SURESH
GYAN VIHAR
UNIVERSITY
Accredited by NAAC with 'A+' Grade

Bachelor of Computer Application

(B.C.A.)

Object Oriented Programming with C++

Semester-iv

Author- Poonam Ponde

SURESH GYAN VIHAR UNIVERSITY

Centre for Distance and Online Education

Mahal, Jagatpura, Jaipur-302025

EDITORIAL BOARD (CDOE, SGVU)

Dr (Prof.) T.K. Jain
Director, CDOE, SGVU

Dr. Dev Brat Gupta
Associate Professor (SILS) & Academic Head, CDOE, SGVU

Ms. Hemlalata Dharendra
Assistant Professor, CDOE, SGVU

Ms. Kapila Bishnoi
Assistant Professor, CDOE, SGVU

Dr. Manish Dwivedi
Associate Professor & Dy, Director, CDOE, SGVU

Mr. Manvendra Narayan Mishra
Assistant Professor (Deptt. of Mathematics) SGVU

Ms. Shreya Mathur
Assistant Professor, CDOE, SGVU

Mr. Ashphaq Ahmad
Assistant Professor, CDOE, SGVU

Published by:

S. B. Prakashan Pvt. Ltd.

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com

© SGVU

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means (graphic, electronic or mechanical, including photocopying, recording, taping, or information retrieval system) or reproduced on any disc, tape, perforated media or other information storage device, etc., without the written permission of the publishers.

Every effort has been made to avoid errors or omissions in the publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice and it shall be taken care of in the next edition. It is notified that neither the publishers nor the author or seller will be responsible for any damage or loss of any kind, in any manner, therefrom.

For binding mistakes, misprints or for missing pages, etc., the publishers' liability is limited to replacement within one month of purchase by similar edition. All expenses in this connection are to be borne by the purchaser.

Designed & Graphic by : S. B. Prakashan Pvt. Ltd.

Printed at :

Syllabus

Object Oriented Programming

Learning Objective

- Provide flexible and powerful abstraction
- Allow programmers to think in terms of the structure of the problem rather than in terms of the structure of the computer.
- Decompose the problem into a set of objects
- Objects interact with each other to solve the problem
- create new type of objects to model elements from the problem space

Unit 1

Introduction to object oriented programming, user defined types, structures, unions, polymorphism, encapsulation. Getting started with C++ syntax, data-type, variables, strings, functions, default values in functions, recursion, namespaces, operators, flow control, arrays and pointers.

Unit II

Abstraction mechanism: Classes, private, public, constructors, destructors, member data, member functions, inline function, friend functions, static members, and references. Inheritance: Class hierarchy, derived classes, single inheritance, multiple, multilevel, hybrid inheritance, role of virtual base class, constructor and destructor execution, base initialization using derived class constructors.

Unit III

Polymorphism: Binding, Static binding, Dynamic binding, Static polymorphism: Function Overloading, Ambiguity in function overloading, Dynamic polymorphism: Base class pointer, object slicing, late binding, method overriding with virtual functions, pure virtual functions, abstract classes. Operator Overloading: This pointer, applications of this pointer, Operator function, member and non member operator function, operator overloading, I/O operators. Exception handling: Try, throw, and catch, exceptions and derived classes, function exception declaration, unexpected exceptions, exception when handling exceptions, resource capture and release.

Unit IV

Dynamic memory management, new and delete operators, object copying, copy constructor, assignment operator, virtual destructor. Template: template classes, template functions. Standard Template Library: Fundamental idea about string, iterators, hashes, iostreams and other types. Namespaces: user defined namespaces, namespaces provided by library. Object Oriented Design, design and programming, role of classes.

References

- Object Oriented Programming with C++ by E. Balagurusamy, McGraw-Hill Education (India)
- ANSI and Turbo C++ by Ashoke N. Kamthane, Pearson Education Reference Books: 1. Big C++ - Wiley India
- C++: The Complete Reference- Schildt, McGraw-Hill Education (India)
- C++ and Object Oriented Programming – Jana, PHI Learning.
- Object Oriented Programming with C++ - Rajiv Sahay, Oxford
- Mastering C++ - Venugopal, McGraw-Hill Education (India)

Contents

1.	**Principle of OOP's**	**14**
1.	Introduction	1-1
2.	What is Object Oriented Development?	1-1
3.	Object Oriented Methodology	1-2
4.	Overview of Procedure Oriented Programming	1-3
5.	What Is Object Oriented Programming?	1-4
6.	Object Oriented Languages	1-12
2.	**Basics of C++**	**14**
1.	A Brief History of C and C++	2-1
2.	Differences between C and C++	2-2
3.	Features of C++	2-3
4.	Advantages and Disadvantages of C++	2-3
5.	Applications of C++	2-4
6.	Writing and Executing a 'C++' Program	2-4
7.	Program Structure and Rules	2-6
8.	Sample C++ Program	2-7
9.	Comments	2-9
10.	Return Type of MAIN()	2-9
11.	Namespace std	2-10
12.	Header File	2-10
13.	Output Statement (COUT)	2-11
14.	Input Statement (CIN)	2-13
3.	**Expression**	**34**
1.	Introduction	3-1
2.	C++ Tokens	3-2
3.	Data Types	3-8
4.	Declaration of Variables	3-14
5.	Initialization of Variables	3-16
6.	Reference Variables	3-16
7.	Operators	3-18
8.	Type Cast Operator	3-24
9.	Memory Management Operators	3-25
10.	Expression	3-27
11.	Statement	3-28
12.	Symbolic Constant	3-29
13.	Type Compatibility	3-30
	Solved Programs	3-30
4.	**Functions in C++**	**22**
1.	Introduction	4-1
2.	Passing Information - Parameters	4-7
3.	Default Arguments	4-10
4.	Constant Arguments	4-11
5.	Function Overloading	4-11

6. Inline Functions ... 4-14
7. Recursive Functions ... 4-15
Solved Programs ... 4-16

5. Classes and Objects **36**

1. Introduction ... 5-1
2. Class ... 5-2
3. Member Functions ... 5-4
4. Making an Outside Function Inline ... 5-6
5. Nesting of Member Functions ... 5-7
6. Private Member Function ... 5-8
7. Arrays within a Class ... 5-8
8. Memory Allocation for Objects ... 5-9
9. Arrays of Objects ... 5-9
10. Objects as Function Arguments ... 5-11
11. Returning Objects ... 5-13
12. Const Member Function ... 5-14
13. Static Class Members ... 5-15
14. Pointer to Members ... 5-19
15. Local Classes ... 5-21
16. Friend Functions ... 5-22
17. Unions and Classes ... 5-25
18. Object Composition and Delegation ... 5-25
Solved Programs ... 5-27

6. Constructor and Destructor **20**

1. Introduction ... 6-1
2. Constructor ... 6-2
3. Multiple Constructors in a Class ... 6-11
4. Constructor with Default Arguments ... 6-12
5. Dynamic Initialization of Objects ... 6-12
6. Const Object ... 6-14
7. Destructor ... 6-15
Solved Programs ... 6-16

7. Operator Overloading and Type Conversion **26**

1. Introduction ... 7-1
2. Overloading Unary Operators ... 7-3
3. Overloading Binary Operators ... 7-6
4. Limitations of Operator Overloading ... 7-8
5. "this" Pointer ... 7-9
6. Overloading << and >> Operators ... 7-10
7. Manipulation of String ... 7-13
8. Type Conversion ... 7-17
Solved Programs ... 7-19

8. Inheritance **48**

1. Introduction ... 8-1
2. Single Inheritance ... 8-2
3. Multiple Inheritance ... 8-12
4. Multilevel Inheritance ... 8-18

5. Hierarchical Inheritance 8-19
6. Hybrid Inheritance 8-21
7. Container Classes 8-23
8. Virtual Base Classes 8-25
9. Constructors in Derived Classes 8-29
10. Destructors in Derived Classes 8-31
11. Nesting of Classes 8-33
12. Pointers to Derived Classes 8-39
13. Virtual Functions 8-41
14. Pure Virtual Functions 8-43
15. Abstract Classes 8-45
Solved Programs 8-47

9. The C++ I/O System Basics **20**

1. Introduction 9-1
2. C++ Stream 9-1
3. C++ Stream Classes 9-2
4. Unformatted I/O Operations 9-3
5. Formatted I/O Operations 9-7
6. Manipulators 9-11
Solved Programs 9-16

10. Working with Files **30**

1. Introduction 10-1
2. Creating a Stream 10-2
3. Opening a File 10-3
4. Closing a File 10-5
5. Checking for Failure with File Commands 10-5
6. Detecting the End-of-File 10-6
7. File Pointers and their Manipulation 10-8
8. Reading / Writing a Character from a File 10-9
9. write() and read() Functions 10-11
10. Buffers and Synchronization 10-12
11. Other Functions 10-12
12. Random Access File Processing 10-15
13. Updating a File: Random Access 10-16
14. Command Line Arguments 10-20
Solved Programs 10-22

11. Template **26**

1. Introduction 11-1
2. Generic Functions 11-2
3. A Function with Two Generic Data Types 11-4
4. Explicitly Overloading a Generic Function 11-5
5. Overloading Function Templates 11-7
6. Using Standard Parameters with Template Functions 11-8
7. Generic Functions Restrictions 11-9
8. Generic Classes 11-11
9. An Example with Two Generic Data Types 11-14

10. Using Non-type Arguments with Generic Class 11-15
11. Using Default Arguments with Template Classes 11-16
12. Template Parameters 11-18
13. Template Specializations 11-19
14. The Typename and Export Keywords 11-21
Solved Programs 11-22

12. Exception Handling **20**

1. Introduction 12-1
2. Exception Mechanism 12-2
3. Using Multiple Catch Statements 12-9
4. Catch-All Exception Handler 12-11
5. Nesting Try-catch Blocks 12-12
6. Exception Specifications 12-13
7. Unexpected Exception 12-14
8. Throwing an Exception from Handler 12-16
9. Uncaught Exception 12-17
Solved Programs 12-19

13. Introduction to Standard Template Library **32**

1. Introduction 13-1
2. The STL Programming Model 13-2
3. Containers 13-3
4. Algorithms 13-22
5. Iterators 13-25
6. Function Objects 13-26
7. Allocators 13-28
8. Adaptors 13-28
Solved Programs 13-30

14. Namespace **10**

1. Introduction 14-1
2. Defining a Namespace 14-2
3. The Standard Namespace 14-4
4. Nested Namespace 14-6
5. Unnamed Namespace 14-8
6. Namespace Alias 14-9

15. New Style Casts and RTTI **14**

1. Introduction 15-1
2. New-Style Casts 15-2
3. Static_cast 15-3
4. Dynamic_cast 15-4
5. Const_Cast 15-6
6. Reinterpret_cast 15-7
7. Run-Time Type Information (RTTI) 15-9
8. A Simple Application of Run-Time Type ID 15-11
9. Typeid can be applied to Template Classes 15-13

* * *

Principle Of OOP's

1. Introduction

The term "Object oriented" means organizing software as a collection of discrete objects that incorporate both data structure and behaviour. This is in contrast to conventional programming in which the data structure and behaviour are loosely connected. There is some dispute about exactly what characteristics are required by an object oriented approach, but they generally include four aspects: identity, classification, polymorphism and inheritance.

2. What is Object Oriented Development?

Object oriented development is fundamentally a new way of thinking and not a programming technique. Its greatest benefits come from helping specifiers, developers and customers, express abstract concepts clearly and let them communicate among each other. It can serve as a medium for specification, analysis, documentation and interfacing as well as for programming. Even as a programming tool, it can have various targets, including conventional programming languages and databases as well as object oriented languages. The essence of object oriented development is the identification and organization of application-domain concepts, rather than their final representation in a programming language, object oriented or not.

3. Object Oriented Methodology

We present a methodology for object oriented development and a graphical notation for representing object oriented concepts.

The methodology consists of building a model of an application domain and then adding implementation details to it during the design of a system. We call this approach as the Object Modeling Technique (OMT).

The methodology has the following stages:

i. **Analysis:** The purpose of the analysis is to state and understand the problem and the application domain so that a correct design can be constructed. A good analysis captures the essential features of the problem without introducing implementation artifacts that prematurely restrict design decisions.

   Analysis begins with a problem statement generated by clients and possibly the developers. The analyst builds a model of the real-world situation showing its important properties. The analyst must work with the requestor to understand the problem because the problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. The objects in the model should be application-domain concepts and not computer implementation concepts such as data structures. A good model can be understood and criticized by application experts who are not programmers. The analysis model should not contain any implementation decisions. Hence, analysis covers the detailed study of the requirements of both user and the software like what are the inputs to the system and what are the outputs expected?

ii. **System Design:** After analyzing a problem, you must decide how to approach the design. System design is the high-level strategy for solving the problem and building a solution. System design includes decisions about the organization of the system into subsystems, the allocation of subsystems to hardware and software components and major conceptual and policy decision that form the framework for detailed design.

   During design, decisions are made about how the problem will be solved, first at a high level, then at increasingly detailed levels.

iii. **Object Design:** The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations. The object design phase adds internal objects for implementation and optimizes data structures and algorithms.

   Object oriented design turns the s/w requirements into specification for objects and derived class hierarchies from which the objects can be created.

iv. **Implementation:** The object classes and relationships developed during object design are finally translated into a particular programming language, database or hardware implementation. Programming should be a relatively minor and mechanical part of the development cycle, because all of the hard decisions should be made during design. During implementation, it is important to follow good software engineering practice so that traceability to the design is straight forward and the implemented system remains flexible and extensible. *For example*: the window class would be coded in a programming language, using calls to the underlying graphics system on the workstation.

Object oriented concepts can be applied throughout the system development life cycle from analysis through design to implementation. The same classes can be carried from stage to stage without a change of notation, although they gain additional implementation details in the later stages. Although the analysis view and the implementation view of window arè both correct, they serve different purposes and represent a different level of abstraction.

Some classes are not part of analysis but are introduced as part of the design or implementation. *For example*: data structures such as trees, hash tables and linked lists are rarely present in the real world and they are introduced to support particular algorithms during design. Such data structure objects are used to implement real-world objects within a computer and do not derive their properties directly from the real world.

4. Overview of Procedure Oriented Programming

Conventional programming, using high-level languages such as COBOL, FORTRON and C is commonly known as Procedure Oriented Programming (POP). Using the procedure oriented approach, the programmer views a problem as a sequence of things to do. The programmer organizes the related data items and write the necessary functions (procedures) to manipulate the data and the process, complete the sequence of tasks that solve the problem. The primary focus is on the functions. A typical program structure for procedural programming is as shown in the following figure. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

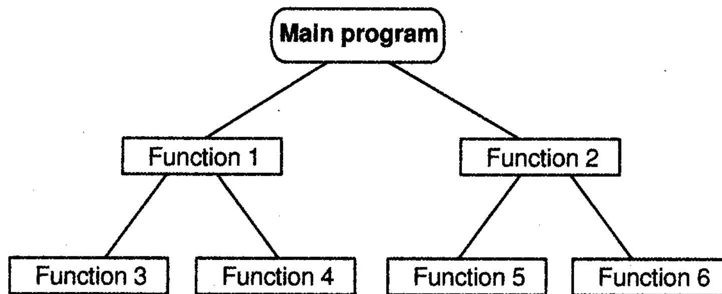


Figure 1.1: Typical structure of procedure oriented programs

For example: Consider a payroll system where employee pay-slip is to be generated. This can be shown in a diagrammatic form as follows:

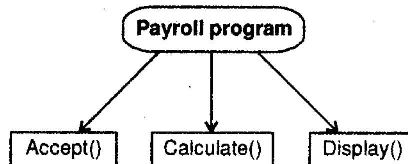


Figure 1.2

From the above diagram, Payroll Program is the main program and Accept(), Calculate(), Display() are the functions.

Accept() : This function can be used to accept the employee details.

Calculate():This function can be used for calculating the salary and the deductions.

Display() : This function can be used for displaying the employee pay-slip.

Procedure oriented programming basically consists of writing a list of instructions or actions for the computer to follow and organizing these instructions into groups known as functions. Normally a flow chart is used to organize these actions and represent the flow of control from one action to another.

In a multi-function program, many important data items are placed as global so that they can be accessed by all the functions. Each function may have its own local data.

4.1 Disadvantages of Procedure Oriented Programming

i. Global data is more important to an inadvertent change by a function. In a large program, it is very difficult to identify what data is used by which function. In such case we need to revise an external data structure, we also need to revise all functions that access the data.

ii. The procedure oriented programming approach does not model real world problems very well. This is because functions are action-oriented and do not really corresponds to the elements of the problem.

iii. The procedure oriented programming fails to eliminate pitfalls such as maintainability, reasonability, portability, security, integrity, etc.

4.2 Features of Procedure Oriented Programming

Features of Procedure Oriented Programming are as follows:

i. Focus is on the functions.

ii. It follows Top-Down approach (while programming).

iii. Program consists of different functions.

iv. Most of the functions share global data.

v. Functions transform data from one form to another.

v. Data is not hidden and can be easily shared.

5. What Is Object Oriented Programming?

Object Oriented Programming (OOP) contains the concepts of Procedure Oriented Programming and also some added additional concepts. In OOP the main focus is on the data that is to be used rather than the function. Once the data, which is to be used, is decided then the different functions that will operate on this data are defined. Thus it follows Bottom-Up approach. OOP allows decomposition of a problem into a number of entities called as objects and then builds data and functions around these objects. The organization of data and functions in object oriented programs is shown in the *figure 1.3*.

The data of an object can be accessed only by the functions associated with that object. However functions of one object can access the functions of other objects.

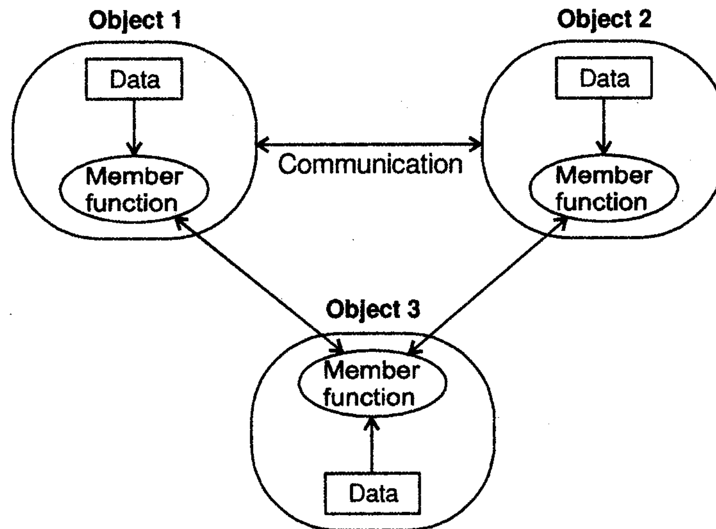


Figure 1.3: Structure of object oriented programming

Note

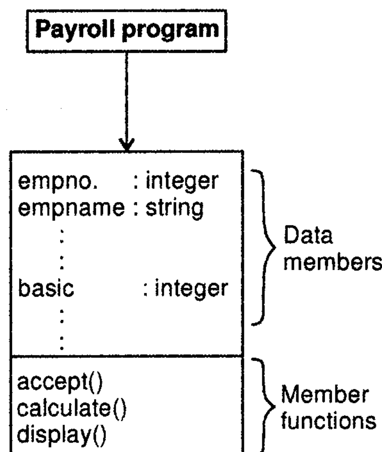
i. Since the data is hidden it cannot be accessed directly by any outside function.

ii. If you want to access or modify the data of an object then you should know the different functions, which are associated with those objects.

iii. Objects can pass messages to each other through functions.

For making this idea more clear let us consider the same example of payroll system (discussed in POP) where employee pay-slip is to be generated.

In POP focus is on the functions but in the OOP focus will be first on the data and then on the functions which will operate on this data. This can be shown as follows:



This program will consist of different objects and functions through which you can access the data.

5.1 Features of Object Oriented Programming

Features of Object Oriented Programming are as follows:

i. Focus is on data rather than procedures or functions.
ii. Program consists of different objects.
iii. Data structures are designed to characterize the objects.
iv. In data structure functions that operate on the data of an object are tied together.
v. Data is hidden and can only be accessed through the object's member functions.
vi. Objects can pass messages to each other through functions.
vii. New data and functions can be easily added whenever necessary.
viii. It follows bottom-up approach.

5.2 Basic Concepts of Object Oriented Programming

The following terms are required to be known when doing object oriented programming:

Terms required to know for Object Oriented Programming
i. Object
ii. Class
iii. Data encapsulation
iv. Data abstraction
v. Inheritance
vi. Polymorphism
vii. Message passing

i. Object:

An object is an identifiable entity with some characteristics and behaviour. Each object has a unique identity, some definitive state or characteristics and some behaviour. Objects are the basic run-time entities in an object oriented system.

In OOP, the programs consist of different objects. An *object* can be thought of as a real life entity, which can represent a person, thing, place, animal or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists.

When an object is mapped into software representations it consists of two parts:

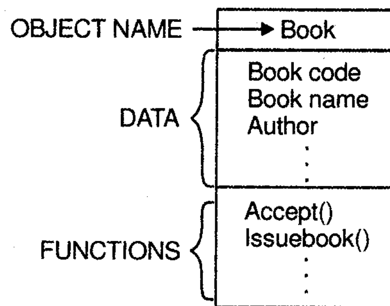
a. Data structure, referred to as *attributes*.
b. Processes that may change the data structure, called *functions* or *methods.*

When a program is executed, the objects interact by sending messages to one another. *For example*: If "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data and code to manipulate the data. Objects can interact without knowing the details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.

Different system can have different objects as discussed below:

a. Data structures like linked lists, stacks, queue, etc.
b. Employee in Payroll System.
c. Item in Inventory System.
d. Customer in Banking System.
e. Book in Library System etc.
f. GUI elements like Windows, Menus, Icons etc.
g. Various elements in computer games like Cannons, Guns, Animals, etc.
h. Computers in a Network Model.

For example: Consider a library system, which has different objects as 'book' and 'member'. Member object can pass message to book object to request for a book.



ii. Class

A class represents a set of objects that share common characteristics and behavior. Objects are variables of the type class.

Class is a user defined data types, which consists of data and member functions of an object. Classes are declared by using the keyword **class** followed by class name. Objects are instances of a class, i.e., multiple objects can be created for a class which are of same type. A class is thus a collection of objects of similar type. It is a blue-print for an object.

For example: Consider a class as snacks for which you want to declare different objects as wafers, chips, popcorns, etc. This can be declared as follows (in C++ language).

(Keyword) (Class Name)

```
(Keyword) (Class Name)
    ↓          ↓
  class      Snacks
  {
     . . .
  };
  Snacks Wafers, Chips, Popcorns;
         \_____________________/
                Objects
```

iii. Data Encapsulation

The binding of data and functions into a single unit is called as encapsulation. Data encapsulation is the most striking feature of a class. The data cannot be accessible to the outside world and only those functions which are present in the class can access it. These functions provide the interface between the object's data and program. This insulation of the data from direct access by the program is called as data hiding or information hiding. Therefore data hiding is a property where the internal data of an object is hidden from the rest of the program making it safe from accidental alteration. Data encapsulation and data hiding are the key terms in the description of OOP. To hide a data we have to put it in a class and make it private.

Syntax

```
class class_name
{
   private:
      variable declaration;
      function declaration;
   public:
      variable declaration;
      function declaration;
};
```

Data hiding is different than security techniques in following manner. Security technique is used to protect computer database. To provide a security measure user has to provide or give a password before accessing the database so data is secured from unauthorized users. But on the other hand data hiding is designed to protect well-intentioned programmers from mistakes. But still programmers can figure out a way to use private data but they will find it hard to do so.

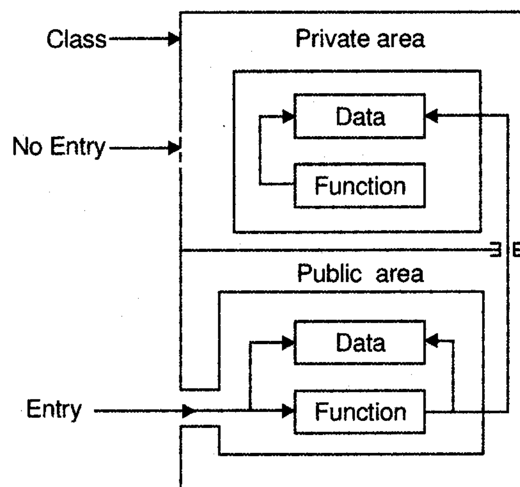


Figure 1.4

iv. Data Abstraction

Abstraction refers to the act of representing essential features without including the background details or explanations. The process of defining a data type, often called an Abstract Data Type (ADT), together with the principle of data hiding is called Data Abstraction. The definition of an ADT involves specifying the internal representation of the ADT's data as well as the functions to be used by others to manipulate the ADT. Data hiding ensures that the internal structure of the ADT can be altered without any fear of breaking the programs that call the functions provided for operations on the ADT. Classes use the concept of abstraction.

Hence, we can safely say that an abstraction is a named collection of attributes and behaviour relevant to modeling a given entity for some particular purpose. Abstraction is always relative to the purpose or user.

Abstraction and encapsulation are complementary concepts: *abstraction* focuses upon the observable behaviour of an object, while *encapsulation* focuses upon the implementation that gives rise to this behaviour.

v. Inheritance

Inheritance is the ability to derive a new class from an existing one. The child class can be a sub or super set of the parent. It supports the concept of hierarchical classification. From the figure given below Vehicle is the base, i.e., parent class, which has its own properties. Two classes namely Two-wheeler and Four-wheeler are derived from Vehicle class. Kinetic and Hero-Honda classes are derived from Two-wheeler class while Santro and Indica classes are derived from Four-wheeler class. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

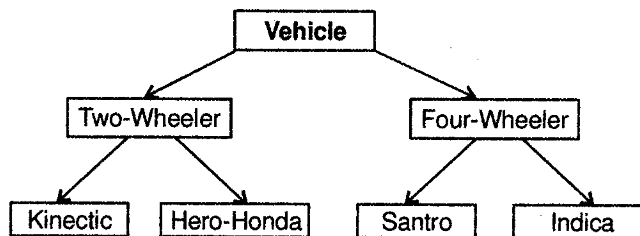


Figure 1.5

The concept of inheritance provides the idea of reusability, i.e., we can add additional features to the existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes, i.e., of base class and derived class.

Note that you can add new sub-classes, which will have their own properties or features in addition to the previous or base class features. Multiple Inheritance is the ability to derive a new class from more than one parent class.

vi. Polymorphism (Poly - many and morphism - Form)

Polymorphism is a Greek word. It is used to express the fact that the same message can be sent to many different objects and interpreted in different ways by each object. *For example*: Consider the operation of addition for two numbers, the operation will generate a sum. If the operands were strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading.*

Following figure illustrates that a single function name can be used to handle different number and different types of argument. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

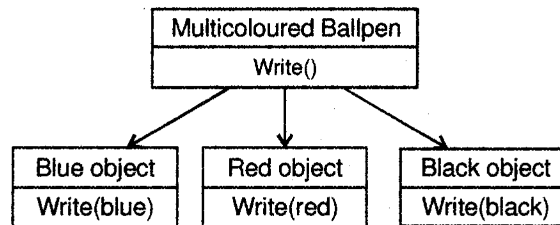


Figure 1.6

In the above *figure 1.6*, there is a multicoloured Ballpen which has different coloured refills and you can select the desired colour for writing. Depending on what argument is passed, the write function will get called. Thus you have same function name write() which behaves differently depending on the different types of arguments.

Polymorphism is extensively used in implementation of Inheritance and Virtual functions.

