: Polygon.h .....	40
Example 15: Polygon.cpp .....	41
Example 16: Lab 6 _2.h.....	42
Example 17: Lab 6 _2.cpp .....	42
Example 18: Lab 6-3.h.....	43
Example 19: Lab 8 _1.cpp .....	48
Example 20: Function_template.cpp .....	49
Example 21: class_template.cpp.....	50
Example 22: item.cpp.....	52
Example 23: Sample Code .....	54
Example 24: Sample Code.....	55
Example 25: Sample Code .....	56
Example 26: Sample Code.....	56
Example 27: Sample Code .....	56
Example 28: Sample code .....	57
Example 29: Sample Code.....	57
Example 30: Sample code .....	57
Example 31: Sample Code .....	58
Example 32: Sample Code .....	58
Example 33: Sample Code .....	58
Example 34: Sample Code.....	59
Example 35: Sample Code.....	59
Example 36: Sample Code.....	60
Example 37: Sample Code .....	60
Example 38: Sample Code.....	60
Example 39: Sample Code.....	61

---