▶ Types of Polymorphism

Early binding: Choosing a function in normal way, i.e., during compilation time is called as early binding or static binding or static linkage. During compilation time, the C++ compiler determines which function is used based on the parameters passed to the function or the function's return type. The compiler then substitutes the correct function for each invocation. Such compiler based substitutions are called as static linkage.

By default, C++ follows early binding. With early binding, one can achieve greater efficiency. Function calls are faster in this case because all the information necessary to call the function are hard coded.

Late binding: Choosing functions during execution time is called as late binding or dynamic binding or dynamic linkage. Late binding requires some overhead but provides increased power and flexibility. The late binding is implemented through virtual functions. An object of a class must be declared either as a pointer to a class or a reference to a class.

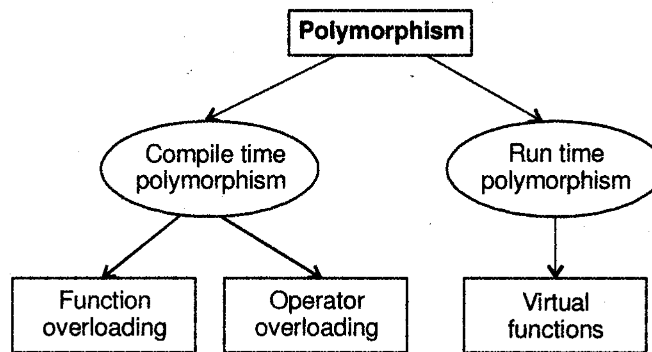


Figure 1.7

vii. Message passing

The act of communicating with an object to get something done is called as messaging or message passing. It consists of invoking a method, supplying that method with the information it requires, and receiving a reply from the method. Clearly, messaging can be something as simple as a function call.

An object oriented program consists of a set of objects that communicate with each other. The process includes the following steps:

a. Creating classes that define objects and their behaviour

b. Creating objects from class definitions and

c. Establishing communication among objects.

Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

For example: `shape.draw(circle);`

where, *shape* is the object name, *draw* is the message and *circle* is the information.

5.3 Applications of OOP

Applications of OOP are beginning to gain importance in many areas. The most popular application of object oriented programming, up to now has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP technique.

OOP has gained popularity in different areas such as object oriented databases where the concept are related to ORDBMS (Object Relational Database Management System), OODBMS (Object Oriented Database Management System), Expert system, Artificial Intelligence which helps in decision making MIS (Management Information System), DSS (Decision Support System).

Similarly in CAD/CAM system, real-time system and also in simulation and modeling where you can model the real world.

5.4 Advantages of OOP

OOP has provided different advantages, which are as follows:

i. Data hiding feature avoids misuse of your data.

ii. Large projects can be easily divided into different modules and can be integrated afterwards thus leading into less time and gaining maximum productivity.

iii. Re-usability of code is possible.

iv. Message passing technique for communication between objects makes the interface descriptions with external systems much simpler.

v. Object can be easily created whenever required.

vi. The real world can be easily modeled into different objects which in turn can communicate with each other, i.e., the real world can be simulated and modeled.

vii. Object oriented systems can be easily upgraded from small to large systems.

viii. Software complexity can be easily managed.

ix. It is possible to map objects in the problem domain to those in the program.

x. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

5.5 POP Vs. OOP

Following table summarizes the differences between the two most fundamental paradigms of programming.

Procedure oriented programming	Object oriented programming
Main focus is on the process or steps that must be taken to get the desired outcome (algorithm).	Main focus is on the data.
Program can be seen as a collection of procedures interacting with each other.	Program can be seen as a collection of objects.
No control on data access; data flows freely in the application.	Controlled data access; data is not directly accessible to the outside world.
Practices top down approach.	Practices bottom up approach.

6. Object Oriented Languages

The concept of OOPs can be implemented using languages such as C and Pascal, However, programming becomes difficult and may generate confusion when the programs grow large. A language which is specially designed to support the OOPs concepts makes it easier to implement such large programs.

The languages should support several of the OOP concepts to claim that they are object oriented. Depending upon the features they support, they can be classified into the following two categories.

6.1 Object based Programming Languages

In a technical sense, the term "Object-based Language" may be used to describe any programming language that is based on the idea of encapsulating data and code inside objects. Object based languages need not support inheritance and dynamic binding but those that do are also said to be object oriented. Object based languages that do not support inheritance or subtyping are usually not considered to be true object oriented languages.

Classification of Object Oriented Languages
i. Object based programming languages
ii. Object oriented programming languages

Languages that support programming with objects are said to be object-based programming languages. *For example*: Ada is a typical object-based programming language.

Another example of a language that is object-based but not object oriented is Visual Basic (VB). VB supports both objects and classes, but not inheritance, so it does not qualify as Object Oriented.

Major features that are required for Object-based Programming are as follows:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

6.2 Object Oriented Programming Languages (OOPLs)

Object oriented Programming Languages (OOPLs) are the natural choice for implementation of an object oriented design because they directly support the object notions of classes, inheritance, information hiding, and dynamic binding. Because they support these object notions, OOPLs make an object oriented design easier to implement. An object oriented system programmed with an OOPL results in less complexity in the system design and implementation, which can lead to an increase in *maintainability*. The genesis of this technology dates back to the early 1960s with the work of Nygaard and Dahl in the development of the first object oriented language called Simula 67. Research progressed through the 1970s with the development of Smalltalk at Xerox. Current OOPLs include C++, Objective C, Smalltalk, Eiffel, Common LISP Object System (CLOS), Object Pascal, Java, and Ada 95.

Object oriented (OO) applications can be written in either conventional languages or OOPLs, but they are much easier to write in languages especially designed for OO programming. OO language experts divide OOPLs into two categories, hybrid languages and pure OO languages. Hybrid languages are based on some non-OO model that has been enhanced with OO concepts. C++ (a superset of C), Ada 95, and CLOS (an object-enhanced version of LISP) are hybrid languages. Pure OO languages are based entirely on OO principles; Smalltalk, Eiffel, Java, and Simula are pure OO languages.

In terms of numbers of applications, the most popular OO language in use is C++. One advantage of C++ for commercial use is its syntactical familiarity to C, which many programmers already know and use; this lowers training costs. Additionally, C++ implements all the concepts of object orientation, which include classes, inheritance, information hiding, polymorphism, and dynamic binding. One disadvantage of C++ is that it lacks the level of polymorphism and dynamics most OO programmers expect. Ada 95 is a reliable, standardized language well suited for developing large, complex systems that are reliable.

The major alternative to C++ or Ada 95 is Smalltalk. Its advantages are its consistency and flexibility. Its disadvantages are its unfamiliarity (causing an added training cost for developers), and its inability to work with existing systems (a major benefit of C++).

EXERCISE

Review Questions

1. What is procedure oriented programming? State its features.
2. What is object oriented programming?
3. Distinguish between the following terms:
   i. Procedure Oriented Programming and Object Oriented Programming
   ii. Objects and Classes
   iii. Inheritance and polymorphism
   iv. Dynamic binding and message passing
4. Encapsulation is the mechanism by which data and functions are bound together within an object definition. Comment.
5. Explain the following terms:
   i. Object
   ii. Class
   iii. Message passing
6. Explain the concept of inheritance.
7. Give the advantages of OOP and application of OOP technology.
8. Which are the Object Oriented Languages?
9. Name the Object Based Languages.

2 Basics Of C++

I. A Brief History of C and C++

1.1 A Brief History of C

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C. Many of the important ideas of C evolve from the old language known as BCPL, developed by *Martin Richards*. The influence of BCPL on C proceeded indirectly through the language B, which was written by *Ken Thompson* in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. BCPL and B are "type less" languages whereas C provides a variety of data types.

In 1972 *Dennis Ritchie* at Bell Labs wrote C and in 1978 the publication of "The C Programming Language" by *Kernighan* and *Ritchie* caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

1.2 A Brief History of C++

The C++ programming language was designed and implemented by *Bjarne Stroustrup* at AT&T Bell Laboratories as a successor to C. It retains compatibility with existing C programs and the efficiency of C. It also adds many powerful new capabilities, making it suitable for a wide range of applications from device drivers to artificial intelligence. C++ will be of interest to UNIX users

because of its intimate relation to C and its potential use for building graphical user interfaces to UNIX for UNIX systems programming, and for supporting large-scale software development under UNIX.

C++ evolved from a dialect of C known as "C with Classes," which was invented in 1980 as a language for writing efficient event-driven simulations. Several key ideas were borrowed from the Simula67 and Algol68 programming languages. *For example:* It supports the features such as classes with inheritance and virtual functions, derived from the Simula67 language, and operator overloading, derived from Algol68.

C++ is best described as a superset of C, with full support for object-oriented programming. This language is in wide spread use. *Figure 2.1* shows the heritage of C++.

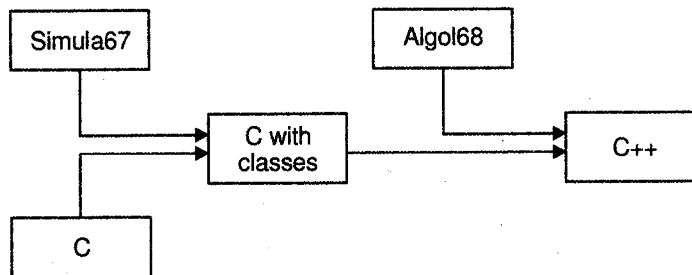


Figure 2.1: The heritage of C++

2. Differences between C and C++

i. C++ is a superset of C or we can also say that C is a subset of C++, C came first. So according to a general rule, a C++ compiler should be able to compile any C program. However, there are small differences between C programs and C++ programs-differences that reveal subtle incompatibilities between the two languages.

   For example: the C++ language adds keywords that are not reserved by the C language. Those keywords legitimately can be used in a C program as identifiers for functions and variables. Although C++ is said to include all of C, clearly no C++ compiler can compile such a C program.

ii. C programmers can omit function prototypes. C++ programmers cannot. A C function prototype with no parameters must include the keyword void in the parameter list. A C++ prototype can use an empty parameter list.

iii. Many standard functions have counterparts in C++ and C++ programmers view them as improvements to the C language. Following are some examples:

   - Malloc and free functions of Standard C are replaced by new and delete memory allocation operators in C++.
   - The character array processing functions declared in the Standard C library's <cstring> header file are replaced by the Standard C++ string class.

- The Standard C's stdio function library for console input and output is replaced by the C++ iostream class library.
- Standard C's setjmp() and longjmp() functions are replaced by C++'s try/catch/throw exception handling mechanism.

iv. Although the languages share common syntax they are very different in nature. C is a procedural language. When approaching a programming challenge the general method of solution is to break the task into successively smaller subtasks. This is known as top-down design. C++ is an object-oriented language. To solve a problem with C++ the first step is to design classes that are abstractions of physical objects. These classes contain both the state of the object, its data members, and the capabilities of the object, its methods. After the classes are designed, a program is written that uses these classes to solve the task at hand.

3. Features of C++

i. C++ supports all features of both structured programming and object oriented programming.
ii. It gives the easiest way to handle the data hiding and encapsulation with the help of powerful keywords like class, private, public and protected etc.
iii. Inheritance, one of the most powerful design concept is supported with single inheritance and multiple inheritance of base class and derived class.
iv. Polymorphism through virtual functions, virtual base classes and virtual destructors give the late binding of the compiler.
v. It provides overloading of operators and functions.
vi. C++ focuses on function and class templates for handling parameterized data types.
vii. Exception handling is done by the extra keywords, namely, try, catch and throw.
viii. Provides friends, static methods, constructors, and destructors for the class objects.

4. Advantages and Disadvantages of C++

The C++ language is a structured language which supports all necessary components of a structured language such as global and local variables, parameter passing by value and reference, and function and/or procedure return values. The C++ compiler also supports the concept of separate compilation of source code modules and the linking of independent object modules either from standalone files or from system or local libraries. This separate compilation feature allows the programmer to recompile only the parts of an application that have changed and relink those modified parts with existing modules to produce a new executable program.

The C++ compiler also allows an interface to assembly language. The compiler allows that block of code to be written in assembly and linked with blocks written in C++. A block of code could be a function or simply several lines of code within a larger C++ function. The programmer only has to use the keyword *asm* followed by an opening {, then the assembly code for the processor being used and then a terminating }. The assembly code can use any variable declared within the C++ code that follows within the scope of the assembly code.

The C language component of a C++ compiler has some weaknesses that make the language difficult to use as an application language. The language lacks strong type checking, meaning that the compiler will allow a character variable to be stored into a floating point variable without complaining. The C++ component of a C++ compiler has much stronger type checking and will issue warnings for such behavior. So for the programmer to receive the maximum protection against mixed type errors, it would be better to use the C++ component of the C and C++ compiler.

C++ does not have bounds checking on arrays. *For example:* If a programmer declares an array of 50 integers, but in the program code he or she by mistake stores a value into array element 52, the compiler will not complain. This lack of bounds checking can cause severe problems in some programs.

In the MS-DOS and PC-DOS operating system, C++ does not have any memory protection from access by pointers. A programmer can load a memory address of any place in memory into a pointer and through that pointer retrieve or set the value at that memory address. This feature can cause the DOS operating system to re-boot, hang-up, or crash completely. In addition, a programmer that is unfamiliar with the use of pointers, can, in MS/PC-DOS, cause his or her hard disk drive to crash, be reformatted, or destroyed. The same also holds true for the video display on a DOS machine. Fortunately, mini-computer operating systems, such as UNIX, have built-in memory protection that prevents such dangerous happenings as stated above.

Also, C++ does not have sophisticated string and record handling capabilities. Strings must be handled with a series of functions supplied in the standard C++ libraries. With C++, sophisticated String objects can be created as well as advanced record management schemes.

5. Applications of C++

C++ is suitable for handling any programming task such as development of editors, compilers, databases, communication system and any complex real-life application systems. It is a versatile language for handling very large programs.

i. C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.

ii. We can build special object-oriented libraries which can be used later by many programmers, since C++ allows us to create hierarchy-related objects.

iii. While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.

iv. It is expected that in the near future C++ will replace C as a general-purpose language.

6. Writing and Executing a 'C++' Program

There are the various stages involved in the process from creation to execution of a program. They are as follows:

i. **Creating the Source Code:** A C++ program is written as ordinary text (called source). A small C++ program may only contain one file (called Source file), but a larger program often contains several. The file containing the source code has to be a 'text' file with an

extension to indicate that it is a C++ program file. C++ implementations use extensions such as .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT & T version uses .cc.

ii. **Compiling the Source Code:** The source file you have created is not a program file so to turn your source code into a program, a compiler is used. The compiler takes .cpp or .cc files, preprocess them (removing comments, add header files) and translates them into *object files* having extension .o or .obj.

For compiling the source code from the operating system's command line, you should type the following statement. *For example:* incase of Borland Turbo C++ compiler use statement *tc <filename>*, incase of Borland C++ compiler use the statement *bc <filename>*.

iii. **Linking the Object Code to create an Executable Code:** After successful compilation the set of object files is processed by a *linker*. This program combines the files, adds necessary libraries and creates an executable file with extension .exe. These steps are illustrated in the *figure 2.2*.

Note: A library is a collection of linkable files that you have created, were supplied with your compiler, or that you have purchased separately. All compilers come with a library of useful functions (or procedures) and classes that you can include in your program. A function is a block of code that performs a service, such as adding two numbers or printing to the screen. A class is a collection of data and related functions.

iv. **Executing the Program**: Once the executable file is created, you can run it by typing its name at the DOS command prompt or through the option provided by the compiler software. If the desired results are not achieved, changes may have to be made to the source code. When the source code is changed, it has to be recompiled and linked to create the correct executable code.

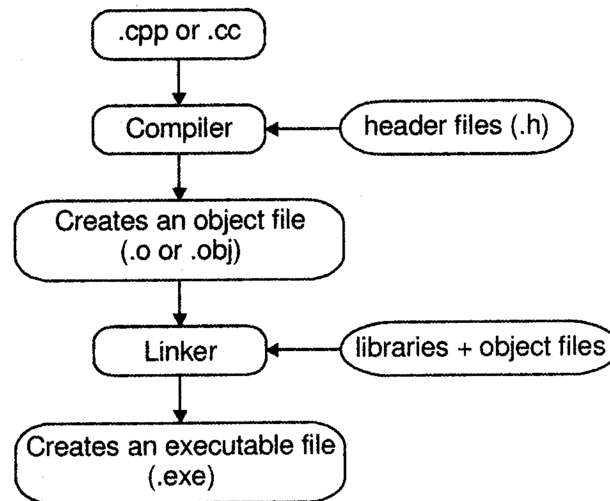


Figure 2.2: Compilation steps

7. Program Structure and Rules

The majority of C++ programs are composed of functions. All functions have the following general form:

```
function-return-type function-name(argument list)
{
   local variable declarations;
   body of the function;
}
```

A general form for a C++ program is as follows:

```
inclusion of header files

defined constants

declarations of global variables
---------------------------------------------------------
function-return-type main(arguments from command line)
{
   declaration of local variables used by main function;

   body of the main function;
}
---------------------------------------------------------
function-return-type next_function(argument list)
{
   declaration of local variables for next_function;
   body of the next_function;
}
```

A C++ program must have one and only one function with the name **main**. **main** must be in lower case. The **main** function should have a return type of either **int** or **void**, with **int** being the preferred return type. Since **main** is the entry point to the program from a calling process, which could be the operating system or another program, the return value of main is used to indicate the completion status of the called program. A program that returns a value 0 is considered to have completed successfully and a return value other than 0 is considered to indicate an abnormal termination.

All functions in the program, besides **main**, can have names the programmer desires. The names of functions should indicate what task or purpose that function is to perform. All C++ keywords must be in lower case. Function names can be composed of letters, digits and underscores and can be a maximum of 30 characters, but must be unique in the first 8 characters.

Functions must have a pair of curly braces, **{ }**, these signify the beginning and end of a function's scope. The names of variables that are declared within a function can be the same as function names, but this is not a good practice. Variables declared within the scope of a function are called local variables. Local variables "live" only while the function has control. When the function terminates, and control returns to the calling function, all local variables cease to exist. The statements within a function are terminated with a semi-colon, this is called the statement terminator. Nested functions or subroutines are not allowed. Nesting a function means that one function is defined within another.

The difference between a declaration and definition is important. A **declaration** announces the properties of a variable or a function. The main reason for declaring variables and functions is for type checking. If you declare a variable or function and then later make reference to it with data objects that do not match the types in the declaration, the compiler will complain. The purpose of the complaint is to catch type errors at compile time rather than waiting until the program is run, when the results can be more disastrous.

A **definition**, on the other hand, actually sets aside storage space (in the case of a variable) or indicates the sequence of statements to be carried out (in the case of a function). A function prototype, or declaration, is a return type followed by a function declarator followed by a semicolon. A function definition is a return type followed by a function declarator followed by a function body enclosed in matching braces, **{ }**. A function definition can also serve as a declaration for all source code following the definition. The only time a function declaration is actually required is when a reference is made to a function before the function definition is specified.

A variable declaration with an **extern** specifier is not a definition, unless it has an initializer. The function prototype is a function declaration, but it is not a definition.

When first learning C++ programming, always start C++ programs with

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

These header files include the function prototypes that allow a program to call the standard input/output functions that make it possible to write to the screen and read from the keyboard.

8. Sample C++ Program

```
// This is my first C++ program
#include<iostream>
using namespace std;
int main()
{
    cout<<"Welcome to C++"<<endl;
    cin.get();
    return 0;
 }
```

If you compile and run this code, you will see the message "Welcome to C++" as output.

Explanation

Let's take a look at the above program line by line.

i. **// This is my first program:** This is a comment line. All the lines beginning with two back slash signs (//) are considered comments and do not have any effect on the behavior of the program. They can be used by the programmer to include short explanations or observations within the source itself. In this case, the line is a brief description of what our program does.

ii. **#include<iostream>:** Sentences that begin with a hash sign (#) are directives for the preprocessor. They are not executable code lines but indications for the compiler. In this case the sentence #include<iostream> tells the compiler's preprocessor to include the **iostream** standard header file. This specific file includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is used later in the program.

iii. **Using namespace std; :** This line, *namespace* defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the *using* directive, like **using namespace std;** here, *std* is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. *using* and *namespace* are the new keywords of C++.

iv. **int main() :** This line corresponds to the beginning of the **main** function declaration. The **main** function is the point from where all C++ programs begin their execution. It is independent of whether it is at the beginning, at the end or in the middle of the code - its content is always the first to be executed when a program starts. For the same reason, it is essential that all C++ programs have a **main** function.

main is followed by a pair of parenthesis () because it is a function. In C++ all functions are followed by a pair of parenthesis () that, optionally, can include arguments within them. The content of the **main** function immediately follows its formal declaration and it is enclosed between curly brackets (**{}**), as in our example.

v. **{**, is the opening brace, marks the start of a code block. In this case it marks the start of function main(). For every opening brace in a program there will be a corresponding closing brace marking the end of the code block.

vi. **cout <<"Welcome to C++" << endl; :** This instruction does the most important thing in this program. **cout** is the standard output stream in C++ (usually the screen), and the full sentence inserts a sequence of characters (in this case "Welcome to C++") into this output stream (the screen). **cout** is declared in the iostream header file, so in order to be able to use it that file must be included.

The endl manipulator outputs a newline and then flushes the stream. Because cout is buffered, output may not be displayed when first written. Flushing the stream forces the contents of the buffer to be output.

Notice that the sentence ends with a semicolon character (**;**). This character signifies the end of the instruction and must be included after every instruction in any C++ program.

vii. **cin.get(); :** This line causes the program to wait for input from the cin input stream. This may not be required, however, there are some programming environments, for instance, running a console program in a Microsoft Windows environment, where the output may disappear before you have an opportunity to see it when the program completes. Adding this line of code will enable you to see the output, and then press Enter to continue. If not required for your environment you can safely remove the line from your program.

viii. **return 0; :** The **return** instruction causes the **main()** function finish and return the code that the instruction is followed by, in this case **0**. Thus it is most usual way to terminate a program that has not found any errors during its execution. As you will see in coming examples, all C++ programs end with a sentence similar to this.

ix. } is the closing brace, marks the end of function main().

9. Comments

Comments are pieces of source code discarded or ignored from the code by the compiler. They do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. Comments can be written anywhere in the program and are used for documentation.

C++ supports two ways to insert comments:

i. `// line comment`

The line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line.

For example: `//This is a line comment`

Everything after the // (double slash) to the end of the line is a comment and ignored by the compiler.

Basically the line comment (double slash comment) is a single line comment.

ii. `/* block comment */`

The second one, the block comment, discards everything between the **/*** characters and the next appearance of the ***/** characters, with the possibility of including several lines.

For example:

```
/* This is our first program of
   C++ to illustrate some of its features*/
```

Everything between /* and */ are comment and ignored by the compiler.

For multi-line comments the block comment (traditional C comment) is more suitable.

If you include comments within the source code of your programs without using the comment character combinations //, /* or */, the compiler will take them as if they were C++ instructions and, most likely cause one or several error messages.

10. Return Type of MAIN()

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a *return(0)* statement; otherwise a warning or an error might occur. Since main() returns an integer type value, return type for main() is explicitly specified as int. The following definition of main() is incorrect and shouldn't be used.

```
void main()
```

The default return type for all functions in C++ is int. The following main without type and return will run with a warning:

```
main()
{
   . . . .
   . . . .
}
```

11. Namespace std

When we include headers from the C++ standard library the contents are in **namespace std**. A namespace is simply a declarative region. Its purpose is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another. For more details refer chapter 14. There are three ways to qualify their use so that we can use them in our program:

i. **A using directive:** `using namespace std;`

This method is simple but it has the disadvantage that it pollutes the global namespace because it makes all of the identifiers from namespace std available globally.

ii. **A using declaration:** `using std::cout;`

This is clearly more limited and has to be repeated for everything we wish to qualify, so if we apply this to our first example instead of a using directive the program would be written:

```
#include<iostream>
using std::cout;
using std::endl;
using std::cin;
int main()
{
  cout <<"Welcome to C++"<< endl;
  cin.get();
  return 0;
 }
```

iii. **Explicit identifier:** `std::cin.get();`

Here an explicit identifier is used as a prefix for each use. Our first program would then be written:

```
#include<iostream>
int main()
{   std::cout << "Welcome to C++" << std::endl;
    std::cin.get();
    return 0;
 }
```

12. Header File

Header files are source code files that contain defined constants, constant values, macros, data types, templates and function prototypes. C and C++ need to define function prototypes for calls to functions residing outside of the current file. All input/output functions, memory management functions, string manipulation functions, math functions and assorted other functions reside in

external libraries and are not part of the syntax of the language. In order for a C and C++ program to make use of these capabilities, the program must first include the proper header file that prototypes the functions desire.

The C++ library includes headers from the C standard library. These have names such as: <stdlib.h>, <string.h>, <time.h>

You can use these headers in your C++ programs but its contents are not in namespace std, they are in the global namespace. Use of these older C headers is deprecated in C++; you can use them but they may disappear from the standard at some time in the future.

C++ provides its own versions of these headers whose contents are in namespace std. The equivalent headers for the above are: <cstdlib>, <cstring>, <ctime>

In the new version of C++, instead of the suffix '.h' they have a 'c' prefix.

C++ also introduces a number of headers that didn't exist in C, such as: <iostream>, <vector>, <algorithm> etc.

There are also some C++ headers, which don't use a '.h' extension, but they don't have a prefix either.

Since the new-style header is a relatively recent addition to C++, you will still find many programs using older versions of the C++ headers, pre-dating the standard. These will have names such as:

<iostream.h>, <fstream.h> instead of: <iostream>, <fstream>

Incase of strings, there are three string headers to consider, the first two:

<string.h>, <cstring>

are for C-style strings (null-terminated character arrays). These contain the string functions such as strcpy(), strcat() and strlen(). The former is the deprecated C library header, the latter is the more recent C++ header.

The third string header is: <string> and this is used for std::string class in C++.

Note that these older versions are not addressed by the C++ standard at all. They're not deprecated, they're not C headers and they're not part of the C++ standard. They are simply old headers that pre-date the standard and a great deal of code has been written using them over the years. Generally you should prefer the use of the C++ headers from the standard library rather than these older versions unless you're maintaining old code, but compilers are likely to continue to provide both versions for some time to come for backward compatibility.

13. Output Statement (COUT)

The **cout** stream is used in conjunction with the overloaded operator << (a pair of *"less than"* signs).

Syntax

```
cout << variable1 << variable2 <<. . . << variablen;
```

The <<operator is known as *insertion operator* since it inserts the data that follows it into the stream that precedes it.

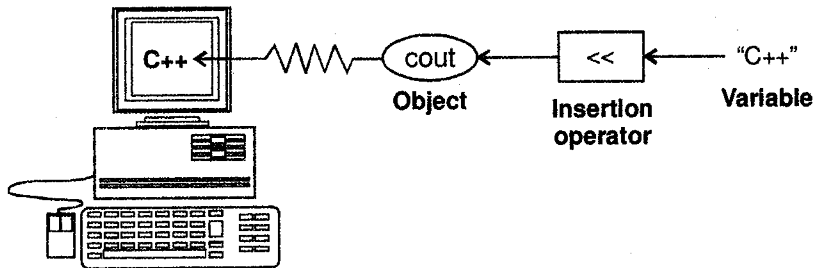


Figure 2.3: Output using insertion operator

For examples

```
1.  cout << "Output sentence"; //print Output sentence on screen
2.  cout << 120;               // print number 120 on screen
3.  cout << x;         // print the content of variable x on screen
```

In the examples above it inserted the constant string "Output sentence," the numerical constant 120 and the variable x into the output stream **cout**.

In the first example output sentence is enclosed between double quotes (" ") because it is a string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes so that they can be clearly distinguished from variables.

Consider the following two sentences, they are very different:

```
cout << "Hello";
cout << Hello;
```

The first sentence prints Hello on screen and another sentence prints the content of Hello variable on screen.

The *insertion* operator (<<) may be used more than once in a same sentence:

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

This sentence would print the message **Hello, I am a C++ sentence** on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is "  <<
    zipcode;
```

If we suppose that variable age contains the number 24 and the variable zipcode contains 90064 the output of the previous sentence would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that **cout** does not add a line break after its output unless we explicitly indicate it, therefore, the following sentences:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown as following on screen:

```
This is a sentence. This is another sentence.
```

Even, if we have written them in two different calls to **cout**. In order to perform a line break on output we must explicitly order it by inserting a new-line character that in C++ can be written as **\n**:

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Additionally, to add a new-line, you may also use the **endl** manipulator. *For example*

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

```
First sentence.
Second sentence.
```

The **endl** manipulator has a special behavior when it is used with buffered streams: they are flushed. But anyway **cout** is unbuffered by default.

You may use either the **\n** escape character or the **endl** manipulator in order to specify a line jump to **cout**.

14. Input Statement (CIN)

Handling the standard input in C++ is done by applying the overloaded operator >> (a pair of greater than sign) known as extraction operator on the **cin** stream.

This must be followed by the variable that will store the data that is going to be read.

Syntax

```
cin >> variable1 >> variable2 >> . . . >> variablen;
```

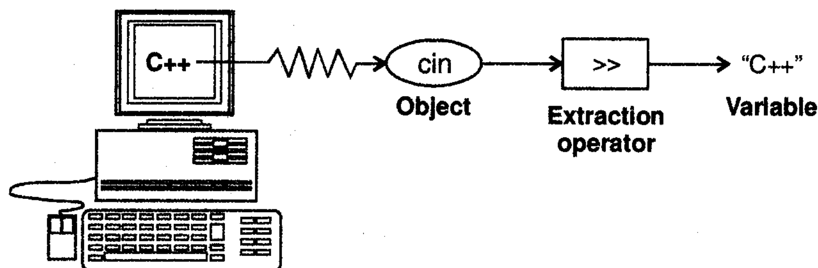


Figure 2.4: Input using extraction operator

For example:

```
int age;
cin >> age;
```

declares the variable age as an *int* type and then waits for an input from cin (keyboard) in order to store it in this integer variable.

cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character **cin** will not process the input until the user presses RETURN once the character has been introduced.

You must always consider the *type* of the variable that you are using as a container with **cin** extraction. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

Program using cin and cout

```
#include<iostream>
using namespace std;
int main()
{ int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << "\n";
  return 0;
 }
```

Output

Please enter an integer value: 920

The value you entered is 920 and its double is 1840.

The user of a program may be one of the reasons that provoke errors even in the simplest programs that use **cin** (like the one we have just seen). Since if you request an integer value and the user introduces a name (which is a string of characters), the result may cause an error. You can also use **cin** to request more than one datum input from the user: `cin >> a >> b;`

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable **a** and another for variable **b** that may be separated by any valid blank separator: a space, a tab character or a new line.

EXERCISES

A. Review Questions

1. What are the features of the C++ language?
2. How can a comment be written in a C++ program?
3. Explain the main() function in C++.
4. How you give comment in C++?
5. Explain the structure of a C++ program.

B. Programming Exercises

1. Write a C++ program that prints the following message on screen "My first C++ program". (Use cout statement).
2. Find errors if any in the following C++ statements:

   i. cout >> “X=”x; ii. cin <<x;<<y;

   iii. cout <<\n”Name:” << name iv. cout << “Enter value:”

3 Expression

I. Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meanings unchanged. However, there are some exceptions and additions. In this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

The C++ Character Set

Basically a character is used to represent information. It can be an alphabet, digit or special symbol.

The C++ character set consists of upper and lower case alphabets, digits, and special characters. The alphabets and digits are together called the alphanumeric characters.

i. Alphabets

A B C ....Z

a b c ....z

ii. Digits

0 1 2 3 4 5 6 7 8 9

iii. Special characters

, . ; : # ' " ! ¦ ~ < > { } ( ) - _ $ % & ^ * + [ ] / \

2. C++ tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, constants, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.

Tokens are usually separated by "white space." White space can be one or more: blanks, horizontal or vertical tabs, new lines, formfeeds, comments, etc.

C++ programs are written using these tokens, white spaces and the syntax of the language. Most of the C++ tokens are basically similar to the C tokens with the exception of some additions and minor modifications.

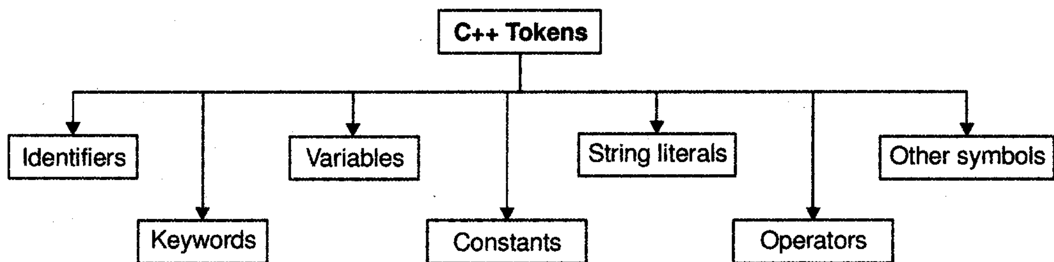


Figure 3.1: C++ tokens

2.1 Identifiers and Keywords

▶ Identifiers

Identifiers can be defined as the name of the variables, functions, class and arrays in our programs. These names, or identifiers, are required to conform to some simple rules.

An identifier must start with a letter and is composed of a sequence of letters, digits and underscore. The identifier name cannot start with a digit. While ANSI C allows the use of only 32 characters, there is no restriction on the length of an identifier.

Identifiers beginning with an underscore followed by an upper case letter are reserved for use by the implementation, as are identifiers containing two consecutive underscores, so as to avoid problems you should refrain from using them for your own identifiers.

There are identifiers reserved explicitly for C++ language. These are used to implement the specific features of the language and are called keywords.

You can't use an identifier that is a C++ keyword as a variable name in your programs.

You should also try to avoid using names from the C++ library, *for example:* swap, max.

C++ is case sensitive, so upper case and lower case characters are distinct. This means that the names "userInput" and "userinput" are recognised as two different identifiers.

Examples of acceptable identifiers are:

calculate_height, readWindSpeed, channel42, foo

Examples of unacceptable identifiers:

calculate height	– space between characters of a single variable	– invalid
delete	– it is keyword	– invalid
2letters	– variable begins with a digit	– invalid
a()test	– use of special characters	– invalid

▶ Keywords

Keywords are the words, which have been defined, in the C++ compiler.

Keywords are **reserved words** and are predefined by the language. They cannot be used by the programmer as variables or identifiers. It is mandatory that all the keywords should be in lower case letters.

Following are the keywords:

The reserved words for the ANSI C language are:

auto	double	int	struct	break	else	long	switch
case	enum	register	typedef	char	extern	return	union
const	float	short	unsigned	continue	for	signed	void
default	goto	sizeof	volatile	do	if	static	while

The following reserved words have been added for C++

catch	protected	delete	public	friend	template	inline	this
new	throw	operator	try	private	virtual	wchar_t	

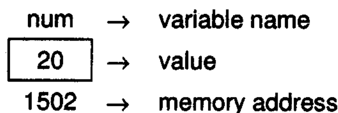
In addition, the X3J16 Technical Committee has proposed that the following keywords be added to the language definition for C++.

and	and_eq	bitand	bitor	bool	compl	not	not_eq	or	or_eq	xor	xor_eq

2.2 Variable

A **variable** name is an identifier or symbolic name assigned to the memory location where data is stored. In other words, it is the data name that refers to the stored value.

A simple variable can have only one value assigned to it at any given time during program execution. Its value may change during the execution of the program.



▶ Rules regarding Naming Variables

i. Since the variable name is an identifier, the same rules apply.

ii. Meaningful names should be given so as to reflect the value it is representing.

Example

studentname	rank1
basic_sal	amount
roll_num	No_of_years

Unlike C, in C++ a variable can be declared at the place of its first use in a program. This makes the program much easier to write and reduces the errors. It also makes it easy to understand the context in which the variable are declared and used.

2.3 Constants (Literals)

Constants
i. Integer
ii. Floating point
iii. Character
iv. String Literal
v. Enumeration
vi. Defined
vii. Declared

Constants or literals refer to fixed values that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integer numbers, floating-point numbers, characters and strings.

Literal constants do not have memory locations.

i. Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents.

They always begin with a digit. You can specify integer constants in decimal, octal or hexadecimal form. They can specify signed or unsigned types and long or short types.

They are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program we will be referring to the value 1776.

An integer constant has to follow the following rules:

a. It contains a sequence of digits from 0 to 9. (Octal contains digits from 0 to 7; Hexadecimal constant contains digits from 0 to 9 and letters A - F or a - f.)

b. An octal constant is preceded with 0 character (zero character) and hexadecimal constant with 0X or 0x (zero, x).

c. No commas, spaces or other symbols are allowed in between.

d. The integer can be either positive or negative. It may or may not be prefixed by a – sign.

e. A size or sign qualifier can be appended at the end of the constant.

U or u for unsigned. S or s for short L or l for long.

Example

123	56789U (unsigned integer)
-31000	7689909L (long integer)
0170	OX34ADL (long hexadecimal)
Ox 2A	6578890994 UL (unsigned long integer)
-100 s	120US (unsigned short)

ii. Floating Point Constants

These are real numbers having a decimal point or an exponential or both. The rules governing the floating point representation are:

a. They have a decimal point and digits from 0 to 9.

b. No embedded spaces, commas and other symbols are allowed.

c. They may or may not be prefixed by a – sign.

d. It is possible to omit digits before or after the decimal point.

Example: 0.246   975.64   – .54   +5.

Exponential notation

This is used to represent real numbers whose magnitude is very large or very small.

Format

```
   mantissa e exponent
Or mantissa E exponent
```

1. The **mantissa** can be a floating point number or an integer.
2. It can be positive or negative.
3. The **exponent** has to be an integer with optional plus or minus sign.

Example: The number 231.78 can be written as 0.23178e3 representing 0.23178×10^{3}.

75000000000 can be written as 75e9 or 0.75e11.

0.0000045 can be written as 0.45e – 5.

iii. Character Constants

A character constant is any single character enclosed within single quotes.

Example:
```
char c;
c = 'A';
```

stores a single character **A** into the character variable **c**.

The value of the character constant is the numeric value (ASCII) of the character. Every character constant requires 1 byte of memory.

Example: The character constant '0' has ASCII value 48, which is unrelated to numeric digit 0.

Difference between x and 'x' is that **x** refers to variable **x**, whereas **'x'** refers to the character constant **'x'**.

Escape Sequences

Like C, C++ supports some special character constants used in output functions (cout) to control cursor movement on the video display device. Some dot matrix printers also support the cursor movement escape sequences to move the print head on the page.

They are also called backslash character constants because they contain a backslash and a character. Although they look like two characters, they represent only one.

The complete set of escape sequence is

Character	Meaning	Action
\a	alert (bell)	sounds a beep
\b	backspace	moves cursor one position to the left of current position
\f	form feed	advances computer stationery to start of next page
\n	newline	moves cursor to start of next line
\r	carriage return	takes cursor to beginning of same line
\t	horizontal tab	moves cursor to next tab stop
\v	vertical tab	moves cursor to next vertical tab stop
\0	null character	It is used to terminate a string
\\	Backslash	used to display a \
\?	question mark	used to display a ?
\'	single quote	used to display a '
\"	double quote	used to display a "
\ϕϕϕ	octal number	Each 0 represents an octal digit
\xhh	hexadecimal number	Each h represents a hexadecimal digit (0–9,a–f)

iv. String Literal

A string constant or string literal is a sequence of zero or more characters enclosed in double quotes.

Example:
```
"Welcome to C++"
"First Line \n Second Line"
```

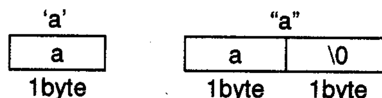
The double quotes are not a part of the string but only act as **delimiters.** If the backslash or double quote is required to be a part of the string, they must be preceded by a backslash (\).

Example: `cout<< "\\ is a backslash";` displays `\ is a backslash`

Technically, the internal representation of a string has a null character ('\0') at the end. Therefore the physical storage required is one more than the number of characters in the string.

Difference between 'a' and "a"

'a' is a character constant and stored as the numeric value of a. "a" is a string literal and consists of the characters, a and '\0'.



v. Enumeration Constant

An enumeration is a user defined type that enables the user to define the range of values for the type. Named constants are used to represent the values of an enumeration.

For example

```
enum
weekday{monday,tuesday,wednesday,thursday,friday,saturday,sunday};
weekday currentDay = wednesday;
if(currentDay==tuesday)
{
   // do something
}
```

The default values assigned to the enumeration constants are zero-based, so in our example above monday == 0, tuesday == 1, and so on.

The user can assign a different value to one or more of the enumeration constants, and subsequent values that are not assigned a value will be incremented.

For example: `enum fruit{apple=3, banana=7, orange, kiwi};`

Here, orange will have the value 8 and kiwi 9.

vi. Defined Constants (#define)

You can define your own names for constants that you use quite often without having to resort to variables, simply by using the **#define** preprocessor directive.

Syntax

```
#define identifier value
```

For example:

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

They define three new constants. Once they are declared, you are able to use them in the rest of the code as any if they were any other constant.

For example:

```
circle = 2 * PI * r;
cout << NEWLINE;
```

In fact the only thing that the compiler does when it finds **#define** directive is to replace literally any occurrence of them (in the previous example, **PI**, **NEWLINE** or **WIDTH**) by the code to which they have been defined (**3.14159265**, **'\n'** and **100**, respectively). For this reason, #define constants are considered *macro constants*.

The #define directive is not a code instruction, it is a directive for the preprocessor, therefore it assumes the whole line as the directive and does not require a semicolon (;) at the end of it. If you include a semicolon character (;) at the end, it will also be added when the preprocessor will substitute any occurrence of the defined constant within the body of the program.

vii. Declared Constants (const)

With the **const** prefix you can declare constants with a specific type exactly as you would do with a variable:

```
const int width = 100;
const char tab = '\t';
const zip = 12440;
```

In case that the type was not specified (as in the last example) the compiler assumes that it is type **int**.

The declared constants are memory locations whose values cannot be changed. Their values cannot be modified by the program in any way. As seen in the examples above, a name is associated with a constant value or expression just like a variable.

A *const* declaration is local to the file of its declaration. To make them global these must be explicitly defined as *extern*.

```
extern const int width = 12440;
```

3. Data Types

When we wish to store data in a C++ program, such as a whole number or a character, we have to tell the compiler which type of data we want to store. The type will have characteristics such as the range of values that can be stored and the operations that can be performed on variables of that type.

Data types in C++ can be classified under various categories as shown in *figure 3.2.*

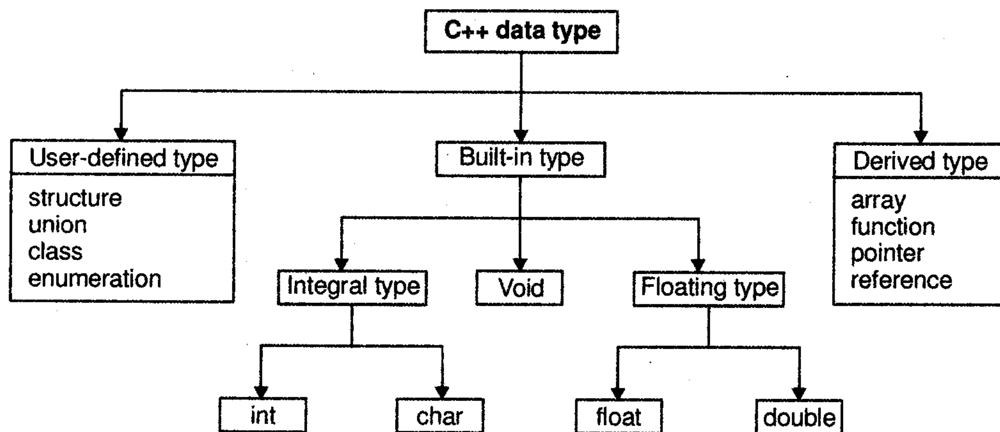


Figure 3.2: Hierarchy of C++ data types

3.1 Fundamental Types

C++ provides the following fundamental built-in types: Boolean, character, integer and floating-point. It also enables us to create our own user-defined types using enumerations and classes.

For each of the fundamental types, the range of values and the operations that can be performed on variables of that type are determined by the compiler. Each compiler should provide the same operations for a particular type but the range of values may vary between different compilers.

Name	Bytes	Description	Range
char	1	character or integer 8 bits length.	signed: -128 to 127 unsigned: 0 to 255
int	2	Integer. Its length traditionally depends on the length of the system's Word type, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).	*signed*: -32768 to 32767 *unsigned*: 0 to 65535
short	2	Type short int (or simply short) is an integral type that is larger than or equal to the size of type char, and shorter than or equal to the size of type int.	*signed*: -32768 to 32767 *unsigned*: 0 to 65535
long	4	Type long (or long int) is an integral type that is larger than or equal to the size of type int. Objects of type long can be declared as signed long or unsigned long. Signed long is a synonym for long.	*signed*: –2147483648 to 2147483647 *unsigned*: 0 to 4294967295
float	4	floating point number.	3.4e - 38 to 3.4e - 38 (7 digits)
double	8	double precision floating point number.	1.7e - 308 to 1.7e + 308 (15 digits)
long double	10	long double precision floating point number.	3.4e - 4932 to 1.1e + 4932 (19 digits)
bool	1	Boolean value. It can take one of two values: true or false. *Note:* This is a type recently added by the ANSI-C++ standard. Not all compilers support it.	true or false
wchar_t	2	Wide character. It is designed as a type to store international characters of a two-byte character set. *Note:* This is a type recently added by the ANSI-C++ standard. Not all compilers support it.	wide characters

Note

1. Signed means the number can be positive or negative.
2. Unsigned means the number must be positive.
3. Values of columns Bytes and Range may vary depending on your system. The values included here are the most commonly accepted and used by almost all compilers.

3.2 User Defined Data Types

User Defined Data Types
i. Structure
ii. Unions
iii. Class
iv. Enumerations

i. Structure

We have used user-defined data types such as struct and union in C, while these data types are not suitable for programming in C++, some more features have been added to make them suitable for object-oriented programming.

For example

```
struct book
{
   char book_id[6];
   char book_name[25];
   char author [25];
   char category[15];
   float price;
};
```

ii. Unions

Unions allow a portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union model_name
{
   type1 element1;
   type2 element2;
   type3 element3;
   .
   .
}object_name;
```

All the elements of the *union* declaration occupy the same space of memory. Its size is one of the greatest element of the declaration. *For example*:

```
union mytypes_t {  char c;
                   int i;
                   float f;
                 }mytypes;
```

defines three elements:

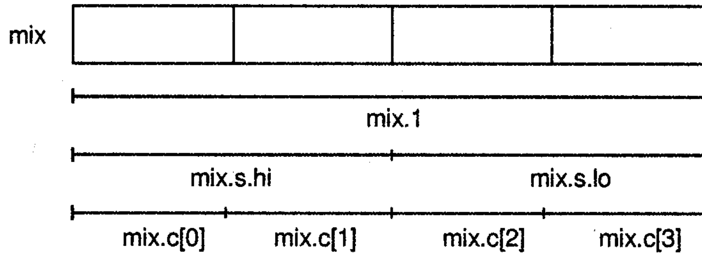
```
mytypes.c
mytypes.i
mytypes.f
```

each one of a different data type. Since all of them are referring to a same location in memory, the modification of one of the elements will affect the value of all of them.

One of the uses a *union* may have is to unite an elementary type with an array or structures of smaller elements. *For example*

```
union mix_t{ long l;
          struct {
                 short hi;
                 short lo;
                 } s;
           char c[4];
          }mix;
```

defines three names that allow us to access the same group of 4 *bytes*: **mix.l**, **mix.s** and **mix.c** and which we can use according to how we want to access it, as **long**, **short** or **char** respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data:



Anonymous Unions

In C++, we have the option that unions be anonymous. If we include a union in a structure without any object name (the one that goes after the curly brackets { }) the union will be anonymous and we will be able to access the elements directly by its name.

For example: Look at the difference between these two declarations:

Union	Anonymous union

struct {
char title[50];
char author[50];
union {
float dollars;
int yens;
} price;
} book;
	struct {
	char title[50];
	char author[50];
	union {
	float dollars;
	int yens;
	};
	} book;

The only difference between the two pieces of code is that in the first one we gave a name to the union (**price**) and in the second we did not. The difference is when accessing members **dollars** and **yens** of an object. In the first case it would be:

```
book.price.dollars
book.price.yens
```

whereas in the second it would be:

```
book.dollars
book.yens
```

Once again I remind you that because it is a union, the fields **dollars** and **yens** occupy the same space in the memory so they cannot be used to store two different values. That means that you can include a price in dollars or yens, but not both.

iii. Class

A class enables us to create sophisticated user defined types. We provide data items for the class and the operations that can be performed on the data. *For example*: To create a square class that has a data item for size, and provides draw and resize operations:

```
class square {
        public:
                square();
                ~square();
                void draw();
```

```
            bool resize(int newSize);
    private:
            int size;
};
```

iv. Enumerations (enum)

Enumerations serve to create data types to contain something different that is not limited to either numerical or character constants nor to the constants **true** and **false**. Its form is the following:

```
enum model_name{ value1, value2, value3, . . .} object_name;
```

For example: We could create a new type of variable called **color** to store colors with the following declaration:

```
enum colors_t{black, blue, green, cyan, red, purple, yellow, white};
```

Notice that we do not include any fundamental data type in the declaration. To say it in another way, we have created a new data type without it being based on any existing one: the type **colors_t**, whose possible values are the colors that we have enclosed within curly brackets {}. *For example:* Once declared the **colors_t** enumeration in the following expressions will be valid:

```
colors_t mycolor;
mycolor = blue;
if(mycolor == green) mycolor = red;
```

In fact our enumerated data type is compiled as an integer and its possible values are any type of integer constant specified. If it is not specified, the integer value equivalent to the first possible value is **0** and the following ones follow a +1 progression. Thus, in our data type **colors_t** that we defined before, **black** would be equivalent to **0**, **blue** would be equivalent to **1**, **green** to **2** and so on.

If we explicitly specify an integer value for some of the possible values of our enumerated type (*For example*: the first one) the following values will be the increases of this,

For example

```
enum months_t {january=1, february, march, april, may, june,
july, august, september, october, november, december} y2k;
```

In this case, variable **y2k** of the enumerated type **months_t** can contain any of the 12 possible values that go from **january** to **december** and that are equivalent to values between **1** and **12**, not between **0** and **11** since we have made **january** equal to **1**.

3.3 Derived Data Types

i. Arrays

The application of arrays in C++ is similar to that in C. The only execution is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant.

For example:

```
char string[2] = "ab";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string[3]= "ab";  //O.K. for C++
```

ii. Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concepts of C++. Some of these were introduced to make the C++ program more reliable and readable.

iii. Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;            // int pointer
ip = &y;            // address of y assigned to ip
*ip = 20;           // 20 assigned to y through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char *const ptrl = "ABC"; //constant pointer
```

We cannot modify the address that ptr1 is initialized to

```
int const *ptr2=&x;       //pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char *const cp = "ab";
```

This statement declares cp as a constant pointer to the string which has been declared a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

Pointers are extensively used in C++ for memory management and achieving polymorphism.

3.4 Definition of own types (typedef)

C++ allows us to define our own types based on other existing data types. In order to do that we shall use keyword **typedef**, whose form is:

```
Typedef  existing_type  new_type_name ;
```

where existing_type is a C++ fundamental or any other defined type and new_type_name is the name that the new type we are going to define will receive.

For example:
```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field[50];
```

In this case we have defined four new data types: **C**, **WORD**, **string_t** and **field** as **char**, **unsigned int**, **char*** and **char[50]** respectively, that we could perfectly use later as valid types:

```
C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;
```

typedef can be useful to define a type that is repeatedly used within a program and it is possible that we will need to change it in a later version, or if a type you want to use has too long a name and you want it to be shorter.

4. Declaration of Variables

In order to use a variable in C++, we must first declare it specifying which of the data types we want it to be. The syntax to declare a new variable is to write the data type specifier that we want (like **int**, **short**, **float**...) followed by a valid variable identifier.

Syntax

```
<data_type> <variable_name>;
```

For example:
```
int number;
float mynumber;
```

Are valid declarations of variables. The first one declares a variable of type **int** with the identifier *number*.

The second one declares a variable of type **float** with the identifier *mynumber*. Once declared, variables *number* and *mynumber* can be used within the rest of their scope in the program. *For example*: The following program illustrates the operation with variables.

```
// Operating with variables
#include<iostream>
using namespace std;
int main()
{ // declaring variables:
  int a, b;
  int result;
  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
  // display the result:
  cout << result;
  // terminate the program:
  return 0;
}
```

All the variables that we are going to use must have been previously declared. An important difference between the C and C++ languages, is that in C++ we can declare variables anywhere in the source code, even between two executable sentences, and not only at the beginning of a block of instructions, which happens in C.

Variables will be declared in three basic places:

Location of declarations	Name of the variable
Inside Functions	Local variables
In the definition of function parameters	Formal parameters
Outside of all functions	Global Variables

i. **Local Variables:** Variables that are declared inside a function are called local variables. In some cases these variables are referred to as automatic variables.

Local variables may be referenced only by statements that are inside the block in which the variables are declared. In other words, local variables are not known outside their own code block.

ii. **Formal Parameters:** If a function is to use arguments, it must declare variables that will accept the values of the arguments.

These variables are called the formal parameters of the functions. They behave like any other local variables inside the functions.

For example:

```
void sub(int x, int y)
{
   : : : } function body
}
```

In the above example, sub is a function which accepts two integer values in variables x and y.

It could be also written as follows:

```
void sub(x, y)
int x;
int y;
{
  function body
}
```

iii. **Global Variables:** Unlike local variables, global variables are known throughout the program and may be used by any piece of code. Also, they will hold their values throughout the program's execution.

You create global variables by declaring them outside of any functions. Any expression can access them.

```
#include<iostream>
int a;       }
char name;   } Global Variables
char p[20];  }
main()
{
```

```
    unsigned short age;                 } Local Variables
    float firstnumber, secondnumber;

    cout << "Enter your age:";  } Instructions
    cin>>age;
    . . .
}
```

In addition to **local** and **global** scopes, there exists external scope, that causes a variable to be visible not only in the same source file but in all other files that will be linked together.

5. Initialization of Variables

When declaring a local variable, its value is undetermined by default. But you may want a variable to store a concrete value the moment that it is declared. In order to do that, you have to append an equal sign followed by the value wanted to the variable declaration:

```
type identifier = initial_value;
```

For example: If we want to declare an *int* variable called *x* that contains the value *3* at the moment in which it is declared, we could write: `int x = 3;`

Additionally to this way of initializing variables (known as C-like), C++ has added a new way to initialize a variable: by enclosing the initial value between parenthesis ():

```
type identifier (initial_value);
```

For example: `int a(0);`

Both ways are valid and equivalent in C++.

6. Reference Variables

C++ introduces a new kind of variable known as the reference variable. A reference is a variable name that is a duplicate of an existing variable. It provides a technique of creating more than one name to designate the same variable. The syntax of creating or declaring a reference is:

```
DataType &ReferenceName = VariableName;
```

To declare a reference, type the variable's name preceded by the same type as the variable it is referring to. Between the data type and the reference name, type the ampersand operator "&". To specify what variable the reference is addressed to, use the assignment operator "=" followed by the name of the variable. The referred to variable must exist already. You cannot declare a reference as:

```
int &Mine;
```

The compiler wants to know what variable you are referring to. Here is an example:

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

```
  int Number  = 228;
  int &Nbr  = Number;
}
```

The ampersand operator between the data type and the reference can assume one of three positions as followed:

```
int& Nbr;
int & Nbr;
int &Nbr;
```

As long as the & symbol is between a valid data type and a variable name, the compiler knows that the variable name (in this case Nbr) is a reference.

Once a reference has been initialized, it holds the same value as the variable it is pointing to. You can then display the value of the variable using either of both.

```
#include<iostream>
using namespace std;
main()
{
  int Number = 228;
  int & Nbr = Number;
  cout << "Number = " << Number << "\n";
  cout << "Its reference = " << Nbr << "\n\n";
}
```

If you change the value of the variable, the compiler updates the value of the reference so that both variables would hold the same value. In the same way, you can modify the value of the reference, which would update the value of the referred to variable.

To access the reference, do not use the ampersand operator; just the name of the reference is sufficient to the compiler. This is illustrated in the following program:

```
#include<iostream>
using namespace std;
main()
{
  int Number = 228; // Regular variable
  int& Nbr = Number; // Reference
  cout << "Number = " << Number << "\n";
  cout << "Its reference = " << Nbr << "\n";
  // Changing the value of the original variable
  Number = 4250;
  cout << "\nNumber = " << Number;
  cout << "\nIts reference = " << Nbr << "\n";
  // Modifying the value of the reference
  Nbr = 38570;
  cout << "\nNumber = " << Number;
  cout << "\nIts reference = " << Nbr << "\n\n";
}
```

In this way, you can use either a reference or the variable it is referring to, to request the variable's value from the user. Here is an example:

```
#include<iostream>
using namespace std;
main()
{
  double Price;
  double& RefPrice = Price;
  cout << "What's the price? $";
  cin >> Price;
  cout << "Price = $" << Price << "\n";
  cout << "Same as: $" << RefPrice << "\n\n";
  cout << "What's the price? $";
  cin >> RefPrice;
  cout << "Price = $" << Price << "\n";
  cout << "Same as: $" << RefPrice << "\n";
}
```

7. Operators

Arithmetic Operators		
Symbol	**Operator**	**Example**
*	Multiplication	a*b
/	Division	a/b
%	Modulo	a%b
+	Addition	a+b
–	Subtraction	a–b

Relational Operators		
Symbol	**Operator**	**Example**
<	Less than	a < b
>	Greater than	a > b
>=	Greater than or equal	a > = b
<=	Less than or equal to	a < = b
==	Equal to	a==b
!=	Not equal	a!=b

Logical Operators		
Symbol	**Operator**	**Example**
!	NOT	!(a < b)
&&	AND	a < b && c > d
||	OR	a || d

Increment and Decrement Operators		
Symbol	Operator	Example
++	Increment	a++ or ++a
– –	Decrement	a- - or - -a

<table>
<tr><th colspan="3">Assignment Operator</th></tr>
<tr><th>Symbol</th><th>Operator</th><th>Example</th></tr>
<tr><td>=</td><td>Assignment</td><td>a = b</td></tr>
<tr><td>+=</td><td>Addition and assignment</td><td>a += b same as
a = a + b</td></tr>
<tr><td>–=</td><td>Subtraction and assignment</td><td>a – = b same as
a = a – b</td></tr>
<tr><td>*=</td><td>Multiplication and assignment</td><td>a*=b same as
a = a*b</td></tr>
<tr><td>/=</td><td>Division and assignment</td><td>a/=b same as
a = a/b</td></tr>
<tr><td>%=</td><td>Modulo and assignment</td><td>a%=b same as
a = a%b</td></tr>
<tr><td>&=</td><td>bitwise AND and assignment</td><td>a&=b same as
a = a&b</td></tr>
<tr><td>|=</td><td>bitwise OR and assignment</td><td>a|=b same as
a = a|b</td></tr>
<tr><td>^=</td><td>bitwise XOR and assignment</td><td>a^=b same as
a = a^b</td></tr>
<tr><td><<=</td><td>shift left and assignment</td><td>a<<=b same as
a = a<<2</td></tr>
<tr><td>>>=</td><td>shift right and assignment</td><td>a>>=4 same as
a = a>>b</td></tr>
</table>

<table>
<tr><th colspan="2">Bitwise Operators</th></tr>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>~</td><td>One's complement (NOT)</td></tr>
<tr><td><<</td><td>Left Shift</td></tr>
<tr><td>>></td><td>Right shift</td></tr>
<tr><td>&</td><td>Biwtise AND</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
</table>

i. The Conditional Operator

The conditional operator in C and C++ is only ternary operator. It works on three values as opposed to the binary operators you have seen that operate on only two values. The conditional operator is used to replace if-else logic in some situations. It is a two-symbol operator, **?:**, with the following format:

```
result = conditional_expression ? expression1 : expression2;
```

The **conditional_expression** is any expression in C and C++ that results in a True (nonzero) or False (zero) answer. If the result of **conditional_expression** is true, **expression1** executes. Otherwise, if the result of **conditional_expression** is false, **expression2** executes. Only one of the expression following the question mark ever executes. Only a single semicolon appears at the end of **expression2**. The internal expressions, such as **expression1**, do not have a semicolon. The resultant value generated by the expression that is executed is returned and can be captured into the result identifier.

If you require simple if-else logic, the conditional operator usually provides a more direct and succinct method, although you should always prefer readability over compact code.

To see how the conditional operator work, consider the section of code that follows:

```
if(a > b)
   ans = 10;
else
   ans = 25;
```

You can easily rewrite this kind of **if-else** code by using a single conditional operator.

```
ans = a > b ? 10 : 25;
```

ii. sizeof Operator

The **sizeof** operator returns the physical size, in bytes, of the data item for which it is applied. It can be used with any type of data item except bit fields. The general form is:

```
size_t sizeof(item);
```

When **sizeof** is used on a character field the result returned is 1 (if a character is stored in one byte). When used on an integer the result returned is the size in bytes of that integer. When used on an array the result is the number of bytes in the array, not the number of characters which appear before a NULL. In the ANSI standard the **sizeof** operator returns a data type of **size_t** which is usually an **unsigned int** value.

```
int nums[10];
cout<<" There are" <<sizeof(nums) <<"types in the array
and"<<sizeof(nums) / sizeof(int) <<"elements";
```

iii. Comma Operator

The comma ',' operator is used to separate a set of expressions. A pair of expressions separated by a comma is evaluated left to right and the type and value of the result is the type and value of the right operand.

Example: Consider `i =(j = 3 , j + 2);`

Here, the right hand side contains two expressions j = 3 and j + 2 which are evaluated L →R Thus 3 is first assigned to j and the value 3 + 2 is assigned to i.

It could also be the used to interchange the values of two variables in a single statement as shown.

```
temp = a, a = b, b = temp;
```

The comma operator has the lowest precedence and associates from L → R.

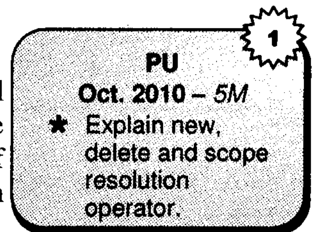
7.1 New Operators

All C operators are valid in C++ also. In addition, C++ introduces some new operators. List of new operators is given below.

Operators	Meaning
endl	Line feed operator
new	Memory allocation operator
delete	For releasing memory
>>	Extraction operator
<<	Insertion operator
::	Scope Resolution operator
:: *	Pointer - to member declarator
. *	Pointer - to member operator
→ *	Pointer- to member operator
setw	Field width operator

In addition, C++ also allows us to provide new definitions to some of the built in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as operator overloading.

i. Scope Resolution Operator (::)



The Scope Resolution operator (::) allows access to the global variable even though there is a local variable of the same name within the main() function. The use of the double colon in front of the variable name instructs the system that we are interested in using the global variable name rather than the local variable.

The use of this technique allows access to the global variable for any use. It could be used in calculations, as a function parameter, or for any other purpose. It is not really good programming practice to abuse this construct, because it could make the code difficult to read. It would be best to use a different variable name instead of reusing this name, but the construct is available to you if you find that you need it sometime.

Consider the example illustrating the use of scope resolution operator.

Program: :: scope resolution operator

```
int i = 1;          // external or global i
#include<iostream>
using namespace std;
int main()
{
int i = 2;                          // redeclares i locally to main
 {                                  // an inner block within a function
    cout << "Enter inner block" << endl;
    int n = i;                      // the global i is still visible
    int i = 3;                 // hides the global i which can only be
                                // referenced by using the :: operator
    // print the local i and the global i
    cout << i << " i <> ::i " << ::i << endl;
    cout << "n = " << n << endl;
```

```
  }    // end of inner block
  cout << "Enter outer block" << endl;
  //   print the current local i and the global i
  cout << i << " i <> ::i " << ::i << endl;
  return 0;
 }
```

Output

```
Enter inner block
3 i <> ::i 1
n = 2
Enter outer block
2 i <> ::i 1
```

You can also use the class scope operator (Scope Resolution Operator) to qualify class names or class member names.

If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

In the following example, the declaration of the variable X hides the class type X, but you can still use the static class member count by qualifying it with the class type X and the scope resolution operator.

```
#include<iostream>
using namespace std;
class X
{ public:
     static int count;
};
int X::count = 10;                  // define static data member
int main()
{  int X = 0;                      // hides class type X
   cout << X::count << endl;       // use static member of class X
 }
```

ii. Member Dereferencing Operators

a. :: : This operator is used to declare to a member of a class.

b. * : This operator is used to access a member using object name and a pointer to that member.

c. →* : This operator is used to access a member using a pointer to the object and a pointer to that member.

iii. Memory Management Operators

In C, the dynamic memory allocation typically involves a call to malloc(), which is paired with free() to deallocate the memory.

C++ defines a new method for carrying out memory allocations and deallocations, i.e.. using the new and delete operators that are discussed in *(Section 9)*.

7.2 Precedence and Associativity of Operators

The operators are listed in order of decreasing precedence.

The Operators grouped together in one level have the same precedence.

Priority	Operator	Description	Associativity
1.	::	Scope operator	Left → Right
2.	( )	Function call	Left → Right
	[ ]	Array element reference	
	→	Pointer to structure member reference	
	•	Structure member reference	
3.	–	Unary Minus	Right → Left
	+	Unary plus	
	++	Increment	
	– –	Decrement	
	!	Logical negation	
	~	One's complement	
	*	Pointer reference (indirection)	
	&	Address	
	sizeof	Size of an object	
	type	Type cast	
	new	New operator	
	delete	Delete operator	
4.	*	Multiplication	Left → Right
	/	Division	
	%	Modulo division	
5.	+	Addition	Left → Right
	-	Subtraction	
6.	<<	Left shift	Left → Right
	>>	Right shift	
7.	<	Less than	Left → Right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
8.	==	Equality	Left → Right
	! =	Inequality	
9.	&	Bitwise AND	Left → Right
10.	^	Bitwise XOR	

<table>
<tr><td>11.</td><td>|</td><td>Bitwise OR</td><td></td></tr>
<tr><td>12.</td><td>&&</td><td>Logical AND</td><td rowspan="2">Left → Right</td></tr>
<tr><td>13.</td><td>||</td><td>Logical OR</td></tr>
<tr><td>14.</td><td>? :</td><td>Conditional</td><td>Left → Right</td></tr>
<tr><td>15.</td><td>= *= /=
%= += -=
& = ^=
|=
<< = >>=</td><td>Assignment</td><td>Right → Left</td></tr>
<tr><td>16.</td><td>,</td><td>Comma</td><td>Left → Right</td></tr>
</table>

8. Type Cast Operator

C++ allows explicit type conversion of variables or expression using type cast operator. Its syntax is similar to the syntax of function-call.

Syntax

```
type-name(expression) // C++ notation
```

A *type-name* followed by an *expression* enclosed in parenthesis constructs an object of the specified type using the specified expressions. The following example shows an explicit type conversion to type int:

For example:

```
int x;
float y = 2.18;
x = int(y);
```

Here the float value gets typecast to an integer value and the value 2 alone is stored.

Explicit type conversions can also be specified using the "cast" syntax, i.e., C syntax.

Syntax

```
(type-name)expression// C notation
```

The previous example, rewritten using the cast syntax, is:

```
x = (int)y;
```

Both cast and function-style conversions have the same results when converting from single value. However, in the function-style syntax, you can specify more than one argument for conversion. This difference is important for user-defined types.

But the function-style syntax usually leads to simplest expression and can be used only if the type is an identifier. *For example*

```
a=int*(b);
```

is illegal. In such cases we must use the C type notation.

```
a=(int*) b;
```

Alternatively, we can use typedef to create an identifier of the required type and use it in the function-style syntax.

```
typedef int * int_pt
a=int_pt(b);
```

ANSI C++ introduces four new casting operators:

i. static_cast, to convert one type to another type;
ii. const_cast, to cast away the "const-ness" or "volatile-ness" of a type;
iii. dynamic_cast, for *safe* navigation of an inheritance hierarchy; and
iv. reinterpret_cast, to perform type conversions on un-related types.

9. Memory Management Operators

Until now, in our programs, we have only had as much memory as we have requested in declarations of variables, arrays and other objects that we included, having the size of all of them fixed before the execution of the program. But, what if we need a variable amount of memory that can only be determined during the program execution (runtime), *For example*: In case what we need a user input to determine the necessary amount of space?

The answer is *dynamic memory*, for which C++ integrates the operators *new* and *delete*.

9.1 new Operator

In order to request dynamic memory, the operator new exists. *new* is followed by a data *type* and optionally the number of elements required within brackets [].

It returns a pointer to the beginning of the new block of assigned memory. Its form is:

```
pointer = new type
```

or

```
pointer = new type[elements]
```

The first expression is used to assign memory to contain one single element of *type*. The second one is used to assign a block (an array) of elements of *type*.

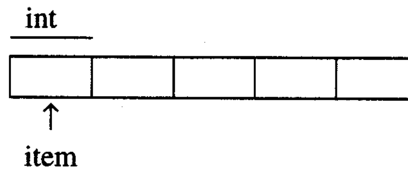
For example

```
int *P = NULL; // define an initially empty pointer
P=new int;//allocates memory for a new integer and assigns its location to
  //"P"
P=new int[100];//allocates a array of 100 int and assigns its location to
  //"P"
```

Consider the following example

```
int * item;
item = new int[5];
```

In this case, the operating system has assigned space for 5 elements of type int in a heap and it has returned a pointer to its beginning that has been assigned to item. Therefore, now, item points to a valid block of memory with space for 5 int elements.



The most important difference between declaring a normal array and assigning memory to a pointer is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows assigning memory during the execution of the program using any variable, constant or combination of both as size.

The dynamic memory is generally managed by the operating system, and in multitask interfaces it can be shared between several applications, so there is a possibility that the memory exhausts. If this happens and the operating system cannot assign the memory that we request with the operator new, a null pointer will be returned. For that reason it is recommended to always check to see if the returned pointer is null after a call to new.

```
int * item;
item = new int[5];
if(item == NULL) {
  // error assigning memory. Take measures.
};
```

9.2 delete Operator

Since the necessity of dynamic memory is usually limited to concrete moments within a program, once it is no longer needed it should be freed so that it becomes available for future requests of dynamic memory. The operator delete exists for this purpose, whose form is:

```
delete pointer;
```

or

```
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for multiple elements (arrays). In most compilers both expressions are equivalent and can be used without distinction, although indeed they are two different operators and so must be considered for operator overloading.

```
delete P;//returns the memory allocated by the new integer back to the
//memory pool
delete []P; // returns the entire array to the memory pool
```

Program

```
#include<iostream>
using namespace std;
int main()
 {int i,n;
  long * l;
  cout << "How many numbers do you want to type in? ";
  cin >>i;
  l= new long[i];
  if(l==NULL) exit(1);
```

```
for(n=0; n<i; n++)
{ cout<<"Enter number: ";
  cin>>l[n];
}
cout<<"You have entered: ";
for(n=0; n<i; n++)
{  cout<<l[n] << ", ";}
  delete[]l;
  return 0;
}
```

Output

```
How many numbers do you want to type in? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

This simple example that memorizes numbers does not have a limited amount of numbers that can be introduced, thanks to us requesting to the system to provide as much space as is necessary to store all the numbers that the user wishes to introduce.

NULL is a constant value defined in manifold C++ libraries specially designed to indicate null pointers. In case that this constant is not defined, you can do it yourself by defining it to 0:

#define NULL 0

It is indifferent to put 0 or NULL when checking pointers, but the use of NULL with pointers is widely extended and it is recommended for greater legibility. The reason is that a pointer is rarely compared or set directly to a numerical literal constant except precisely number 0, and this way this action is symbolically masked.

10. Expression

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may include function calls, which return values. An expression may consists of one or more operands, and zero or more operators to produce a value.

Types of Expression

i. **Constant Expressions** consists of only constant values.

   Example: 20 / 4+24, 'p', 2

ii. **Integral Expressions** produce integer results after implementing all the automatic (implicit type) and explicit type conversions.

   Example: 4+int(6.0), n * y, where n, y and p are integer variables.

iii. **Float Expressions** produce floating point results after all conversions.
Example: 45.10, p + q, p*q/5 where p and q are floating point variables.

iv. **Pointer Expressions** produce address values.
Example: &p, "pqr", ptr

v. **Relational Expressions or Boolean Expressions** returns result of type bool which takes a value true (1) or false (0).
When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results are compared.
A relational expression is always a logical expression. Logical expressions are either relational expressions or a combination of multiple expressions joined together by the logical operators: &&, ||, and!.
Examples of relational expressions (also logical expressions) are as follows:
```
1.   a< 2 * b - 7
2.   c != -1
3.   b > c + 4 * 7
```

vi. **Logical Expressions** combines two or more relational expressions and produce bool type results.
Example of a logical expression (not a relational expression):
```
(a < b) || ( b < c)
```
If a = 5, b = 3, and c = 10, the result of this expression is 1 (true).
A quick way to tell if an expression is logical but not relational is if one of the logical operators is being used.

vii. **Bitwise Expressions** are used to manipulate data at bit level. They are basically used for testing or shifting bits.
Examples:
```
y >> 3        // shift 3 bit position to right
p << 2        // shift 2 bit position to left
```
Shift operators are often used for multiplication and division by powers of two.

11. Statement

C++ statements are the program elements that control how and in what order objects are manipulated. A statement is composed of expressions and operators and it is a complete instruction instructing the compiler to carry out some task. Statements always end with a semicolon except the preprocessor directive.

Categories of statements are as follows:

i. **Expression statements:** These statements evaluate an expression for its side effects or for its return value.

ii. **Null statements:** These statements can be provided where a statement is required by the C++ syntax but where no action is to be taken.

iii. **Compound statements:** These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever the grammar calls for a single statement.

iv. **Selection statements:** These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false (zero).

v. **Iteration statements:** These statements provide for repeated execution of a block of code until a specified termination criterion is met.

vi. **Jump statements:** These statements either transfer control immediately to another location in the function or return control from the function.

vii. **Declaration statements:** Declarations introduce a name into a program. (Declarations provide more detailed information about declarations.)

12. Symbolic Constant

Symbolic Constants are memory locations whose contents also cannot be changed. Or we can say that a name used to represent a constant value within a program is called as Symbolic Constants.

There are two ways of creating Symbolic Constants

i. Using the keyword **const** and

ii. Defining a set of integer constants using **enum** keyword.

i. **Using the keyword const:** The keyword const can be added to a declaration to make an object a constant. Once you declared an object as a constant its value cannot be modified by the program in any way. Such an object must be initialized; it cannot be assigned a value. A const variable can serve as a symbolic constant in programs. Here, you can associate (declare) a name with a constant expression similar to the way you associate a name with a variable.

Example: `const float pi = 3.1415;`

A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. To give const values an external linkage so that it can be referenced from another file, we must explicitly define it as an extern in C++.

For example: `extern const total=100;`

ii. **Using the keyword enum:** Another method of naming integer constants is by enumeration as under;

```
enum{P,Q,R};
```

This defines P,Q and R as integer constants with values 0,1 and 2 respectively. This is equivalent to:

```
const P=0;
const Q=1;
const R=2;
```

We can also assign values to P,Q and R explicitly. *For example:*

```
Enum{P=100, Q=300, R=200};
```

Such values can be any integer values. Enumerated data type has been already discussed in 3.2.

13. Type Compatibility

The concept of compatible type combines the notions of being able to use two types together without modification (as in an assignment expression), being able to substitute one for the other without modification, and uniting them into a composite type. A *composite type* is that which results from combining two compatible types. Determining the resultant composite type for two compatible types is similar to following the usual binary conversions of integral types when they are combined with some arithmetic operators. Obviously, two types that are same are compatible; their composite type is also the same type. Less obvious are the rules governing type compatibility of non-identical types, function prototypes, and type-qualified types. In C, Names in `typedef` definitions are only synonyms for types, and so `typedef` names can possibly indicate identical and therefore compatible types. Pointers, functions, and arrays with certain properties can also be compatible types. A separate concept of type compatibility as distinct from being of the same type does not exist in C++. Generally speaking, type checking in C++ is stricter than in C: identical types are required in situations where C would only require compatible types. *For example*: int, short int and long int are three different types in C++. They must be cast when their values are assigned to one another. Similarly, unsigned char, char and signed char are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions are necessary in C++ in order to support function overloading where two functions with the same name are distinguished using the type of function arguments. Another notable difference between C and C++ is that in case of C the char constant are stored as ints but in C++, however, char is not promoted to the size of int. *For example*: in C, sizeof('x') is equivalent to sizeof(int), but in case of C++, sizeof('x') is equals to sizeof(char).

Solved Programs

1. Write a C++ program to find roots of Quadratic Equation $ax^2 + bx + c = 0$.

Solution

```
#include<iostream>
using namespace std;
void main()
{
   clrscr();
   int a, b ,c ;
   float t1, x1, x2;
   cout << "Enter values a, b, c ";
cin >> a >> b >> c;
t1 = sqrt(b * b - 4 * a * c);
   x1 = (-b + t1) / (2 * a);
   x2 = (-b - t1) / (2 * a);
   cout << "Roots of equation is:= " << x1 << x2;
}
```

2. Write a C++ program to read the values of a, b and c and display the value of x where $x = a / (b - c)$.

Solution

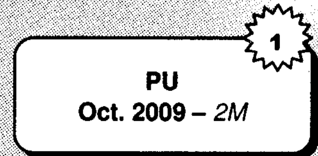
```
#include<iostream>
using namespace std;
void main()
{
  int a, b, c, temp;
  float x;
  cout << "\n Enter the values of a, b, c:= ";
  cin >> a >> b >> c;
  temp = b - c;
  x = a / temp;
  cout << " \n The value of x is:= " << x;
}
```

3. Write program to convert characters to integers.

Solution

```
#include<iostream>
using namespace std;
int main()
{
  int number;
  char character;
  cout << "Type any character:\n";
  cin >> character;
  number = character;
  cout << "The character is" << character;
  cout << " is represented as the number ";
  cout << number << " in the computer.\n";
  return 0;
}
```

4. What will be the output of following program?



i.
```
#include<iostream.h>
void main()
{
    int a=10;
    while(1)
    {
        switch(a)
        {
            case 10: cout<<a++;
            case 11: cout<<a--;
            case 12: cout<<a++;
        }
    }
}
```

Solution

Output: In this program display the value in Infinite loop.

ii.
```
# include<iostream.h>
# include<conio.h>
void main()
{
  char s[] = "CLASS";
  int i;
  for(i =0; s[i]; i++)
  cout <<"\n"<< s[i]<<*(s + i)<<*(i + s)<< i[s];
}
```

PU
Apr. 2010 – *2M*

Solution

Output of the given code is:

CCCC
LLLL
AAAA
SSSS
SSSS

Explanation: In the given code variable 's' is a character array (i.e., string), holding the string 'CLASS'. The given code is displaying each character of the string 'CLASS' four times on new lines. In the given code s[i], *(s+i), *(i+s) and i[s] are nothing but the ith (i=0, 1, 2, 3, 4) character in the string 'CLASS'.

iii.
```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    char s[] = "OBJECT";
    int i;
    for(i=0; s[i]; i++)
        cout<<"\n"<<s[i++]<<*(s+i++)<<*(i+s)<<i[s];
}
```

PU
Oct. 2010 – *2M*

Solution

Output of the given code is:

BOOO
CEEE

Explanation: During the first iteration the system stack would contain following values.

i[s] (i.e., s[0], i.e., O) → element at the bottom of stack

*(i+s) (i.e., s[0], i.e., O)

*(s+i++) (i.e., s[0], i.e., O and value of i would be incremented by 1, so i=1 now)

s[i++] (i.e., s[1], i.e., B and value of i would be incremented by 1 once again, so i=2 now) → element at the top of stack.

Therefore during first iteration, the output would be: BOOO

The next of the loop would increment the value of i by 1. So, now i=3. For the second iteration, stack contains would be:

i[s] (i.e., s[3], i.e., E) → element at the bottom of stack

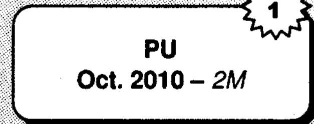
*(i+s) (i.e., s[3], i.e., E)

*(s+i++) (i.e., s[3], i.e., E, and value of i would be incremented by 1, so i=4 now)

s[i++] (i.e., s[4], i.e., C and value of i would be incremented by 1 once again, so i=5 now) → element at the top of stack

Therefore during second iteration, the output would be: CEEE

d.
```
void main()
{
    int a, *pa, &ra;
    pa = &a;
    ra = a;
    cout<<"a="<<a<<"*pa="<<*pa<<"ra="<<ra;
}
```



Solution

Above code would give the following error:
Reference variable ra must be initialized.
In the declaration statement,
int a, *pa, &ra;
ra is not initialized.

EXERCISES

A. Review Questions

1. List 5 rules for forming variable names.
2. List all possible data types. (All basic data types along with their possible modifiers or qualifiers.)
3. Write C++ instructions to ask a user to type in three numbers and to read them into integer variables first, second and third.
4. Write C++ instructions to output the value of a variable x in a line as follows:
   ```
   The value of x is ......
   ```
5. Write C++ instructions to generate output as follows:
   ```
   A circle of radius .....
           has area .....
    and circumference .....
   ```
   where the values of the radius, the area and the circumference are held in variables rad, area, and circum.
6. Correct the syntax errors in the following C++ program:
   ```
   include iostream.h
   main();
   {
     float x,y,z;
     cout < "Enter two numbers ";
     cin >> a >> b
     cout << "The numbers in reverse order are"<< b,a;    }
   ```
7. Show the form of output displayed by the following statements when total has the value 352.74.
   ```
   cout << "The final total is: " << endl;
   cout << "$" << total << endl;
   ```
   What data types would you use to represent the following items?
   i. the number of students in a class
   ii. the grade (a letter) attained by a student in the class
   iii. the average mark in a class
   iv. the distance between two points
   v. the population of a city
   vi. the weight of a postage stamp
   vii. the registration letter of a car

B. Programming Exercises

1. Write a program to read in four characters and to print them out, each one on a separate line, enclosed in single quotation marks.
2. Write a program which prompts the user to enter two integer values and a float value and then prints out the three numbers that are entered with a suitable message.

Collection of Questions asked in Previous Exams PU

1. What will be the output of following program?

i. **[Oct. 2009 – 2M]**

```
#include<iostream.h>
void main()
{
    int a=10;
    while(1)
    {
        switch(a)
        {
            case 10: cout<<a++;
            case 11: cout<<a--;
            case 12: cout<<a++;
        }
    }
}
```

ii. **[Apr. 2010 – 2M]**

```
#include<iostream.h>
#include<conio.h>
void main()
{
  char s[] = "CLASS";
  int i;
  for(i =0; s[i]; i++)
  cout <<"\n"<< s[i]<<*(s + i)<<*(i + s)<< i[s];
}
```

iii. **[Oct. 2010 – 2M]**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    char s[] = "OBJECT";
    int i;
    for(i=0; s[i]; i++)
        cout<<"\n"<<s[i++]<<*(s+i++)<<*(i+s)<<i[s];
}
```

iv. **[Oct. 2011 – 2M]**

```
void main()
{
  int a, *pa, &ra;
  pa = &a;
  ra = a;
  cout<<"a="<<a<<"*pa="<<*pa<<"ra="<<ra;
}
```

2. Explain new, delete and scope resolution operator. **[Oct. 2010 – 5M]**

4 Functions In C++

1. Introduction

Functions are a basic building block for writing C/C++ programs. Breaking up a program into separate functions, each of which performs a particular task, makes it easier to develop and debug a program.

1.1 Advantages of using functions

There are several advantages of using functions in program development.

i. Functions allow for breaking down the program into discrete units.

ii. Programs that use functions are easier to design, program, debug and maintain.

iii. It is possible to perform separate compilation of functions.

iv. Functions can receive data via arguments and can return a value.

v. Functions have local variables plus have access to global variables.

vi. A well-written function may be reused in multiple programs.

vii. Different programmers working on one large project can divide the workload by writing different functions.

1.2 Declaring a Function

A function is declared with a prototype. A function prototype consists of the return type, a function name and a parameter list. The function prototype is also called the function declaration.

Functions are declared similar to variables, but they enclose their arguments in parenthesis (even if there are no arguments, the parenthesis must be specified).

Syntax

```
return_type function_name(list of parameters);
```

For example

```
1.  int sum(int to); /* Declaration of sum with one argument */
2.  int bar();        /* Declaration of bar with no arguments */
3.  void foo(int ix, int jx);/*Declaration of foo with two arguments*/
4.  void max(void); /* Declaration of max with no arguments */
```

1.3 Defining a Function

The function definition consists of the prototype and a function body, which is a block of code enclosed in parenthesis.

A declaration or prototype is a way of telling the compiler the data types of any return value and of any parameters, so it can perform error checking. The definition creates the actual function in memory.

```
return_type function_name(list of parameters)
{ local_variable declarations;
   statements;
}
```

The parameter list of a function may have parameters of any data type and may have from no to many parameters.

Variables can be declared within a function. These variables are local variables and can only be used within the function.

Statement is the function's body. It can be a single instruction or a block of instructions. Statements perform the task of the function, which can also include calls to other C++ functions.

The return statement can be used to return a single value from a function. The return statement is optional.

1.4 Returning Values

The return statement allows a function to return a value of the stated data type. This statement immediately pushes a value onto the return stack and causes control to move to the ending curly brace, }, of the function, which returns control back to the calling function. Without a return statement, a function implicitly returns a value of zero for the data type for which the function was typed. The general form of the return statement is:

Syntax

```
return; or return value; or return(value);
```

For example

```
int sum(int to)
{   int ix, ret;
    ret = 0;
    for(ix = 0; ix < to; ix = ix + 1)
      ret = ret + ix;
      return ret;       /* return function's value */
} /* sum */ }
```

1.5 Calling a Function

Function which does not return a value can be called by simply writing the function name and giving it arguments (if any). And if the function returns a value, it can be used like any expression.

For example

1. `display_message();`
2. `display_value(x);`
3. `ncr  =  fact(n) /(fact(r)  *  fact(n-r));`
4. `greatest = max(c, max(a,b));`

1.6 Some Important Features of Functions

i. The structure of a function definition is like the structure of "main()", with its own list of variable declarations and program statements.

ii. A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.

iii. A function has to be declared in a function declaration at the top of the program, just after any global constant declarations, and before it can be called by "main()" or in other function definitions.

iv. Function declarations are a bit like variable declarations - they specify which type the function will return.

v. A function may have more than one "return" statement, in which case the function definition will end execution as soon as the first "return" is reached. *For example*

```
double absolute_value(double number)
{ if(number >= 0)
    return number;
  else
    return 0 - number;
}
```

1.7 How a Function Works?

A C++ program does not execute the statements in a function until the function is **invoked** or called. When the function is called, control passes to the function and returns back to the calling part after the execution of function is over.

The calling program can send information to the functions in the form of argument. An argument stores data needed by the function to perform its task. Functions can send back information to the program in the form of a return value.

Following program explains the working of function:

```
#include<iostream>
using namespace std;
int addition(int a, int b)   //declaration of a function
{ int r;
  r=a+b;
  return(r);
}
int main()
{ int z;
  z = addition(5,3);    //calling a function
  cout << "The result is " << z;
  return 0;
 }
```

Output: 8

▶ Explanation

We know that every C++ program always begins its execution from the *main* function. So we will begin from main.

In the main function we have, first declared the variable z of type int. Then a call to a function named addition() is made. If we pay attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself in the code lines above:

```
int addition(int  a, int  b);
                  ↑      ↑
z = addition(     5   ,  3 );
```

The parameters have a clear correspondence. Within the main function we called the addition() passing two values: 5 and 3 that correspond to the int a and int b parameters declared for the function addition.

At the moment at which the function is called from main, control is lost by main and passed to function addition. The value of both parameters passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares a new variable (int r;), and by means of the expression r=a+b;, it assigns to r the result of a plus b. Because the passed parameters for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

Finalizes function addition, and returns the control back to the function that called it (main) following the program from the same point at which it was interrupted by the call to addition. But additionally, return was called with the content of variable r (return (r);), which at that moment was 8, so this value is said to be returned by the function.

```
int addition(int  a, int  b);
↓ 8
z = addition( 5  ,  3 );
```

The value returned by a function is the value given to the function when it is evaluated. Therefore, z will store the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in *main* is:

```
cout << "The result is " << z;
```

That, as you may already suppose, produces the printing of the result on the screen.

Consider another example of functions:

```
// function example
#include<iostream>
using namespace std;
int subtraction(int a, int b)
{ int r;
  r=a-b;
  return (r);
}
int main()
{ int x=5, y=3, z;
  z = subtraction(7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction(7, 2) << '\n';
  cout << "The third result is " << subtraction(x, y) << '\n';
  z= 4 + subtraction(x, y);
  cout << "The fourth result is " << z << '\n';
  return 0;
 }
```

Output

The first result is 5

The second result is 5

The third result is 2

The fourth result is 6

In this case we have created the function subtraction. The only thing that this function does is to subtract passed parameters and to return the result.

Nevertheless, if we examine the function main( ) we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to understand well these examples you must consider once again that a call to a function could be perfectly replaced by its return value.

For example: The first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction(7,2);
cout << "The first result is " << z;
```

If we replace the function call by its result (that is *5*), we would have:

```
z = 5;
cout << "The first result is " << z;
```

As well as,

```
cout << "The second result is " << subtraction(7, 2);
```

has the same result as the previous call, but in this case we made the call to subtraction directly as a parameter for cout. Simply imagine that we had written:

```
cout << "The second result is " << 5;
```

since *5* is the result of *subtraction (7,2)*.

In the case of, `cout << "The third result is " << subtraction(x,y);`

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to the function subtraction are the values of x and y, that are 5 and 3 respectively, giving 2 as result. The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction(x, y);
```

we have put: z = subtraction(x, y) + 4;

with exactly the same result. Notice that the semicolon sign (;) goes at the end of the whole expression. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its result:

```
z = 4 + 2;
z = 2 + 4;
```

1.8 Functions with no types - The use of *void*

If you remember the syntax of a function declaration:

```
return_type function_name(list of parameters);
```

You will see that it is obligatory that this declaration begin with a return_type, that is the type of the data that will be returned by the function with the return instruction. But what if we want to return no value? Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value, moreover, we do not need it to receive any parameters. For these cases, the void type is used.

```
// void function example
#include<iostream>
using namespace std;
void dummyfunction(void)
{  cout << "I'm a function!"; }
int main()
{ dummyfunction();
  return 0;
}
```

Output: I'm a function!

Although in C++ it is not necessary to specify *void*, its use is considered suitable to signify that it is a function without parameters or arguments and not something else.

We know that calling a function includes specifying its name and enclosing the arguments between parenthesis. The non-existence of arguments does not exempt us from the obligation to use parenthesis. For that reason the call to dummy function is

```
dummyfunction();
```

This clearly indicates that it is a call to a function and not the name of a variable or anything else.

2. Passing Information - Parameters

Parameters are a means of communication between the calling function and the called function. Parameters are of two types depending upon where they are used.

i. **Actual Parameters:** These are the variables or values used in the function call (or reference point). Actual parameter can be variables, constants, literals, expressions.

ii. **Formal Parameters (Dummy parameters):** These are the variables used in the function header in the function definition. Formal parameters have to be variables.

Methods of Passing Parameters

Parameters to a function may either be pass by value or pass by reference.

> **Methods of Passing Parameters**
> i. Pass by value
> ii. Pass by reference
> iii. Return by reference

i. Pass by Value

If parameters are passed by value, only a copy of the variable has been passed to the function. Any changes to the value will not be reflected back to the calling function.

For example: Suppose that we called our first function *addition* using the following code:

```
int x=5, y=3, z=2;
z = addition(x, y);
```

What we did in this case was to call function *addition* passing the values of *x* and *y*, that means *5* and *3* respectively, not the variables themselves.

```
int addition(int a, int b);
                 ↑5     ↑3
z = addition( x   ,  y );
```

This way, when function *addition* is being called the value of its variables *a* and *b* become *5* and *3* respectively, but any modification of *a* or *b* within the function addition will not affect the values of x and y outside it, because variables *x* and *y* were not passed themselves to the function, only their values.

But there might be some cases where you need to alter the values of the original variables. For that purpose we have to use *arguments passed by reference*.

ii. Pass by Reference

A function that uses call-by-reference arguments reflects changes to the values of the arguments to the calling function. In traditional C, this is achieved by passing the address of the variable, and using pointers to de-reference the arguments. C++ introduces a reference operator (&), that allows true call-by-reference functions, eliminating the need to dereference arguments passed to the function.

The reference operator is placed before the variable name of arguments in the parameter list of the function. The function can then be called without passing the address, and without dereferencing within the function. The following example contains a function to swap two values. As the values require changing in the calling function, the values are passed by reference.

```
// passing parameters by reference
#include<iostream>
using namespace std;
void duplicate(int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}
int main()
{
  int x=1, y=3, z=7;
  duplicate(x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Output: x=2, y=6, z=14

The first thing that should call your attention is that in the declaration of *duplicate* the type of each argument was followed by an *ampersand* sign (&), that serves to specify that the variable has to be passed *by reference* instead of *by value*, as usual.

When passing a variable *by reference* we are passing the variable itself and any modification that we do to that parameter within the function will have effect in the passed variable outside it.

```
void duplicate(int& a, int& b, int& c);
                ↕x      ↕y      ↕z
      duplicate( x  ,   y   ,   z );
```

To express it another way, we have associated *a, b* and *c* with the parameters used when calling the function (*x, y* and *z*) and any change that we do on *a* within the function will affect the value of *x* outside. Any change that we do on *b* will affect *y* and the same with *c* and *z*.
That is why our program's output, that shows the values stored in *x, y* and *z* after the call to *duplicate,* shows the values of the three variables of *main* doubled.
If when declaring the following function:
`void duplicate(int& a, int& b, int& c)`
We had declared it thus:
`void duplicate(int a, int b, int c)`
that is, without the *ampersand* (&) signs, we would have not passed the variables *by reference*, but their values, and therefore, the output on screen for our program would have been the values of *x, y* and *z* without having been modified. 4
This type of declaration "*by reference*" using the *ampersand* (&) sign is exclusive of C++.
In C language we had to use pointers to do something equivalent. Passing by reference is an effective way to allow a function to return more than one single value. *For example*: Here is a program that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include<iostream>
using namespace std;
void prevnext(int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}
int main()
{
  int x=100, y, z;
  prevnext(x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Output: Previous = 99, Next = 101

iii. Return by Reference

A function can also return a reference. Consider the following function:
`int & minx(int &x, int &y)`

```
{  if(x < y)
      return x;
   else
      return y;
}
```

Since the return type of min() is int &, the function returns reference to x or y (and not the values). Then a function call such as min(a, b) will yield a reference to either a and b depending on their values. This means that this function call can appear on the left hand side of an assignment statement. That is, the statement

```
min(a, b) = -1;
```

is legal and assigns –1 to a if it is smaller, otherwise –1 to b.

3. Default Arguments

When declaring a function we can specify a default value for each parameter. This value will be used if that parameter is left blank when calling to the function. To do that we simply have to assign a value to the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is stepped on and the passed value is used.

```
// default values in functions
#include<iostream>
using namespace std;
int divide(int a, int b=2)
{ int r;
  r=a/b;
  return (r); }
int main()
{  cout << divide(12);
   cout << endl;
   cout << divide(20,4);
   return 0;
 }
```

Output: 6
5

As we can see in the body of the program there are two calls to the function *divide*. In the first one: `divide(12);`

We have only specified one argument, but the function *divide* allows up to two. So the function *divide* has assumed that the second parameter is *2* since that is what we have specified (notice the function declaration, which finishes with *int b=2*). Therefore the result of this function call is *6 (12/2)*.

In the second call: `divide(20,4)`

There are two parameters, so the default assignation (int b=2) is stepped on by the passed parameter, that is *4*, making the result equal to *5 (20/4)*.

A default argument is checked for type at the time of declaration and evaluated at the time of call. Only the trailing arguments can have default values.

It is important to note that we must add defaults from right to left. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are

```
void func1(int first=0, int second=0, int third=0);    // valid
void func2(int first, int second=0, int third=0);      // valid
void func3(int first, int second, third=0);            // valid
void func4(int first, int second, int third);          // valid
void func5(int first = 0, int second)                  // invalid
void func6(int first = 0, int second, int third = 0); // invalid
```

The default arguments are used to add new parameters to the existing functions and also used to combine similar functions into one. Default arguments are useful in situations where some arguments always have the same values.

4. Constant Arguments

The keyword *const* can also be used as a guarantee that a function will not modify a value that is passed in. This is really only useful for references and pointers (and not things passed by value), though there's nothing syntactically to prevent the use of const for arguments passed by value.

Consider the following functions:

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier const tells the compiler that the function should not modify the argument. When this condition become false the compiler will generate an error.

5. Function Overloading

C++ allows both functions and operators to be overloaded. An overloaded function, is a function with the same name as another function, but with different parameter types, that means you can give the same name to more than one function if they have either a different number of arguments or different types in their arguments. This promotes programming flexibility. Overloaded functions allow the following:

i. Use the same function name

ii. Carry out operations using different data types

For example: Suppose we were required to find the product of two numbers, either of which may be of type int or double. All we would have to do is write four functions, each with the same name and define to ensure that we get the correct result, regardless of the types of the arguments we use in the function call. The appropriate function is selected depending on the data types of the parameters.

The following is the function prototypes:

```
int    product(int x, int y);
double product(int x, double y);
double product(double x, int y);
double product(double x, double y);
```

The compiler will then choose the appropriate function. From a readability point of view, function overloading should only be used when the functionality is the same.

Overloaded functions must adhere to the following two rules:

a. The compiler does not use the return type of the function to distinguish between function instances.
b. The argument list of each of the function instances must be different.

The following example overloads a divide function for integers, and floating point numbers.

```
// overloaded function
#include<iostream>
using namespace std;
int divide(int a, int b)
{
  return (a/b);
}
float divide(float a, float b)
{
  return (a/b);
}
int main()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << divide(x, y);
  cout << "\n";
  cout << divide(n, m);
  cout << "\n";
  return 0;
 }
```

Output

2
2.5

In this case we have defined two functions with the same name, but one of them accepts two arguments of type *int* and the other accepts them of type *float*. The compiler knows which one to call in each case by examining the types when the function is called. If it is called with two *int*s as arguments it calls to the function that has two *int* arguments in the prototype and if it is called with two *float*s it will call to the one which has two *float*s in its prototype. For simplicity I have included the same code within both functions, but this is not compulsory. You can make two functions with the same name but with completely different behaviors. Following is another example which overloads a swap function for integers, and floating point numbers.

```
#include<iostream>
using namespace std;
// Overload the swap function for int and float
void swap(int &a, int &b);
void swap(float &a, float &b);
int main()
{
    int i1 = 3, i2 = 5;
    float f1 = 3.14159f, f2 = 1.23f;
    cout << "Integers before swap:" << endl;
    cout << i1 << ", " << i2 << endl;
    swap(i1, i2);
    cout << "Integers after swap:" << endl;
    cout << i1 << ", " << i2 << endl;
    cout << "Floating points before swap:" << endl;
    cout << f1 << ", " << f2 << endl;
    swap(f1, f2);
    cout << "Floating points after swap:" << endl;
    cout << f1 << ", " << f2 << endl;
    return 0; }
// An integer swap function
void swap(int &a, int &b)
{
     int temp = a;
     a = b;
     b = temp;
     return;}
// An overloaded float swap function
void swap(float &a, float &b)
{
     float temp = a;
     a = b;
     b = temp;
     return;
 }
```

Output

Integers before swap:

3, 5

Integers after swap:

5, 3

Floating points before swap:

3.14159f, 1.23f

Floating points after swap:

1.23f, 3.14159f

6. Inline Functions

In the C language, the macro substitution directive (#define) allows us to define a macro whose value is substituted in place of the macro name in the program. A macro can also have parameters. The following example defines an argumented macro called SQR.

```
# define SQR(x)    (x)*(x)
```

The advantage of using a macro is that whenever the name SQR appears, it will be replaced by its value. Hence, the execution will be faster. This is ideal when the code is small. However, macro's are not compiled which can lead to errors. Instead of a macro, we can define a function called sqr which will perform the same task as above. However, function calls and returns add to program overheads.

C++ provides the facility of making a function **inline**. Any function declared using the keyword inline will be expanded inline, i.e., the function code will be replaced at the point where the function is called. The *inline* directive can be included before a function declaration to specify that the function must be compiled as code at the same point where it is called. To make an inline function, the keyword, "inline" precedes the function prototype and function definition. This is equivalent to declaring a macro. Its advantage is only appreciated in very short functions, in which the resulting code from compiling the program may be faster if the overhead of calling a function (stacking of arguments) is avoided.

The format for its declaration is:

```
inline type name(arguments ...) {instructions ...}
```

and the call is just like the call to any other function. It is not necessary to include the *inline* keyword before each call, only in the declaration.

Functions containing the following would not be suitable for an inline function:

a. Static variables
b. Iteration constructs
c. A switch statement
d. Arrays
e. Recursive calls to istelf.

The following example uses an inline function to determine if a given year is a leap year. If the request is successful, the function definition will be expanded in the main part of the program, eliminating the need for a function call.

```
#include<iostream>
using namespace std;
// Prototype declaration
inline int leap(int year);
int main()
{   int year;
    cout << "Enter a year:";
    cin >> year;
    if(leap(year))
      cout << "The year " << year << " is a leap year" << endl;
    else
     cout << "The year " << year << " is not a leap year" << endl;
    return 0;}
// Definition of leap
```

```
inline int leap(int year)
{  if(((year % 4) == 0) && (((year % 100) != 0)||((year%400)==0)))
return 1;
return 0;
 }
```

Difference between Inline Function and Macro [Oct. 2011 *5M*]

	Inline function	Macro
i.	Inline function is the optimization technique used by the compilers. One can simply prefix inline keyword to function prototype to make a function inline. Inline function instructs compiler to insert complete body of the function wherever that function got used in code.	A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.
ii.	Inline functions follow all the protocols of type safety enforced on normal functions.	Since we don't specify the type with macros, type safety is not enforced.
iii.	Expressions passed as arguments to inline functions are evaluated once.	In some cases, expressions passed as arguments to macros can be evaluated more than once.
iv.	Inline functions are parsed by the compiler directly instead of the preprocessor.	The C++ preprocessor implements macros by using simple text replacement.
v.	Debugging inline function is as easy as debugging a normal function.	Debugging macros is difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself.

7. Recursive Functions

Recursivity is the property that functions call themselves. It is sometimes called circular definition. It is useful for tasks such as some sorting methods or to calculate the factorial of a number. *For example*: To obtain the factorial of a number (n) its mathematical formula is:

```
n! = n * (n-1) * (n-2) * (n-3) ..., * 1
```

more concretely, 5! (factorial of 5) would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

and a recursive function to do that could be this:

```
// factorial calculator
#include<iostream>
using namespace std;
long factorial(long a)
{  if (a > 1)
   return (a * factorial(a-1));
  else
```

```
   return (1); }
int main()
{ long l;
  cout << "Enter a number: ";
  cin >> l;
  cout << l << "!" << " = " << factorial(l);
  return 0;
 }
```

Output

Type a number: 9

9! = 362880

When factorial() is called with an argument of 1, the function returns 1, otherwise it returns the product of factorial (a–1)*a. To evaluate this expression, factorial() is called with a–1. This happens until a equals 1 and the calls to the function begin returning. Notice how in function *factorial* we included a call to itself, but only if the argument is greater than *1*, since otherwise the function would perform *an infinite recursive loop* in which once it arrived at *0* it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime). This function has a limitation because of the *data type* used in its design (long) for more simplicity. In a standard system, the type *long* would not allow storing factorials greater than *12!*. When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

Solved Programs

1. Write a function "print_pyramid(...)" which takes a single integer argument "height" and displays a "pyramid" of this height made up of "*" characters on the screen. Test the function with a simple "driver" program, which should be able to reproduce the following example output: This program prints a 'pyramid' shape of a specified height on the screen.

How high would you like the pyramid?: 37

Pick another height (must be between 1 and 30): 6

```
     **
    ****
   ******
  ********
 **********
************
```

Solution

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

```
void print_pyramid(int height);
int main()
{ int pyramid_height;
  cout << "This program prints a 'pyramid' shape of";
  cout << "a specified height on the screen.\n\n";
  /* input with check using a "while" loop */
  cout << "How high would you like the pyramid?: ";
  cin  >> pyramid_height;
  while(pyramid_height > 30 || pyramid_height < 1)
  { cout << "Pick another height (must be between 1 and 30): ";
    cin >> pyramid_height;
  }
  print_pyramid(pyramid_height);
  return 0;
}
void print_pyramid(int height)
{ int line;
  int const MARGIN = 10;
  cout << "\n\n";
  for(line = 1 ; line <= height ; line++)
  { int count;
    int total_no_of_spaces = MARGIN + height - line;
    for(count = 1 ; count <= total_no_of_spaces ; count++)
      cout << ' ';
    for(count = 1 ; count <= line * 2 ; count++)
      cout << '*';
      cout << '\n'; }
      cout << "\n\n";
}
```

2. Write a program to calculate x^y using function.

Solution

```
#include<iostream>
#include<cstdlib>
using namespace std;
float raised_to_power(float number, int power);
int main()
{ float my_number;
  int my_power;
  cout << "This program raises a number x to a non-negative integer
powern\n";
  cout << "Please enter a number: ";
  cin  >> my_number;
  cout << "To what power would you like it raised? ";
  cin  >> my_power;
  cout << my_number << "raised to the power" << my_power;
  cout << " is " << raised_to_power(my_number, my_power);
  return 0; }
float raised_to_power(float number, int power)
{ if(power < 0)
  { cout << "\nError - can't raise to a negative power\n";
    exit(1); }
```

```
    else if(power == 0)
      return (1.0);
    else
      return (number * raised_to_power(number, power - 1));
}
```

3. Write a program to calculate the GCD of two positive numbers.

Solution

```
#include<iostream>
#include<cstdlib>
using namespace std;
int greatest_common_divisor(int first_number, int second_number);
int main()
{  int first_positive_number, second_positive_number;
   cout << "Enter a positive integer:";
   cin  >> first_positive_number;
   cout << "Enter another positive integer: ";
   cin  >> second_positive_number;
   cout << "\nThe greatest common divisor of " << first_positive_number;
   cout << " and " << second_positive_number << " is ";
   cout<<greatest_common_divisor(int first_positive_number,
      second_positive_number);
   return 0; }
int greatest_common_divisor(int first_number, int second_number)
{  if(first_number < 1 || second_number < 1)
   {  cout << "Error - non-positive argument to
      'greatest_common_divisor'\n";
      exit(1);  }
   else if(first_number == second_number)
      return first_number;
   else if(first_number < second_number)
   return greatest_common_divisor(first_number,(second_number -
      first_number));
   else
   return greatest_common_divisor(second_number, (first_number -
      second_number));
}
```

4. What will be the output of following program?

PU
Apr. 2010 – *2M*

```
#include<iostream.h>
#define SQUARE(x) x*x
inline float square(float y)
{
      return y*y;
}
int main()
{
      float a = 0.5, b = 0.5, c, d;
      c = SQUARE(++a);
      d = square(++b);
      cout << c << endl << d;
      return 0;
}
```

Solution

Output of the given code is

6.25

2.25

Explanation: We are passing the value of variable 'a' to the macro SQUARE(x) x*x. When this macro is executed, the final value of 'a' would be 2.5, (because of ++a). So the output of the macro SQUARE(x) x*x would be 6.25.

When the function inline float square (float y) is executed, the value of variable 'b' that it receives is 1.5 because the value of 'b' would increment by 1 (i.e., now b=1.5) and then it is passed to the square function. So square function would return 2.5.

5. Write a program to demonstrate default arguments and constant arguments for a function.

1
PU
Apr. 2010 – *5M*

Solution

Default Argument

```
#include<iostream>
void f(int x=0, int y=100);//prototype
int main()
{
  f(1,2);//function call without the need of any default
  //argument
  f(10);//function call with the need of one default
  //argument(y=100)
    f();//function call with the need of two default
  //arguments(x=0,y=100)
  return 0;
}
void f(int x, int y)
{
  cout <<"x:"<<x<<",y:"<<y<<"\n";
}
```

Constant Argument

```
#include<iostream>
float Perimeter(const float I, const float w)
//ConstantArguments
{
   double p;
   p=2*(l+w);
   return p;
}
int main()
{
```

```
  float length, width;
  cout<< "Rectangle dimensions.\n";
  cout<< "Enter the length: ";
  cin >> length;
  cout<< "Enter the width: ";
  cin >> width;
  cout<< "\nThe perimeter of the rectangle is: "
      << Perimeter(length, width) << "\n\n";
  return 0;
}
```

6. Write a program to demonstrate overloading of constructor.

PU Oct. 2010 – *5M*

Solution

```
/*Program to demonstrate overloading of constructor */
#include<iostream.h>
class MyClass
{
  public:
    int x;
    int y;

  //Overload the default constructor.
  MyClass()
  {
    x=y=0;
  }

  //Constructor with one parameter.
  MyClass(int i)
  {
    x=y=i;
  }

  // Constructor with two parameters.
  MyClass(int i, int j)
  {
    x=i;y=j;
  }
};

int main()
{
  MyClass t;           // invoke default constructor
  MyClass t1(5);       // use MyClass(int)
  MyClass t2(9, 10); // use MyClass(int, int)
  cout<<"t.x:"<< t.x<<",t.y:"<<t.y<< "\n";
  cout<<"t1.x:"<< t1.x<<",t1.y:"<<t1.y<< "\n";
  cout<<"t2.x:"<< t2.x<<",t2.y:"<<t2.y<< "\n";
  return 0;
}
```

EXERCISES

A. Review Questions

1. What is function? List out advantages and disadvantages of using functions in C++?
2. How a function is declared in C++?
3. What is meant by call by reference and call by value?
4. What is the purpose of return statement?
5. Explain how a static member is defined and declared in C++.
6. What is a static class member?
7. What is a recursive function?

B. Programming Exercises

1. Write a function in C++ to find the sum of the following series:
   a. sum = 1+2+3+. . .+n
   b. sum = 1+3+5+. . .+n
   c. sum = $1^2 + 2^2 + 3^2 + \ldots + n^2$
   d. sum = $1^3+3^3+5^3+\ldots+n^3$
2. Write a function in C++ to generate a Fibonacci series of 'n' numbers, where n is defined by a programmer. (The series should be:1 1 2 3 5 8 13 21 32 . . .).
3. Write a function in C++ to generate the following pyramid of numbers.

```
        0
      1 0 1
    2 1 0 1 2
  3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4
```

4. Write an object-oriented program in C++ to read any five real numbers and print the average using a static member class.

Collection of Questions asked in Previous Exams PU

1. What will be the output of the following program? **[Apr. 2010 – 2M]**

```
#include<iostream.h>
#define SQUARE(x) x*x
inline float square(float y)
{
    return y*y;
}
int main()
{
    float a = 0.5, b = 0.5, c, d;
    c = SQUARE(++a);
    d = square(++b);
    cout << c << endl << d;
    return 0;
}
```

2. Write a program to demonstrate default arguments and constant arguments for a function. **[Apr. 2010 – 5M]**
3. Write a program to demonstrate overloading of constructor. **[Oct. 2010 – 5M]**
4. Difference between inline function and macro. **[Oct. 2011 – 5M]**

Innovation Throughout
VISION

5 Classes And Objects

I. Introduction

The word "class" is the most important feature of C++. It's significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type.

A class is a logical method to organize data and functions in the same structure. They are declared using keyword **class**, whose functionality is similar to that of the C keyword **struct**, but with the possibility of including functions as members, instead of only data.

Structures and Classes

C++ language has extended the role of the structure, making it an alternative way to specify a class. Since both class and struct have almost the same functionality in C++, struct is usually used for data-only structures and class for classes that have procedures and member functions.

A structure contains one or more data items called members, which are grouped together as a single unit. On the other hand, a class is similar to a structure data type but consists of not only data elements but also functions, which are operated on the data elements. Secondly, in a structure, all the elements are public by default while in a class all the elements are private by default. In all other respects, structures and classes are equivalent.

The presence of two virtually equivalent keywords **struct** and **class** can be justified for various reasons. Firstly, to increase the capabilities of a structure by allowing them to include member

functions. Secondly, to make porting easier between C and C++. Finally, providing two different equivalent keywords allows the definition of a class to be free to evolve. For C++ to remain compatible with C, the definition of struct must always be tied to its C definition.

2. Class

A class is a user defined data type, which binds the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. The internal data of a class is called member data (or data member) and the functions are called member functions. Only the member functions can have access to the private data members and private functions. However, the public members (both data and functions) can be accessed from outside of the class. The variables of a class are called objects or instances of class.

The general form of a class declaration is:

```
class class_name {
   permission_label_1:
     member1;
   permission_label_2:
     member2;
   . . .
}object_list;
```

The above class can also be created using the "**struct**" keyword as shown below.

```
struct class_name {
   permission_label_1:
     member1;
   permission_label_2:
     member2;
   . . .
}object_list;
```

where class_name is a name for the class (user defined type) and the optional field object_list is one, or several, valid object identifiers. The object_list is optional. The body of the declaration can contain members, that can be either data or function declarations, and optionally permission labels or access specifiers, that can be any of these three keywords: private:, public: or protected:. They make reference to the permission, which the following members acquire:

i. **Private:** The members, which are declared in the private section, can be accessed only from within the class, i.e., by the member functions of their same class and friends of this class. The member functions and friends of this class can always read or write private data members. The private data member is not accessible to the out of the class.

ii. **Protected:** The members in the protected section can be accessed by the member functions and friends of this class, and also from member functions and friends derived from this class. It is not accessible to the outside world (out of the class).

iii. **Public:** The members declared in the public section can be accessed by any function in the outside world (out of the class). Public data members can always be read and written from outside this class. A member function can be inline, which means that member function can be defined within the body of the class constant.

The default member access of a class created using **class** keyword is **private** and using **struct** keyword is **public.**

For example

```
class CRectangle
{
  int x, y;
  public:
    void set_values(int,int);
    int area(void);
} rect;
```

In the above example, *CRectangle* is the class name and *rect* is an object of this class (type). This class contains four members: two variables of type *int* (*x* and *y*) in the *private* section (because private is the default permission) and two functions in the *public* section: *set_values()* and *area()*, of which we have only included the prototype.

The difference between class name and object name: In the above example, *CRectangle* is the class name (i.e., the user-defined type), whereas *rect* is an object of type CRectangle. Is the same difference that *int* and *a* have in the following declaration:

`int a;` int is the class name (type) and a is the object name (variable).

In general, all data members of a class should be made private to that class so as to achieve encapsulation. But, there may be situations where a variable would need to be declared public. The protected access specifier is needed only when inheritance is involved.

2.1 Creating Object

Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). *For example*

```
CRectangle rect;       //memory for rect is created.
```

The above statement creates a variable *rect* of type *CRectangle.* In C++, the class variables are known as objects therefore rect is called an object of type CRectangle.

We may declare more than one object in one statement. *For example*

```
CRectangle rect, rectb, rectc;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure, provides only a template and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace.

For example

```
class CRectangle
{    - - -
     - - -
} rect, rectb, rectc;
```

The above definition would create the objects of type rect, rectb and rectc of type CRectangle.

2.2 Accessing Class Members

In the body of the program we can refer to any of the public members of the object as if they were normal functions or variables, just by putting the object's name followed by the dot operator and then the class member (like we did with C structs).

Syntax

```
Object_name.function_name(actual arguments);
```

For example

```
class abc
{  int p, q;
   public:
      int r;
};
  . . .
  . . .
abc s;
s.p = 0;         // error, p is private
s.r = 20;        //ok, r is public
  . . .
  . . .
```

In the above example, the statement s.p = 0 is illegal because p is the private member and they can only be accessed by the members of that same class (for details see next section). In the next statement s.r = 20; is valid since r is the public member and public members can be accessed by the objects directly.

3. Member Functions

Member functions are the functions that are designed to implement the operations allowed on the data type represented by a class. To declare a member function, place its prototype in the body of the class. The definition of the function can be inside the class or outside the class but in the same file or in a separate file.

Member functions can be defined in two places:

Place in which Member Functions can be defined
i. Outside the class definition
ii. Inside the class definition

i. Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions except that a member function incorporates a membership 'identity label' in the header. This label tells the compiler which *class* the function belongs to.

The general form of a member function definition is:

```
return-type class-name :: function-name(argument declaration)
{
   function body;
}
```

The membership label *class-name ::* tells the compiler that the function *function-name* belongs to the class *class-name.* That is, the scope of the function is restricted to the *class-name* specified in the header line. The symbol:: is called the scope resolution operator.

Characteristics of Member Functions

The member functions have some special characteristics that are often used in the program development. The characteristics are as follows:

a. The same function name can be used by different classes. The 'membership label' will resolve their scope.
b. Member functions can access the private data of the class. A nonmember function cannot do so (except friend function, discussed later).
c. A member function can call another member function directly, without using the dot operator.

ii. Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

When a function is defined inside a class, it is treated as an inline function *(Refer 1.6 in chapter 4).* Therefore all the restrictions and limitations that apply to an inline functions are also applicable here. Normally, only small functions are defined inside the class definition.

A Simple Class Program

Program for class

```
// class example
#include<iostream>
using namespace std;
class CRectangle {
  int x, y;                        //  private by default
  public:
    void set_values(int,int);    //  Prototype declaration
    int area(void) {return (x*y);} //   function defined inside class
                              //i.e., inline function
};
// Member Function Definition(Function defined outside of class)
void CRectangle::set_values(int a, int b)
{
  x = a;                           //  private variables
  y = b;                           //  directly used
}
// Main Program
int main() {
  CRectangle rect, rectb;      //  create objects rect, rectb
  rect.set_values(3,4);        //  call member function
  rectb.set_values(5,6);       //  call member function
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
 }
```

This program features the class CRectangle. This class contains two private variables and two public functions. The member function set_values() is defined outside the class with the help of scope resolution operator (::). This operator specifies the class to which the member being declared belongs, granting exactly the same scope properties as if it was directly defined within the class. *For example*: In the function *set_values( )*, we have referred to variables x and y, that are members of class CRectangle and that are only visible inside it and its members(since they are *private*).

The only difference between defining a class member function completely within its class and to include only the prototype is that in the first case the function will automatically be considered *inline* by the compiler, while in the second it will be a normal (not-inline) class member function.

The use of statement such as `x = a;`

In the function definition of *set_values()* shows that the member functions can have direct access to private data items.

The member function *area()* has been defined inside the class and therefore behaves like an inline function. This function displays the values of private variables x and y.

The program creates two objects rect, rectb. Note that the call to *rect.area()* does not give the same result as the call to *rectb.area()*. This is because each object of class *CRectangle* has its own variables *x* and *y*, and its own functions *set_value()* and *area()*.

Output

rect area: 12

rectb area: 30

Note: The reason why we have made *x* and *y private* members (remember that if nothing else is said all members of a class defined with keyword *class* have *private* access) is because we have already defined a function to introduce those values in the object (*set_values()*) and therefore the rest of the program does not have a way to directly access them but in greater projects it may be very important that values should not be modified in an unexpected way (unexpected from the point of view of the object).

4. Making an Outside Function Inline

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition.

Example

```
class CRectangle
{ . . .
  . . .
  public:
    void set_values(int a, int b)     //declaration
};
inline void CRectangle :: set_values(int a, in b)    //definition
{ x = a;
  y = b;}
```

5. Nesting of Member Functions

Nesting of member functions means a member function of a class can be called by using its name inside another member function of the same class. We know that a member function of a class can be called only by an object of that class using a dot operator.

Following program illustrates this feature. Program for nesting of member functions.

```
#include<iostream>
using namespace std;
class min
{  int a,b;
   public :
        void getdata(void);
        void display(void);
        int smallest(void);
};
int min :: smallest(void)
{
   if(a<=b)
        return(a);
   else
        return(b);
}
void min ::getdata(void)
{  cout << "Enter the values of a and b "<<"\n";
   cin>>a>>b;
}
void smallest :: display(void)
{  cout << "The smallest value is = "<< smallest()
            << "\n";//calling member function.
}
int main()
{  min M;
   M getdata();
   M.display();
   return 0;
 }
```

The output of a program is

Enter the values of a and b

36 16

The smallest value is 16.

6. Private Member Function

Generally all data items are placed in a private section and all the functions in public section. But in some cases it may require to hide certain functions (i.e., private data) from the outside class. To handle this situation we can place such functions in the private section and such functions are called as private member functions.

A private member function can only be called by another function which is a member of its class. A private member function cannot be invoked by an object using the dot operator.

Consider a class defined below:

```
class vehicle
{  int vcode ;
   void read(void); //private member function.
   public:
        void update(void);
        void write(void);
};
```

If v1 is an object of vehicle class, then

```
v1.read();     //won't work; objects cannot access private members.
```

is illegal, However, read() function is a private member function so it can be called by a function which is a member of its class, i.e., by update() function to update the value of vcode.

```
void vehicle :: update(void)
{  read();//Simple call; no object is used.
}
```

7. Arrays within a Class

We can use array as a member variable within a class.

For example

```
const int size = 100; //provides value for array size
class xyz
{  int a[size];       //'a' is int type array
public:
   void setdata(void);
   void displaydata(void);
};
```

The array variable a[ ] declared as a private member of the class, can be used in the member functions, like any other array variable. Any operations can be performed on it. *For example*: In the above class definition, the member function setdata sets the values of the elements of the array a[ ], and displaydata() function displays the values. We can also use other member functions to perform any other operations on the array values.

8. Memory Allocation for Objects

Whenever the objects are declared, the memory space for the object is allocated and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member function, no separate space is allocated for member function when the objects are created. Only space for member variables is allocated separately for each object. Since member variables will hold different data values for different objects separate memory locations for the objects are essential. This is shown in the following *figure 5.1*.

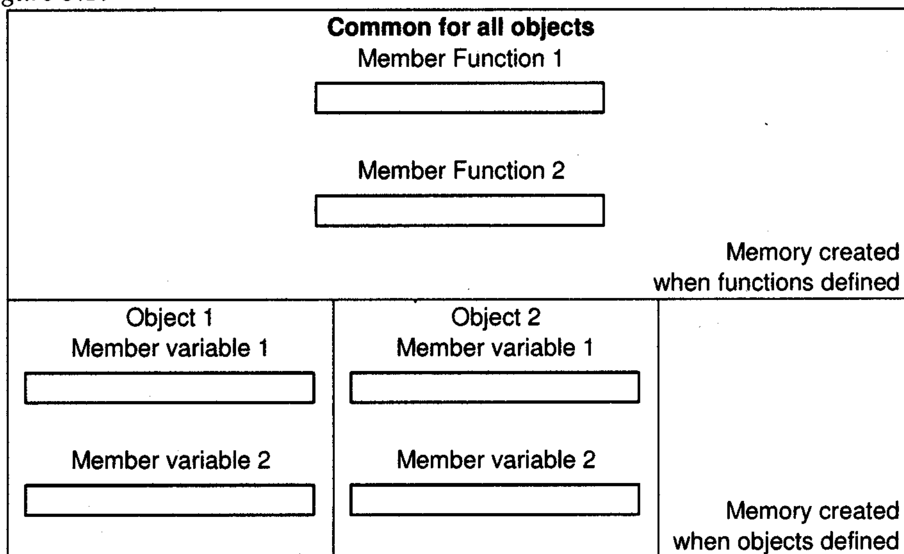


Figure 5.1 : Memory allocation for objects

9. Arrays of Objects

Any object, whether built-in or user defined, can be stored in an array. When you declare the array, you tell the compiler the type of object to store and the number of objects for which to allocate room. The compiler know how much room is needed for each object based on the class declaration. The class must have a default constructor *(for more details refer chapter 6)* that takes no arguments so that the objects can be created when the array is defined.

Consider the following class definition:

```
class  vehicle
{ int vcode;
  char vname[30];
  public:
     void getinfo(void);
     void disinfo(void);
};
```

The identifier *vehicle* is a user defined data type and can be used to create objects that relate to different categories of the vehicles.

Example:
```
Vehicle v1[3];
Vehicle v2[4];
```

The array v1 contains three objects namely, v1[0], v1[1] and v1[2] of type *vehicle* class. Similarly, the v2 array contains 4 objects namely v2[0], v2[1], v2[2] and v2[3].

Accessing member data in an array of objects is a two-step process. You identify the member of the array by using the index operator ([ ]), and then you add the member operator (.) to access the particular member variable.

For example: The statement `v1[i].disinfo();`

will display the data of the ith element of the array v1. That is, this statement requests the object v1[i] to invoke the member function disinfo().

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array v1 is represented in the figure given below. Note that only the space for data items of the object is created. Member functions are stored separately and will be used by all the objects.

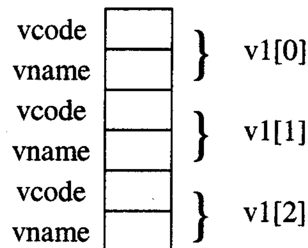


Figure 5.2 : Storage of data items of an object array

Program for Array of Objects

```
#include<iostream>
using namespace std;
class vehicle
{  int vcode;
   char vname[30];        //String as class member
   public:
      void getdata(void);
      void putdata(void);
};

void vehicle :: getdata(void)
{  cout << "Enter the Vehicle Code:";
   cin  >> vcode;
   cout << "Enter the Vehicle Name:";
   cin  >> vname;
}
void vehicle :: putdata(void)
```

```
{ cout << "Code:"<< vcode << "\n";
  cout << "Name:"<< vname << "\n";
}
const int size =3;
int main()
{ vehicle v1[size];       //Array of object
  for(int i=0; i<size; i++)
  { cout <<"\n Details of Vehicle:" << i+1 << "\n";
    v1[i].getdata();
  }
  cout << "\n";
  for(i=0; i<size; i++)
  { cout <<"\n Vehicle" << i+1 << "\n";
    v1[i].putdata();
  }
  return 0;
}
```

Input given to program

```
Details of vehicle1
Enter the Vehicle Code : 001
Enter the Vehicle Name : Scooty

Details of vehicle2
Enter the Vehicle Code : 002
Enter the Vehicle Name : Scooter

Details of vehicle3
Enter the Vehicle Code : 003
Enter the Vehicle Name : Hero Honda
```

Output of the program

```
Vehicle1
Code : 001
Name : Scooty

Vehicle2
Code : 002
Name : Scooter

Vehicle3
Code : 003
Name : Hero Honda
```

10. Objects as Function Arguments

An object may be used as a function argument like any other data types. This can be done in following two ways.

i. Pass-by-value, i.e., a copy of the entire object is passed to a function.

ii. Pass-by-reference, i.e., only the address of the object is transferred to the function.

In the first-method, a copy of the object is passed to the function, any changes made to the object inside the function, do not affect the object used to call the function.

In the second method, when an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by-referènce method is more efficient than the pass-by-value method since it requires to pass only the address of the object not the entire object.

Following program illustrates the use of objects as function arguments. If performs the addition of two complex numbers.

```
#include<iostream>
using namespace std;
class complex
{
  float real, imag;
public :
void getdata(float x, float y)
{
  real = x;
  imag = y;
}
void display(void);
{
  cout << real << "+i " <<imag<<"\n";
  void sum(complex, complex); // declaration with object as arguments
};
void complex:: sum(complex C1, complex C2)//C1, C2 are objects.
{
  real = C1.real+C2.real;
  imag = C1.imag +C2.imag;
}
int main()
{
  complex C1, C2, C3;
  C1.getdata(1,1);   // get C1
  C2.getdata(3,3); //get C2
  C3.sum(C1,C2);  // C3 = C1 + C2
  cout << "C1 = " ; C1.display(); // display C1
  cout << "C2 = " ; C2.display(); // display C2
  cout << "C3 = " ; C3.display(); // display C3
return 0;
 }
```

Note: Since the member function sum( ) is invoked by the object C3, with the objects C1 and C2 as arguments, it can directly access the real and imag variables of C3. But the members of C1 and C2 can be accessed only by using the dot operator like C1.real and C1.imag. Therefore, inside the function sum( ) , the variables real and imag refer to C3, C1.real and C1.imag refers to those of C1 and C2.real and C2.imag refer to of C2.

We can also pass an object as an argument to a non-member function. However, these functions can have access to the public member functions only through the objects passed as an arguments to it. These functions cannot hàve access to the private data members.

11. Returning Objects

A function may return an object to the function or caller. Following program illustrates how an object is created within a function and return to another function.

Program for returning objects from a function

```
#include<iostream>
using namespace std;
class complex // x+iy form
{  float real, imag;
     public;
        void getdata(float x, float y)
        {  real = x;
           imag = y;}
     friend complex sum(complex, complex);
     void display(complex);
};
  complex sum(complex C1, complex C2)
  {  complex C3;    // objects C3 is created.
     C3.x = C1.x+C2.x;
     C3.y = C1.y+C2.y;
     return(C3);    //return object C3
  }
  void complex :: display(complex C)
  {  cout <<C.x<<"+i"<<C.y<<"\n";
  }
  int main()
  {  complex P, Q, R;
     P.getdata(6.1,3.1);
     Q.getdata(5.8,7.2);
     R=sum(P,Q);//R=P+Q
     cout << "P="; P.display(P);
     cout << "Q="; Q.display(Q);
     cout << "R="; R.display(R);
     return 0;
 }
```

Output of the Above Program

P = 6.1 + i3.1

Q = 5.8 + i7.2

R = 11.9 + i10.3

The above program adds two complex numbers P and Q to produce a third complex number R and displays all the three numbers.

When an object is returned by a function, a temporary object is automatically created which holds the return value. It is this object that is actually returned by the function. After returning a value, the object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. There are various ways to solve this problem one way is that define a copy constructor, which will be explained in later chapter.

12. Const Member Function

The const modifier becomes a part of the object type and also can be applied to user-defined types:

For example

```
class Invoice {
 public:
 void giveDiscount(int d) {
  //...
 }
};
const Invoice i;          // a constant user-defined object
i.giveDiscount();         // error: 'i' is const
```

Because Invoice::giveDiscount() has the potential to alter the internal state of the object, the compiler refuses to let you invoke this member function on i, which is declared as const. However, individual member functions themselves can be declared const, which tells the compiler that those member functions do not alter the internal state of the object and therefore are safe to invoke on const objects. *For example*

```
class Invoice {
 public:
  int sum() const { // a constant member function
   //...
  }
};
const Invoice i;       // declares a constant user-defined object
int s = i.sum();       // ok: Invoice::sum() is const
```

C++ directly supports const as a language feature, so the compiler provides free compile-time checking of const objects. With properly designed interfaces that utilize const member functions, you are free to declare and use constant user-defined objects; only the const interface of those objects will be available. It is much better to use const early and often in your designs.

When designing const member functions, you may encounter a situation in which altering the internal state of the class is necessary. *For example*

```
class CustomerInvoice : public Invoice {
 public:
  int sum() const {
   // calculate sum, and save for later
   sumCache = items.size() + custItems.size();  // error
   return sumCache;
  }
  //...
 protected:
  //...
  int sumCache;
};
```

In the above example, the value of sumCache cannot be altered, because CustomerInvoice::sum() is declared to be const. However, we really do want to cache the value for later use; so how can this be done? The answer is "with the mutable keyword", which is the converse of const when applied to member variables. *For example*: CustomInvoice can be corrected as follows:

```
class CustomerInvoice : public Invoice {
 public:
  int sum() const {
   // calculate sum, and save for later
   sumCache = items.size() + custItems.size();   // ok
   return sumCache;
  }
  //...
 protected:
  //...
  mutable int sumCache;           // mutable member variable
};
```

Now, the compiler allows sumCache to be modified within the CustomerInvoice::sum() const member function because sumCache was declared to be mutable. The mutable keyword instructs the compiler to accept changes to the variable declared with it, even in const member functions.

13. Static Class Members

A C++ class, as you well know, can contain data and functions. It turns out that both data and function members of a class can be made *static*. Static members behave like ordinary members in many ways. They obey the same C++ class access rules provided by the keywords *public*, *private*, and *protected*. They are contained within the scope of the class, and do not include the global namespace or names used in other classes. Finally, they must be accessed using the . or -> operators to get inside the class scope boundary. However, unlike normal class members, static members may be accessed directly by applying the :: scope operator to the class type name. It is not even necessary that an object of the class to be constructed to access the static members.

13.1 Static Data Members

A data member of a class can be qualified as static. The properties of a *static* member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. *For example*: a static data member can be used as a counter that records the occurrences of all the objects.

Following program illustrates the use of a static data member.

Program

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

```
class item
{
     static int count;
     int number;
  public:
     void getdata(int a)
     {
          number = a;
          count++;
     }
     void getcount(void)
     {
       cout << "count: " ;
       cout << count << "\n";
     }
};
int item :: count;
int main()

{
  item a, b, c;    // count is initialized to zero
  a.getcount();    // display count
  b.getcount();
  c.getcount();
  a.getdata(100);  //getting data into object a
  b.getdata(200);  //getting data into object b
  c.getdata(300);  //getting data into object c
  cout << "After reading data" << "\n";
  a.getcount();       // display count
  b.getcount();
  c.getcount();
  return 0;
}
```

Output

count : 0

count : 0

count : 0

After reading data

count : 3

count : 3

count : 3

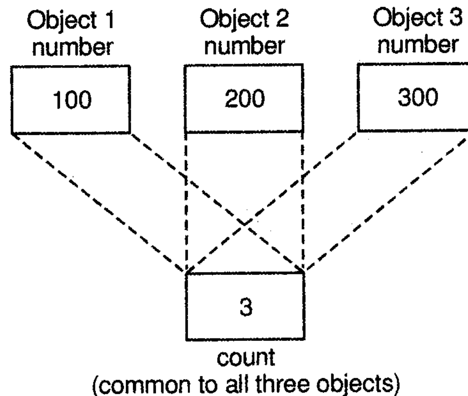
Note: Notice the following statement in the program:

```
int item:: count;     // definition of static data member
```

Note that the type and scope of each *static* member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables.*

The *static* variable *count* is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed.

Following figure shows how a static variable is used by the objects.



Sharing of a static data member

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

13.2 Static Member Functions

Like *static* member variable, we can also have *static* member functions. A member function that is declared *static* has the following properties.

- A *static* function can have access to only other static members (functions or variables) declared in the same class.
- A *static* member function can be called using the class name (instead of its objects).

Following program illustrates the implementation of these characteristics. The *static* function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the *static* variable count.

The function **showcode()** displays the code number of each object.

Program using static member function

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

```
{
     int code;
     static int count;                     // static member variable
     public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number:" << code << "\n";
        }
        static void showcount(void)  // static member function
        {  cout << "count: "<< count<< "\n";
        }
};
int test:: count;
int main()
{
   test t1, t2;
   t1.setcode();
   t2.setcode();
   test:: showcount();                     // accessing static function
   test t3;
   t3.setcode();
   test::showcount();
   t1.showcode();
   t2.showcode();
   t3.showcode();
   return 0;
 }
```

Output

count : 2

count : 3

object number :1

object number :2

object number :3

Note: Note that statement

```
code = ++count;
```

is executed whenever **setcode()** function is invoked and the current value of count is assigned to **code.** Since each object has its own copy of code, the value contained in **code** represents a unique number of its object.

Remember, the following function definition will not work:

```
static void showcount()
{ cout<< code;     // code is not static   }
```

14. Pointer to Members

PU
Oct. 2010 – 5M
★ Explain pointer to data members using suitable example.

A pointer that points to a member of a class and not to a specific instance of that member in an object, such a pointer is called a pointer to a class member or a pointer-to-member. It is not same as a normal C++ pointer. It only provides an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the operators like · and → cannot be applied to them. Special pointer -to- member operators .* and →* must be used to access a member of a class given a pointer to that member.

The address of a member can be obtained by applying the operator '&' to a "fully qualified" class member name. And the class member pointer can be declared using the operator ::* with the class name.

The dereferencing operator→* is used to access a member when we use pointers to both the object and the member. The dereferencing operator . * is used when the object itself is used with the member pointer.

By using the deferencing operators (.*,→*) in the main we can also invoke a pointer to member functions as shown below

```
(Object-name.* pointer-to-member function) (20);
(pointer-to-object→* pointer-to-member function)(20);
```

Consider the following example:

```
class A
{
  private:
  int x;
  public:
  void display();
}
```

A pointer to the member x can be defined as:

```
int A::*pm = &A::x;
```

The pm pointer created acts like a class member in that it must be invoked with a class object. It can also be used to access the member x inside member functions.

```
A a; //a is an object of A
cout<<a.*pm ;//display
cout<<a.x; //same as above
```

Following program illustrates the use of .* operator

```
#include<iostream>
using namespace std;
class A
{
  public:
  int value;
  void getdata(int i)
```

```
   {value =i;}
   int double_value()
   { return value + value ;}
};
   int main()
   { int A::*data;           // data member pointer
     A a;                          // a is the object of class A
     data =&A::value;              //get address of value
     void(A::*pt)(int) = A::getdata() //pointer to function getdata()
     a.*pt(12);     //involves getdata
     cout<<"The value is "<<a.*data<<"\n"
     cout<<"Total = "<<double_value()<<"\n";
     A *po = &a;
     po→*pf(17);
     cout<<"Total="<<double_value()<<"\n";
     return 0;
}
```

In the above program, in main(), the two member pointers are created, i.e., data and pt. Look at the syntax of each declarations. While declaring pointers to members, you must specify the class and use the scope resolution operator. The program also creates object of A called a. According to the above program, member pointers may point to either functions or data. Next, the program obtains the addresses of value and getdata(). These addresses are really just offsets into an object of type A, at which point value and double_value() will be found. Next, to display the values of each object's value, each is accessed through data. Finally the program uses pt to call the getdata function. In order to correctly associate the .* operator the extra parenthesis are necessary.

Next the program declares *po*, the pointer-to -object *a* and stores the address of *a*. *po* accesses the pt which stores the value 17 in the member value of class A.

We have already seen that →* operator is used when you are using a pointer to the object and the member.

Continuing from the previous example;

```
A * pa;
pa = &a; //pa is pointer to object a
cout<<pa→*pm; //display x
cout<<pa→x; // same as above
```

Following program illustrates this.

```
#include<iostream>
using namespace std;
class A
{
  public:int value;
  void getdata(int i)
  {
    value = i;
  }
  int double_value()
```

```
    {
      return value + value;
    }
};
int main()
{
    int A::*data;                    //data member pointer
    void (A::*pt)(int);                //function member pointer
    data = &A::value;
    pt(int)= &A::getdata;
    A a; A* P1; P1 = &a;
    a.*pt(12);
    int (A::*func)();
    func = &A::double_value;
    cout<<"The values are:";
    cout<<P1→*data<<"\n"
    cout <<"The value is doubled";
    cout << (P1→*func)();
return 0;
}
```

The P1 is pointer to object of type A, Therefore, the →* operator is used to access value and double_value().

15. Local Classes

Classes which are defined and used inside a function or block are called as local classes.

When a class is declared within a function, it is known only to that function and unknown outside of it.

For example

```
void result(int r)       //   function
{    ......
     ......
     class student    //   local class
     { ......
       ......
                      //   class definition
     };
     ----------
     ----------
     student S1(r)    //   creates student object
                      //   use student object.
}
```

There are several restrictions in constructing local classes. First, all member functions must be defined within the class declarations. A local class can access the static local variables which are declared within the function or those variables which are declared as extern. The local class does not access the local variables of the function in which it is declared. It may access the typenames and enumerators, defined by the enclosing function. No static variables may be declared inside a local class. Because of these restrictions local classes are not common in C++ programming.

16. Friend Functions

We know that there are three levels of internal protection for different members of a class: *public*, *protected* and *private*. In the case of *protected* and *private* members, these could not be accessed from outside the same class at which they are declared. That is, a non-member function cannot have an access to the private data of a class however, there could be a situation where we would like two classes to share a particular function. *For example*: Consider a case where two classes *student* and *teacher* have been defined. We would like to use a function *school()* to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of both these classes. Such a function need not be a member of any of these classes.

PU Oct. 2009 – 5M 1
★ What is friend function? Write a program to exchange private values of two classes using friend function.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class.

Syntax

```
friend return_type function_name(arguments);
```

For example

```
class xyz
{
  . . . .
  . . . .
  public:
    . . . .
    . . . .
  friend void pqr(void); // declaration
};
```

The function declaration should be preceded by keyword *friend*. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword friend or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. Though a friend function is not a member function but it has full rights to access the private members of the class.

In the following example we declare the friend function *duplicate:*

Program using Friend Function

```
// friend functions
#include<iostream>
using namespace std;
class CRectangle {
    int width, height;
     public:
    void set_values(int, int);
    int area(void) {return (width * height);}
    friend CRectangle duplicate(CRectangle);
};
void CRectangle::set_values(int a, int b) {
   width  = a;
```

```
  height = b;
}
CRectangle duplicate(CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return(rectres);
}
int main()
{
  CRectangle rect, rectb;
  rect.set_values(2,3);
  rectb = duplicate(rect);
  cout << rectb.area();
}
```

Output: 24

From within the *duplicate* function, that is a friend of *CRectangle,* we have been able to access the members *width* and *height* of different objects of type *CRectangle*. Notice that neither in the declaration of *duplicate()* nor in its later use in *main()* have we considered *duplicate* as a member of class *CRectangle*. It isn't.

The friend functions can serve, *for example*: To conduct operations between two different classes. Generally the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to make the process.

Such as in the previous example, it would have been shorter to integrate *duplicate()* within the class *CRectangle*.

Member functions of one class can be friend functions of another class. In such cases, they are defined using the scope resolution operator which is shown as follows:

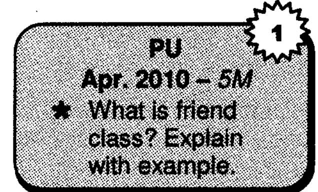
```
class p
{ . . .
  . . .
  int func1();     // member function of p
  . . .
};
class q
{ . . .
  . . .
  friend int p :: func1();  / func1 of p is friend of q
  . . .
};
```

The function func1() is a member of class p and a friend of class q.

Friend Classes

Like friend function, we can also define a class as friend of another class, allowing that the second one can access to the *protected* and *private* members of the first one.

Program using Friend Class




```
#include<iostream>
using namespace std;
class CSquare;
class CRectangle
{
    int width, height;
  public:
    int area(void)
      {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;
 }
```

Output: 16

In this example we have declared *CRectangle* as a friend of *CSquare* so that *CRectangle* can access the *protected* and *private* members of *CSquare*, more concretely *CSquare::side,* that defines the square side width.

You may also see something new in the first instruction of the program, that is the empty prototype of class *CSquare*. This is necessary because within the declaration of *CRectangle* we refer to *CSquare* (as a parameter in *convert()*).

The definition of *CSquare* is included later, so if we did not include a previous definition for *CSquare* this class would not be visible from within the definition of *CRectangle*.

Consider that friendships are not corresponded if we do not explicitly specify it. In our *CSquare* example *CRectangle* is considered as a class friend, but *CRectangle* does not have the same thing with *CSquare*, so *CRectangle* can access the *protected* and *private* members of *CSquare* but not the reverse way. Although nothing prevents us from declaring *CSquare* as a friend of *CRectangle.*

17. Unions and Classes

A union is a user defined data type whose size is sufficient to contain one of its members. At most, one of the members can be stored in an union at any time. A union is also used for declaring classes in C++. The members of a union are *public* by default.

Syntax

```
union user_defined_name{
private:
   //data
   //methods
public:
   //methods
protected:
   //data
};
user_defined_name object;
```

Declaration of an union without the user defined name or union tag is called as an anonymous union. The names of the members of an anonymous union must be distinct from other names. A global anonymous union must be declared static.

An anonymous union may not have protected or private members. An anonymous union may not have a member function also.

For example

```
union student
{
private:
   int rno;
   char name;
public:
   void getinfo();
   void putinfo();
};
```

18. Object Composition and Delegation

18.1 Object Composition

In computer science, **object composition** (not to be confused with function composition) is a way to combine simple objects or data types into more complex ones. Compositions are a critical building block of many basic data structures, including the tagged union, the linked list and the binary tree, as well as the object used in object-oriented programming.

A real-world example of composition may be seen in the relation of an automobile to its parts, specifically - the automobile has or is composed from objects including steering wheel, seat, gearbox and engine.

When in a language, objects are typed, types can often be divided into composite and noncomposite types and composition can be regarded as a relationship between types: an object of a composite type (e.g., *car*) "has an" object of a simpler type (e.g., *wheel*).

Composition must be distinguished from subtyping, which is the process of adding detail to a general data type to create a more specific data type. For instance, cars may be a specific type of vehicle: *car* is a *vehicle*. Subtyping doesn't describe a relationship between different objects, but instead, says that objects of a type are simultaneously objects of another type.

In programming languages, composite objects are usually expressed by means of references from one object to another; depending on the language, such references may be known as **fields, members**, **properties** or **attributes** and the resulting composition as a **structure**, storage record, tuple, **User-Defined Type (UDT)** or composite type. Fields are given an unique name so that each one can be distinguished from the others. However, having such references doesn't necessarily mean that an object is a composite. It is only called composite if the objects it refers to are really its parts, i.e., have no independent existence.

Example

```
C++ Composition
//composition
class car
{
private:
  Motor*motor;
public:
car() {motor=new * Motor();}
~ car() { delete motor;}
};
```

18.2 Delegation

In object-oriented programming, there are three related notions of **delegation**.

- Most commonly, it refers to a programming language feature making use of the method lookup rules for dispatching so-called self-calls as defined by Lieberman in his 1986 paper "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". Delegation as a language feature supports the prototype-based programming model.
- In its original usage, delegation refers to one object relying upon another to provide a specified set of functionalities. In research, this is often referred to as **consultation** or as **aggregation** in modeling.
- In CLI, a delegate is a form of type-safe function pointer usually used in an observer pattern as a means of telling which method to call when an event is triggered, keeping the method type.

Despite delegation being fairly widespread, relatively few major programming languages implement delegation as an alternative model to static inheritance. The self programming language incorporates the notion of delegation through its notion of mutable parent slots that are used upon method lookup on self calls.

In object-oriented programming, a **multicast delegate** is a delegate that points to several methods. Multicast delegation is a mechanism that provides functionality to execute more than one method. There is a list of delegates maintained internally, and when the multicast delegate is invoked, the list of delegates is executed.

Delegation is the simple yet powerful concept of handling a task over to another part of the program. In object-oriented programming, it is used to describe the situation where one object defers a task to another object, known as the delegate. This mechanism is sometimes referred to as aggregation, consultation or forwarding (when a wrapper object doesn't pass itself to the wrapped object.

Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime. It is used throughout Mac OS X (and its predecess or NeXTStep) as a means of customizing the behavior of program components. It enables implementations such as making use of a single OS-provided class to manage windows because the class takes a delegate that is program-specific and can override default behavior as needed. For instance, when the user clicks the close box, the window manager sends the delegate a windowShouldClose: call and the delegate can delay the closing of the window if there is unsaved data represented by the window's contents.

It has been argued that delegation may in some cases be preferred for inheritance to make program code more readable and understandable.

18.3 Language Feature

The short definition is that delegation defines method dispatching the way it is defined for virtual methods in inheritance: It is always the most specific method that is chosen during method-lookup. Hence it is the *original* receiver entity that is the start of method lookup even though it has passed on control to some other object (through a delegation link, not an object reference). Delegation has the advantage that it can take place at run-time and affect only a subset of entities of some type and can even be removed at run-time. Inheritance on the other hand typically targets the type rather than the instances and is restricted to compile time. On the other hand, inheritance can be statically type-checked while delegation generally cannot without generics (G. Kniesel has shown that a restricted version of delegation can be statically typesafe). Delegation can be termed "run-time inheritance for specific objects".

Solved Programs

1. Write a program in C++ to input the given string (include spaces) and reverse it using function which locates the end of the string and swaps the first character, the second last character and so on.

Solution

```
#include<iostream>
#include<string.h>
#include<cstdio.h>
using namespace std;
class reverseStr
{
```

```
  public:
    char str[256];
    int len;
    void getstr()
    {
      cout << "Please enter a string";
      cin>>str;
      int i;
      for(i = 0;str[i]! = '\0'; i++)
      len = len + 1;
    }
    void reverse()
    {
      char t;
      int a = len -1;
      for(int i=0; i<len/2; i++)
      {
        t = str[a];
        str[a] = str[i];
        str[i] = t;
        a--;}
      }
    };
void main()
{
  reverseStr string;
  string.getStr();
  string.reverse();
}
```

2. Write a program in C++ to find the reverse number of given number.

Solution

```
#include<iostream>
#include<string.h>
#include<cstdio.h>
#include<cstdlib.h>
using namespace std;
class revNo
{ public:
    long n;
    void getN()
    {cout<<"enter a number"; cin >> n; }
    long reverse()
    { int t, x;
      long n1;
      x = t = 0;
      n1 = n;
      while(n1 > 0)
        { t = n1 % 10;
          x = (x*10)+t;
          n1 = n1 / 10;
        }
        return x;
      }
};
```

```
void main()
{ revNo num;
  num.getN();
  long n = num.reverse();
  cout << endl << n << endl;
}
```

3. What will be the output of following program?

PU Oct. 2009 – *2M*

i.

```
#include<iostream.h>
class outer
{
        class inner
        {
                int data;
        } i;
        public:
          void create()
          {
                i.data=100;
          }
          void display()
          {
                cout<<i.data;
          }
    };
    void main()
    {
            outer obj;
            obj.create();
            obj.display();
    }
```

Solution

Output

Two Error

1. In i.data=100 statement show the error, outer::inner::data is not accessible.
2. In cout <<i.data statement show the error, outer::inner::data is not accessible.

PU Oct. 2009 – *2M*

ii.

```
#include<iostream.h>
class base
{
    private: int i;
};
class derived: public base
{
    private: int j;
};
void main()
{
    cout<<endl<<sizeof(derived)<<endl<<sizeof(base):
    derived dobj;
    base bobj;
    cout<<endl<<sizeof(dobj)<<endl<<sizeof(bobj);
};
```

Solution

Output

4

2

4

2

iii.

```
#include<iostream.h>
class mca
{
  public: int a;
  private: int b;
  protected: int c;
};
void main()
{
  mca obj1;
  cout <<obj1.a<<obj1.b<<obj1.c;
}
```

PU Apr. 2010 – *2M*

Solution

The given code will result into following errors:

mca::b is not accessible.

mca::c is not accessible.

Explanation: An object of class cannot directly access the private and protected members of that class. In the given code, the class mca is having 'b' as a private member and 'c' as a protected member. We are trying to access these members by using the statement:

cout <<obj1.a<<obj1.b<<obj1.c; which is not allowed.

iv.

```
# include<iostream.h>
class test
{
  int a, b;
  public :
      void getdata(int x, int y)
      { a = x; b = y;
      }
      void mul() const
      {
        a =a*b;
        cout<< a;
      }
};
void main()
{
  test t;
  t.getdata(2, 4);
  t.mul();
}
```

PU Apr. 2010 – *2M*

Solution

The given code will result into following error:

Cannot modify a const object.

Explanation: The above mentioned error would occur because we are trying to modify the value of variable 'a' inside a const block. Anything which is constant or inside a constant block needs the assignment of value to the variable directly (*Example*, a = 10); however we are assigning an expression to 'a' (a = a * b).

v.

```
class base
{
        private:int x;
        protected: int y;
};
class derived:public base
{
        protected:
        int a, b;
        void change()
        {
                a=x;
                b=y;
        }
};
```

PU Oct. 2010 – *2M*

Solution

Given code would result into following error:

base::x is not accessible.

Explanation: Private members of class cannot be inherited and hence becomes inaccessible if we try to access them in derived class. In given code 'x' is private in base class and we are trying to access it in derived class.

vi.

```
class test
{
        static int count;
        public:
                static void showcount(void)
                {
                        cout<<"count:"<<++count;
                }
};
int test::count;
main()
{
        test t1,t2,t3;
        t1.showcount();
        t2.showcount();
        t3.showcount();
}
```

PU Oct. 2010 – *2M*

Solution

The given code will result into following error:

Undefined symbol 'cout'

Explanation: Since header file <iostream.h> is not included, the above mentioned error would occur. If we include this header file in the given code then the output would be: Count:1Count:2Count:3.

vii.
```
class Test
{
  static int i;
  int j;
};
int Test :: i;
int main()
{
  cout<<sizeof(Test);
  return 0;
}
```

PU Oct. 2011 – *2M*

Solution

If <iostream.h> file is included in the given code then the output would be: 2

Since static members are stored in a separate memory location that is not in the scope of object.

4. **What is friend function? Write a program to exchange private values of two classes using friend function.**

PU Oct. 2009 – *5M*

Solution

```
#include<iostream.h>
#include<conio.h>
class exchange
{
  private: int a, b;
  public:friend exchange test(int x, int y);
};
exchange test(int m ,int n)
{
  exchange temp;
  temp = m;
  m = n;
  n = temp;
  cout << "\n The value of a =" << m;
  cout << "\n The Value of b = "<< n}
}
void main()
{
  int a, b;
  exchange rec;
  clrscr();
  cout << "\n Enter value of a and b";
  cin >> a>>b;
  rec.test(a,b);
  getch();
}
```

5. **Write a C++ program illustrating static member function.**

PU Oct. 2009 – *5M*

Solution

```
#include<iostream.h>
#include<conio.h>
  class example
```

```
{
        private: static int sum;
        int x;
      public: example()
            { sum=sum+1;
              x=sum;
            }
            ~example()
              { sum=sum-1; }
            static void exforsys()
            {
                  cout<<"\n Result is: "<<sum;
            }
            void number()
            {
              cout<<"\n Number is: "<<x;
}
};
void main()
{
   example e1;
   clrscr();
   example::exforsys();
   example e2,e3,e4;
   example::exforsys();
   e1.number();
   e2.number();
   e3.number();
   e4.number();
  getch();
}
```

EXERCISES

A. Review Questions

1. What is a class? How does it accomplish data hiding?
2 How does a class declaration differ from a class definition?
3. What are objects? How they are created?
4. Describe how the data members of a class can be initialized in C++.
5. How is the member function of a class accessed in C++?
6. What is an array of object?
7. What is a friend function? What are the merits and demerits of using friend functions?
8. What are the two types of member functions?
9. When do you define a member function inside and outside the class definition?
10. What is meant by local classes?

B. Programming Exercise

1. Define a class to represent a back account. Include the following members:

   Data members: Name of the depositor, Account number, type of account, balance amount in the account.

   Member functions: to assign initial values, to deposit an amount, to withdraw an amount after checking the balance, to display name and balance. Write a program to test your class.

Collection of Questions asked in Previous Exams PU

1. What will be the output of following program?

   i. **[Oct. 2009 – 2M]**

```
#include<iostream.h>
class outer
{
        class inner
        {
                int data;
        } i;
        public:
          void create()
          {
               i.data=100;
          }
          void display()
          {
               cout<<i.data;
          }
      };
      void main()
      {
             outer obj;
             obj.create();
             obj.display();
      }
```

   ii. **[Oct. 2009 – 2M]**

```
#include<iostream.h>
class base
{
     private: int i;
};
class derived: public base
{
     private: int j;
};
void main()
{
     cout<<endl<<sizeof(derived)<<endl<<sizeof(base):
     derived dobj;
     base bobj;
     cout<<endl<<sizeof(dobj)<<endl<<sizeof(bobj);
};
```

iii. **[Apr. 2010 – 2M]**

```
# include<iostream.h>
class mca
{
  public: int a;
  private: int b;
  protected: int c;
};
void main()
{
  mca obj1;
  cout <<obj1.a<<obj1.b<<obj1.c;
}
```

iv. **[Apr. 2010 – 2M]**

```
# include<iostream.h>
class test
{
  int a, b;
  public :
       void getdata(int x, int y)
       { a = x; b = y;
       }
       void mul() const
       {
         a =a*b;
         cout<< a;
       }
};
void main()
{
  test t;
  t.getdata(2, 4);
  t.mul();
}
```

v. **[Oct. 2010 – 2M]**

```
class base
{
    private:int x;
    protected: int y;
};
class derived:public base
{
    protected:
    int a, b;
    void change()
    {
        a=x;
        b=y;
    }
};
```

vi. **[Oct. 2010 – 2M]**

```
class test
{
    static int count;
    public:
        static void showcount(void)
        {
            cout<<"count:"<<++count;
        }
};
int test::count;
main()
{
    test t1,t2,t3;
    t1.showcount();
    t2.showcount();
    t3.showcount();
}
```

vii. **[Oct. 2011 – 2M]**

```
class Test
{
  static int i;
  int j;
};
int Test :: i;
int main()
{
  cout<<sizeof(Test);
  return 0;
}
```

2. What is friend function? Write a program to exchange private values of two classes using friend function. **[Oct. 2009 – 5M]**
3. Write a C++ program illustrating static member function. **[Oct. 2009 – 5M]**
4. What is friend class? Explain with example. **[Apr. 2010 – 5M]**
5. Explain pointer to data members using suitable example. **[Oct. 2010 – 5M]**

6 Constructor And Destructor

1. Introduction

Generally, objects are needed to initialize variables or assign dynamic memory during their process of creation to become totally operative and to avoid returning unexpected values during their execution. Consider the following program,

Program

```
#include<iostream>
using namespace std;
class CRectangle
{  int x, y;
  public:
     void set_values(int, int);
     int area(void) {return (x*y);
};
void CRectangle::set_values(int a, int b)
{ x = a;
  y = b;}
int main()
{  CRectangle rect, rectb;
  rect.set_values(3,4);
  rectb.set_values(5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area:" << rectb.area() << endl;
 }
```

What would happen if in the previous example we called the function *area()* before having called function *set_values*? Probably an indetermined result since the members *x* and *y* would have never been assigned a value.

In order to avoid that, a class can include a special function: a *constructor*, which can be declared by naming a member function with the same name as the class. This constructor *function* will be called automatically when a new instance of the class is created (when declaring a new object or allocating an object of that class) and only then. Constructors are also called when an object is created as part of another object.

Similarly, when your object goes out of scope, the memory used by the object must be reclaimed. C++ provides a special member function, called the destructor, which is called whenever your object is destroyed so that you may perform any clean-up processing, such as freeing memory or other system resources obtained by the object.

C++ also provides two other special functions that play a special role. Whenever an object must be copied, its copy constructor is invoked.

Finally, whenever an object is assigned a value, its assignment operator is invoked.

2. Constructor

A constructor is a special member function for automatic initialization of an object. They have the same name as the class name. There can be any number of overloaded constructors inside a class, provided they have a different set of parameters. There are some important qualities for a constructor to be noted.

2.1 Rules for Writing a Constructor Function

i. Constructors have the same name as the class.

ii. Constructors do not return any values (not even void).

iii. Constructors are invoked first when the objects are created. Any initializations for the class members, memory allocations are done at the constructor.

iv. Constructors should be declared in the public section and in rare circumstances it should be declared in private section.

v. Constructors cannot be virtual or static.

vi. An object with a constructor (or destructor) cannot be used as a member of a union.

vii. They make implicit calls to the operators new and delete where memory allocation is required.

viii. Constructors can have default arguments.

The general syntax of the constructor function in C++ is as follows:

```
class  user_name
{  private:
          . . . .
          . . . .
   protected:
          . . . .
          . . . .
   public:
        user_name();     //constructor declared
        . . . .
        . . . .};
user_name :: user_name()  // constructor defined
{  . . . .
   . . . . }
```

For example

```
class CRectangle
{ int width, height;
  public:
    CRectangle();              //constructor declared
    int area(void);
};
CRectangle::CRectangle()       //constructor defined
{
   . . . .
   . . . .
}
```

2.2 Types of Constructors

Constructors come in many forms. They are parameterized constructor, default constructor, non-default constructor, copy constructor and dynamic constructor.

Types of Constructors
i. Parameterized
ii. Default
iii. Copy
iv. Dynamic

i. Parameterized Constructor

In the above example, the constructor CRectangle initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values where they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.

The constructors that can take arguments are called as parameterized constructors.

For example

```
class CRectangle
{  int width, height;
   public:
       CRectangle(int w, int h);  //Parameterized constructor
       int area(void);
};
CRectangle::CRectangle(int w, int h)
{  width  = w;
   height = h;
}
```

ii. Default Constructor

A default constructor is a constructor that accepts no parameters. This may be achieved by either providing default arguments for the constructor, or overloading the constructor with another constructor that has no arguments. The following example creates a default constructor by providing default values for the arguments.

It is used for initializing the objects of a class. In other words, a default constructor function initializes the data members with no arguments.

Syntax

```
className();   // for default constructor
```

Example: The default constructor for a class x has the form x :: x().

A constructor which has all default arguments, is also a default constructor, since it can be called with no arguments.

Example: x :: x(const int x=0)

Default constructors allow objects to be created without passing any parameters to the constructor. *For example*: The declaration

```
String s;
```

results in a string s that does not yet have a value; it is an empty string.

The default constructor usually creates an object that represents a "null" instance of the particular type the class denotes. The default constructor for a complex number might result in an object with value zero, while the default constructor for a linked list might would result in an empty list.

Often you will allow users of your classes to pass arguments to the constructor. Rather than provide a separate default constructor that takes no arguments, it is better to provide a constructor with default arguments that can serve as a default constructor or as a constructor that takes the arguments it specifies. This makes your code more compact and leads to more code reuse.

For example: The two constructors of a String, String() and String(const char* str), can be combined into a single constructor that has a default argument: String(const char* str=0).

There can only be **one** default constructor, so don't add default arguments to all the arguments of every constructor - your compiler is likely to complain.

Default arguments are not restricted to constructors, they can be used by any member function. Provide appropriate default values wherever appropriate.

```
#include<iostream>
#include<iomanip>
class Date
{ // Attributes of the class
  private:
  int day;
  int month;
  int year;
  // Methods of the class
  public:
  // Default constructor
```

```
   Date(int d=1, int m=1, int y=2000) { set(d, m, y); }
   void set(int d=1, int m=1, int y=2000);
   void validate();
   void display()
   {   cout << setw(2) << setfill('0') << day << "/"
            << setw(2) << setfill('0') << month << "/"
            << setw(4) << setfill('0') << year << endl;
   }
};
```

The following declarations are all valid for the Date class.

```
Date d1(20, 1, 1964);  // 20/01/1964
Date d2(20, 1);        // 20/01/2000
Date d3(20);           // 20/01/2000
Date d4;               // 01/01/2000
```

In case of non-default argument it has parameters. These parameters are used to initialize the object to a particular state.

You can overload the non-default constructors (like any function), if the number, order and type of the parameters are different. This turns out to be very handy for objects with complex state.

A non-default constructor can use default parameters and can thus emulate a default constructor.

Syntax

```
className(parameter list);    // for non default constructor
```

iii. Copy Constructor

Before looking at the copy constructor, it's important to understand the difference between initialisation and assignment. Initialisation occurs when an object is created, and assignment occurs when an assignment operator is used in an expression to set a previously defined object to a value.

PU
Oct. 2011 – *5M*
★ Write short note on copy constructor.

```
Date d1(20, 1, 1964);
Date d2 = d1;  // Initialisation
Date d3;
d3 = d1;       // Assignment
```

In the above example, each member of the instance d1 is copied to the instance d2, a process called memberwise initialisation. A copy constructor is a special type of constructor that is used to initialise the class object, to the value of another class object of the same class. The copy constructor is like any other constructor, but has a reference to the object to be copied as its parameter.

Syntax

```
className(const className &parameterName);
```

The following example defines a copy constructor for the date class.

```
#include<iostream>
#include<iomanip>
```

```
class Date
{
  // Attributes of the class
  private:
      int day,month, year;
    // Methods of the class
  public:
    // Default constructor
  Date(int d=1, int m=1, int y=2000) {set(d, m, y);}
    // Copy Constructor
  Date(const Date &copy)
  {
    day = copy.day;
    month = copy.month;
    year = copy.year;
  }
  void set(int d=1, int m=1, int y=2000);
  void validate();
  void display()
  {
    cout << setw(2) << setfill('0') << day << "/"
         << setw(2) << setfill('0') << month << "/"
         << setw(4) << setfill('0') << year << endl;
  }
};
```

The copy constructor may be called implicitly when an object is initialised to another object, or explicitly by passing the object to be copied as a parameter. The following example illustrates the two methods.

```
Date d1(20, 1, 1964);
Date d2 = d1;   // Implicit Copy
Date d3(d1);    // Explicit Copy
```

In this particular example, member wise initialization would have been sufficient. Our copy constructor merely performs a member wise initialization so it isn't really required. When data members are more complex data objects, relying on the member wise initialization process may result in errors.

Note: Always provide a copy constructor for your classes. Do not let the compiler generate it for you. If your class has pointer data members, you must provide the copy constructor.

iv. Dynamic Constructor

While creating objects we can use constructors to allocate memory. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, this results in saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of object. "new" operator is used to allocate the memory. When you use new to get memory for a class object, the compiler executes the new operator function first to allocate the memory and then calls the class's constructor function. Likewise the "delete" operator is used to deallocate the memory. When you use delete to return the memory, the compiler calls the class's destructor function.

Usually, the compiler provides the new and delete operator functions. They are the same functions that allocate and return memory for intrinsic type objects.

The following program demonstrates the relationship involving new and the constructor functions and delete and the destructor function.

```
#include<iostream>
using namespace std;
// The Date class
class Date
{ int mo, da, yr;
public;
  Date()
  { cout<< "Date constructor"<< endl;}
  ~Date()
  { cout<< "Date destructor" << endl;}
  };
  int main()
  { Date * dt = new date;
    cout<< "Process the date" <<endl;
    delete dt;
    return 0;}
```

The output of the above program is

Date constructor

Process the date

Date destructor

In the above program, Date class contains a constructor and a destructor. These functions display messages when they run. When the new operator initializes the dt pointer, the constructor function executes. When the delete operator deletes the memory pointed to by the pointer, the operation calls the destructor function.

2.3 Guidelines for Implementing Constructors

Constructors are called frequently, not only by you when you declare objects, but also when the compiler creates temporaries. It is, therefore, important for the constructors to be compact and as efficient as possible. Below are a few guidelines to consider when implementing constructors.

i. When initializing data members from the constructor, use initialization rather than assignment. This will make your code more efficient and avoid extra calls to the constructors.

Suppose you have a Shape class with data members: center of type Point and color of type int. When implementing the constructor, you might be tempted to use assignment, to initialize the data members:

```
Shape::Shape(const Point center, const int color)
{ center = center;
  color  = color;
}
```

However, this results in an extra call to the constructor for Point and makes the code execute slower. To see why, let's look at how objects are constructed. There are two phases to object construction:

a. Data members are initialized in the order of their declaration in the class. This is called member initialization.

b. The body of the constructor is executed.

Also, for derived classes, these two steps are performed on the base classes first.

In the Shape constructor, `center` is constructed during Step 1, when the default constructor for Point is invoked. Then in Step 2, when the body of the constructor executes the assignment operator is invoked which changes the value of `center`.

Note that `center` will always be initialized before the body of the constructor is ever entered. However, you can control which constructor is called for `center` in the initialization list of the Shape constructor:

```
Shape::Shape(const Pointer center,const int color) : center(center)
color(color)
{ }
```

Specifying `center(center)` in the initialization list, tells the compiler to call the copy constructor rather than the default constructor during member initialization. Since center is properly initialized, the assignment is no longer needed in the body of the constructor.

When implementing your constructors, try to initialize all your data members using this technique. In most cases, all can be initialized this way, so there will be no need for any code in the body of the constructor. This makes your code much more readable and maintainable.

Actually, built-in types, such as int, do not have constructors. Therefore, it makes no difference whether you use assignment or initialization for variables with built-in types. But, the code is more manageable and easier to read if all data members are initialized the same way.

ii. Pay attention to the order of member initialization. If you use a data member, *d1* to initialize another data member *d2*, make sure *d1* is in fact initialized before initializing *d2*.

Consider a StringHandle class, which declares data members in the following order:

```
String _str;
int  _handle;
```

C++ language rules tell us that the members of a class are initialized in the order they appear in the class declaration, **not** in the order they appear in the initialization list of the constructor (e.g., _str will be initialized before _handle).

The following constructor will compile, but will result in run-time error:

```
StringHandle::StringHandle(const int h)
: _handle(h),_str(QueryHandle(_handle))
{    }
```

Because _str appears first in the class declaration, it will be intilialized before _handle. Unfortunately, _str uses _handle in its construction, which has not yet been constructed. Specifying _handle before _str in the initializer list doesn't matter.

To fix this problem, we can use the incoming argument instead of the class member:

```
StringHandle::StringHandle(const int h)
: _handle(h),_str(QueryStringHandle(h))
{    }
```

The reason that the initializer list order is ignored is so that the compiler can ensure that variables are destroyed in the reverse order of their construction. There is no guarantee that the constructor and destructor are implemented in the same source file. However, both the constructor and destructor see the same class declaration, so there is no ambiguity involved by using the declaration to drive the order of initialization.

iii. Reference data members and const objects **must** be initialized using the initializer syntax.

References must point to another already existing object and, therefore, must be initialized to point at the aliased object during initialization. A reference can only have one assignment for its lifetime. Similarly, constant objects can never be assigned a value, so they must also be initialized using an initializer. Here is a possible implementation of the constructor:

iv. Make the constructor as compact as possible to reduce overhead and to minimize errors during construction.

The constructor is called each time an object is created. Lots of times this occurs without your control, such as when the compiler creates temporaries. Initialization of an object should not take a very long time. Only do what is minimally necessary during construction. Reducing the amount of processing you perform during construction also reduces the chance of an error occurring.

v. Remember to invoke base class constructors from derived class constructors.

Constructors (as well as destructors and assignment operators) are not inherited by derived classes. During member initialization, just before the constructor body is executed, constructors are called for the data members and base classes of the class. As we know, the compiler uses the initializer list to determine which constructor to call. If you do not call the base class constructor from the initializer list of a derived class, the compiler will generate a call to the default constructor of the base class. If the base class does not have a default constructor, the compiler will complain.

It is a good idea to initialize base classes and your data members from the initializer list of derived classes. I tend to put my calls to base class constructors before initializing the data members.

Suppose we decided to derive a class Triangle from Shape:

```
class Triangle : public Shape
{
   public:
      Triangle(const Point center,const int color,
           const Point v1, const Point v2, const Point v3);
   private:
      Point _v1, _v2, _v3;
};
```

We call the Shape constructor from the initialization list of Triangle's constructor:

```
Triangle::Triangle( const Point center,const int color,
                const Point v1, const Point v2, const Point v3)
: Shape(center, color), _v1(v1), _v2(v2), _v3(v3)
{    }
```

2.4 Overloading Constructors

Like any other function, a constructor can also be overloaded with several functions that have the same name but different types or numbers of parameters. Remember that the compiler will execute the one that matches at the moment at which a function with that name is called. In this case, at the moment at which a class objects is declared. In fact, in the cases where we declare a class and we do not specify any constructor the compiler automatically assumes two overloaded constructors ("*default constructor*" and "*copy constructor*"). *For example*

```
class CExample
{
  public:
    int a,b,c;
    void multiply(int n, int m) { a=n; b=m; c=a*b; };
  };
```

with no constructors, the compiler automatically assumes that it has the following constructor member functions:

a. *Empty constructor:* It is a constructor with no parameters defined as *nop* (empty block of instructions). It does nothing.

```
CExample::CExample() { };
```

b. *Copy constructor:* It is a constructor with only one parameter of its own type that assigns to every non-static class member variable of the object a copy of the passed object.

```
CExample::CExample(const CExample& rv) { a=rv.a;  b=rv.b;  c=rv.c; }
```

It is important to realize that both default constructors- the *empty constructor* and the *copy constructor* exist only if no other constructor is explicitly declared. In case that any constructor with any number of parameters is declared, none of these two default constructors will exist. So if you want them to be there, you must define your own ones. Of course, you can also overload the class constructor providing different constructors for when you pass parameters between parenthesis and when you do not (empty):

```
// overloading class constructors
#include<iostream>
using namespace std;
class CRectangle {
  int width, height;
  public:
    CRectangle();
    CRectangle(int,int);
    int area(void) {return (width*height);}
};
CRectangle::CRectangle()
{ width = 5;
  height = 5; }
CRectangle::CRectangle(int a, int b)
{ width = a;
  height = b;}
int main() {
  CRectangle rect(3,4);
```

```
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output: rect area : 12

rectb area: 25

In this case **rectb** was declared without parameters, so it has been initialized with the *constructor* that has no parameters, which declares both **width** and **height** with a value of **5**.

Notice that if we declare a new object and we do not want to pass parameters to it we do not include parenthesis ():

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

3. Multiple Constructors in a Class

C++ allows us to define more than one constructor in the same class.

For example

Consider the definition of following class.

```
class Sample
{  int p,q;
   public:
      Sample() { P=0; q=0; // constructor_no argument
   Sample(int m, int n)    // constructor_two arguments
   { p= m; q=n;}
   Sample(sample &i)       // Constructor-one argument
   {p=i.p;q=i.q;}
};
```

The above sample class contains the three constructors. The first constructor receives no arguments. The second receives two integer arguments and the third receives one integer object as an argument. *For example*: The declaration,

```
Sample S1;
```

will invoke the first constructor and set both p and q of S1 to zero. The statement,

```
Sample S2(40,60);
```

would call the second constructor and initialize the data members p and q of S2 to 40 and 60 respectively. Finally, the statement

```
Sample S3(S2);
```

would call the third constructor and copies the values of S2 into S3. In other words, sets the value of every data element of S3 to the value of the corresponding data element of S2. As already seen, such a constructor is called the copy constructor. As explained in previous chapters, the process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class we say that the constructor is overloaded as already seen.

4. Constructor with Default Arguments

In C++, we can define constructors with default arguments.

For example

The constructor CRectangle( ) can be declared as follows:

```
CRectangle(int width, int height=3);
```

The default value of the argument height =3. Then, the statement CRectangleC(5); assigns the value 5 to the width variable and 3 to the height variable (by default). However, the statement,

```
CRectangle(4,7);
```

assigns 4 to width and 7 to height. The actual parameter when specified, overrides the default value. The difference between the default constructor (add :: add()) and the default argument constructor (add:: add(int=0)) – is that the default argument constructor int =0 can be called with either one argument or no arguments, When called with no arguments, it becomes a default constructor. When both forms, i.e., default argument constructor and default constructor are used in a class, it causes ambiguity, *for example*: The declaration

```
add obj;
```

is ambiguous, i.e., which one constructor to invoke, i.e., add :: add() or add :: add(int=0, int =0)

5. Dynamic Initialization of Objects

In C++, the class objects can be initialized at run time (dynamically), We have the flexibility of providing initial values at execution time. The advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different formats of data at run time depending upon the situation. The following program illustrates this concept.

Dynamic Initialization of objects

```
#include<iostream>
using namespace std;
class employee
{  int empl_no;
   float salary;
   public:
       employee() // default constructor
       {  }
          employee(int empno, float a)
                  //  constructor with arguments
   {  empl_no = empno;
      salary = a;}
   employee(employee &emp)//copy constructor
   {  cout << "\n Copy constructor working \n";
      empl_no= emp.empl_no;
      salary = emp.salary;}
   void display(void)
   {  cout << "\n Emp No.:" << empl_no<<"salary:"<<salary<<endl;} };
   void main()
```

```
{ int eno;
float sal;
clrscr();
cout <<"Enter the employee number and salary \n";
cin >>eno>>sal;
employee obj1(eno, sal); // dynamic initialization of object
cout < "Enter the employee number & salary \n";
cin >> eno >> sal;
employee obj2(eno,sal); // dynamic initialization of object
obj1.display(); // function called
obj2.display();
employee obj3 = obj2 ; // copy constructor called
obj3.display();
getch();
 }
```

The output of the above program will be :

```
Enter the employee number and salary
2000      5000
Enter the employee number and salary
2001      7000
Emp.No. : 2000     Salary :  5000
Emp.No. : 2001     Salary :  7000
```

Copy constructor working

```
Emp.No. : 2001Salary : 7000
```

The following code segment shows a constructor with default arguments;

```
#include<iostream>
using namespace std;
class add
{ private:
          int num1,num2,num3;
  public:
          add(int=0,int=0);// Default argument constructor
                           // to reduce the number of constructors
          void enter(int, int);
          void sum();
          void display(); };
  //Default constructor definition
add:: add(int n1, int n2)
{ num1=n1;
  num2=n2;
  num3=0;}
void add::sum()
{ num3=num1+num2; }
void add::display()
{ cout   << "\n The sum of two numbers is:"<< num3 << endl;
 }
```

Now using the above code, objects of type add can be created with no initial values, one initial value or two initial values.

For example

```
add obj1, obj2(7), obj3(15,20);
```

Here, obj1 will have values of data members

```
num1=0, num2 =0 and num3=0
```

obj2 will have values of data members num1=7,num2=0 and num3=0

obj3 will have values of data members num1=15,num2=20 and num3=0

If two constructors for the above class add are

```
add::add() { } // Default constructor
and add::add(int=0, int=0); //Default argument constructor
```

Then the default argument constructor can be invoked with either two or one or no parameter(s).

6. Const Object

A const object is defined the same for a user defined data type as a built-in type.

For example

```
const blob b(2); // object b is constant
```

Here, b is a const object of type blob. Its constructor is called with an argument of two. Consider another example:

```
const complex C(real, imag); //object C is constant
```

Here, C is a const object of type complex. If we modify the values of real and imaginary the compiler will generate error.

If you declare a member function const, you tell the compiler the function can be called for a const object. A member function that is not specifically declared const is treated as one that will modify data members in an object, and the compiler will not allow you to call it for a const object. In short, a const object can call only const member functions and if it calls non-const member functions, the compiler generates error. Consider the following program,

```
#include<iostream>
#include<date.h>
using namespace std;
class Date
{
  int month, day, year;
  public:
  Date(int m, int d, int y)
  {
    month = m; day =d;  year =y;
  }
  void display()
```

```
  {
    cout<<month<<'/'<<day<< '/'<<year;
  }
};
int main()
{
  const date dt(y,25,2006); // dt is a const object of type date
  dt.display(); //compiler generate error
}
```

The call to the display member function generates a compiler error message because we are calling a non_const function, i.e., display for a const object. Even though the function does not change the data members of the object, the compiler has no way of knowing that, and generates the error.

7. Destructor

The destructor is an opposite of constructor.

Like constructor, the destructor is a member function whose name is the same as the class name and it is automatically called when an object is released or destroyed from the memory, either because its scope of existence has finished (*For example*: If it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using operator delete.

Rules for Writing a Destructor Function

i. The destructor must have the same name as the class with a tilde (~) as prefix.
ii. It never takes any arguments nor returns any value.
iii. It cannot be declared as static, volatile or const.
iv. It takes no arguments and therefore cannot be overloaded.
v. It should have public access in the class declaration.

The use of destructors is specially suitable when an object assigns dynamic memory during its life and at the moment of being destroyed we want to release the memory that it has used. Whenever new is used to allocate memory in the constructors, we should use delete to free that memory.

Syntax

```
~className( ); // destructor
```

Program

```
#include<iostream>
using namespace std;
class CRectangle
{
    int *width, *height;
```

```
    public:
    CRectangle(int, int);
    ~CRectangle();      //Destructor
    int area(void) {return (*width * *height);}
};
CRectangle::CRectangle(int a, int b)
{
  width   = new int;
  height  = new int;
  *width  = a;
  *height   = b; }
CRectangle::~CRectangle()
{
  delete width;
  delete height;}
int main()

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

Output: rect area : 12
rectb area : 30

Solved Programs

1. Write a program to demonstrate overloading of constructor.

PU
Oct. 2010 – *5M*

Solution

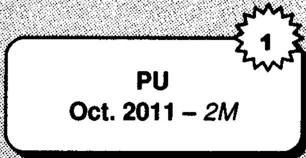
```
/*Program to demonstrate overloading of constructor*/
#include<iostream.h>
class MyClass
{
  public:
    int x;
    int y;
  //Overload the default constructor
  MyClass()
  {
    x=y=0;
  }
  //Constructor with one parameter
  MyClass(int i)
  {
    x=y=i;
```

```
  }
  // Constructor with two parameters
  MyClass(int i, int j)
  {
    x=i;y=j;
  }
};
int main()
{
  MyClass t;          // invoke default constructor
  MyClass t1(5);      // use MyClass(int)
  MyClass t2(9, 10); // use MyClass(int, int)
  cout<<"t.x:"<< t.x<<",t.y:"<<t.y<< "\n";
  cout<<"t1.x:"<< t1.x<<",t1.y:"<<t1.y<< "\n";
  cout<<"t2.x:"<< t2.x<<",t2.y:"<<t2.y<< "\n";
  return 0;
}
```

2. Write outputs with explanation:

i.

```
class sample
{
  public:
  int *ptr;
  sample()        {ptr = NULL;}
  sample(int i)
  {
    ptr = new int[i];
  }
  ~sample()
  {
    delete ptr;
  }
  void printVal()
  {
    cout<<"The value"<<*ptr;
  }
};
void someFunc(sample x)
{
  cout<<"say i am in someFun"<<endl;
}
int main()
{
  sample s1 = 10;
  someFunc(s1);
  s1.printVal();
}
```



Solution

If <iostream.h> file is included in the given code then the output would be:

say i am in someFun → call to someFunc(s1) will produce this output.

The value-1045 (some garbage value) → call to s1.printVal() will produce this output. The output is some garbage value because there is no value at the location pointed to by ptr.

ii.
```
class Test
{
  public:
  Test();
};
Test :: Test()
{
  cout<<"Constructor called\n";
}
int main()
{
  cout<<"Start\n";
  Test t1();
  cout<<"End\n";
  return 0;
}
```

PU
Oct. 2011 – *2M*

Solution

If <iostream.h> file is included in the given code then the output would be:

Start → output of the statement cout<<"Start\n";

End → output of the statement cout<<"End\n";

If the statement Test t1() is written as Test t1, then constructor will be called.

EXERCISES

A. Review Questions

1. What is constructor? What are the uses of declaring a constructor member function in a program?
2. Explain the different types of constructors.
3. What is parameterized constructor?
4. What is copy constructor? What is its purpose?
5. What is a default constructor?
6. Explain the new and delete operators.
7. What is destructor?
8. What is dynamic constructor?
9. What is mean by const object?

B. Programming Exercises

1. Write a class called CAccount which contains two private data elements, an integer accountNumber and a floating point accountBalance, and three member functions:
   a. A constructor that allows the user to set initial values for accountNumber and accountBalance and a default constructor that prompts for the input of the values for the above data members.
   b. A function called inputTransaction, which reads a character value for transactionType ('D' for deposit and 'W' for withdrawal) and a floating point value for transactionAmount, which updates accountBalance.
   c. A function called printBalance, which prints on the screen the accountNumber and accountBalance.
2. A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.

   Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.
3. Write a program to print the factorial of a given number using a constructor and a destructor member function.
4. Write a program to read n numbers (where n is defined by user) and find the average of the non-negative integer numbers. Also find, the deviation of the numbers using new and delete operators.

Collection of Questions asked in Previous Exams PU

1. Write a program to demonstrate copy constructor. **[Apr. 2010 - 5M]**
2. Write a program to demonstrate overloading of constructor. **[Oct. 2010 - 5M]**
3. Write a program to demonstrate overloading of constructor.
   i. **[Oct. 2011 - 2M]**

```
class sample
{
  public:
  int *ptr;
  sample()      {ptr = NULL;}
  sample(int i)
  {
    ptr = new int[i];
  }
  ~sample()
  {
    delete ptr;
  }
  void printVal()
```

```
    {
      cout<<"The value"<<*ptr;
    }
  };
  void someFunc(sample x)
  {
    cout<<"say i am in someFun"<<endl;
  }
  int main()
  {
    sample s1 = 10;
    someFunc(s1);
    s1.printVal();
  }
```

ii. **[Oct. 2011 – 2M]**

```
  class Test
  {
    public:
    Test();
  };
  Test :: Test()
  {
    cout<<"Constructor called\n";
  }
  int main()
  {
    cout<<"Start\n";
    Test t1();
    cout<<"End\n";
    return 0;
  }
```

4. Write short note on copy constructor. **[Oct. 2011 – 5M]**

7 Operator Overloading And Type Conversion

1. Introduction

PU
Apr. 2010 - *10M*
★ What is operator overloading? Write a program to demonstrate how insertion and extraction operators are overloaded.

Operator overloading allows us to assign additional meaning to most of the standard C++ operators. When overloading operators, it is good practice to ensure that the overloaded operator has a similar behaviour to the original operator. *For example*: It would make more sense to use the + operator for concatenation of strings than the = operator. Overloading operators does not change the precedence and associativity of the operator. New operators cannot be introduced using operator overloading. You can overload operators by creating operator functions. An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**.

The general syntax of operator function is

```
return_type class name :: operator operator_to_be_overloaded(parameters);
```

The keyword **operator** must be preceded by the return type of a function which gives information to the compiler that overloading of operator is to be carried out.

Operator functions can be either members or nonmembers of a class. Nonmember operator functions are always friend functions of the class. However, the way operator functions are written differs between member and nonmember functions. The basic difference between them is that a friend function will have only one argument for unary and binary operators, while a member

function has no arguments for unary operators and one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.

Thus, to declare **operator+** as a friend function of **class X**, the following would be written:

```
friend X operator+(X&, Y&);    // assume X is a class
```

and thereafter, to evaluate the expression **x1 + x2** for two instances of **class X**, the C++ compiler will call the function **operator+(x1, x2)**.

List of the operators that can be overloaded

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus/assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND/assignment	Binary
( )	Function call	
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication/assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment	Unary
+=	Addition/assignment	Binary
–	Subtraction	Binary
–	Unary negation	Unary
– –	Decrement	Unary
– =	Subtraction/assignment	Binary
>	Member selection	Binary

Operator	Name	Type
>*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division/assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift/assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift/assignment	Binary
[ ]	Array subscript	
^	Exclusive OR	Binary
^=	Exclusive OR/assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR/assignment	Binary
||	Logical OR	Binary
~	Ones complement	Unary
delete	delete	
new	new	

List of the operators that can not be overloaded

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional
#	Preprocessor convert to string
##	Preprocessor concatenate
sizeof	object size information
typeid	object type information

Although overloaded operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called:

```
Point pt;
pt.operator+( 3 );   // Call addition operator to add 3 to pt.
```

*The following points will help in designing classes with overloaded operators (assume that **a** and **b** are instances of appropriate class types).*

i. C++ does not "understand" the meaning of an overloaded operator. It is the programmer's responsibility to provide meaningful overloaded functions.

ii. C++ is not able to derive complex operators from simple ones. For instance, if you define overloaded operator functions **operator*** and **operator=**, C++ cannot evaluate the expression **a *= b** correctly.

iii. The programmer can never change the syntax or original meaning of an operator. Operators that are binary must remain binary. Unary operators must remain unary.

iv. The programmer cannot create new operators for use in expressions. Only those operators that are predefined or listed in C++ compiler can be overloaded. However, the programmer can always write functions for special cases.

v. The overloaded operator must have atleast one user defined type operand.

vi. The programmer cannot use friend function to overload certain operators such as Function call operator (), Subscript operator [], Class member access operator -> and Assignment operator =.

vii. The programmer may overload the operators **++** and **- -**.

2. Overloading Unary Operators

Unary operator overloaded by member functions takes no formal arguments, whereas when they are overloaded by friend functions they take a single argument.

2.1 Overloading of Unary Minus Operator

The unary minus or negation (–) operator changes the sign of an operand when applied to a basic data item. We will see how to overload this operator so that it can be applied to an object much the same way as us applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream>
using namespace std;
class sample
{ int a, b;
  public:
```

```
    void getdata(int a, int b);
    void display(void);
    void operator-();     //overload unary minus
}
void sample::getdata(int a, int b)
{ x = a;
  y = b; }
void sample :: display(void)
{ cout << x <<" ";
  cout << y << "\n"; }
void sample :: operator-()
{ x = -x;
  y = -y;}
int main()
{ sample S;
  S.getdata(10,-20);
  cout << "S:";
  S.display();
  -S;      // activates operator-() function
  cout << "S:";
  S.display();
  return 0;
 }
```

Output

S : 10 –20

S : –10 20

The function operator–( ) takes no argument and it changes only the sign of data members of the object S.

Since this function is a member function of the same class, it can directly access the members of the object which activated it.

The statement like S2 = –S1; will not work because, the function operator–( ) does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator-(sample &s);   //declaration
void operator-(sample &s);          //definition
{ s.x = -s.x;
  s.y = -s.y;
  s.z = -s.z; }
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator–(). Therefore, the changes made inside the operator function will not reflect is called object.

2.2 Overloading the Increment and Decrement Operators

The increment and decrement operators fall into a special category because there are two variants of each:

- Pre-increment and post-increment
- Pre-decrement and post-decrement

When you write overloaded operator functions, it can be useful to implement separate versions for the prefix and postfix versions of these operators. To distinguish between the two, the following rule is observed: The prefix form of the operator is declared exactly the same way as any other unary operator; the postfix form accepts an additional argument of type **int**. *Note:* When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type **int**; specifying any other type generates an error. *The following example shows how to define prefix and postfix increment and decrement operators for the* `Point` *class:*

Program for overloading increment and decrement operators

```
class Point
{ public:
   // Declare prefix and postfix increment operators.
   Point& operator++();       // Prefix increment operator.
   Point operator++(int);     // Postfix increment operator.
   // Declare prefix and postfix decrement operators.
   Point& operator--();       // Prefix decrement operator.
   Point operator--(int);     // Postfix decrement operator.
   // Define default constructor.
   Point() { x = y = 0; }
   // Define accessor functions.
   int x() { return x; }
   int y() { return y; }
  private:
   int x, y; };
// Define prefix increment operator.
Point& Point::operator++()
{  x++;
   y++;
   return *this; //Refer 5 in this chapter
}
// Define postfix increment operator.
Point Point::operator++(int)
{  Point temp = *this;
   ++*this;
   return temp; }
// Define prefix decrement operator.
Point& Point::operator--()
{  x--;
   y--;
   return *this;}
// Define postfix decrement operator.
Point Point::operator--(int)
{  Point temp = *this;
   --*this;
   return temp; }
int main()
{ . . .
  . . .
 }
```

The same operators can be defined in file scope (globally) using the following function heads:

```
friend Point& operator++( Point& )       // Prefix increment
friend Point& operator++( Point&, int )  // Postfix increment
friend Point& operator--( Point& )       // Prefix decrement
friend Point& operator--( Point&, int )  // Postfix decrement
```

3. Overloading Binary Operators

To declare a binary operator function as a non-static member, you must declare it in the form:

```
ret_type operatorop( arg )
```

where, *ret-type* is the return type, *op* is the operator to be overloaded and *arg* is an argument of any type. To declare a binary operator function as a global function, you must declare it in the form:

```
ret_type operatorop( arg1, arg2 )
```

where, *ret-type* and *op* are as described for member operator functions and *arg1* and *arg2* are arguments. At least one of the arguments must be of class type.

Note: There is no restriction on the return types of the binary operators; however, most user-defined binary operators return either a class type or a reference to a class type. The following example overloads the **+** operator to add two complex numbers and returns the result.

```
#include<iostream>
using namespace std;
class complex
{    public:
     complex() { }     // constructor 1
     complex(float r, float i)
     { x = r; y =i; }
     complex operator+(complex);
     void display(void)
     private:
     float x, y; };
// Operator overloaded using a member function
complex complex::operator+(complex c)
{ complex temp;            // temporary
  temp.x = x + c.x;        // float additions
  temp.y = y + c.y;
  return(temp); }
void complex :: display(void)
{ cout << x << "+j" << y << "\n"; }
int main()
{ complex a = complex(1.2, 3.4);
  complex b = complex(5.6, 4.8); complex c;
  c = a + b;
  cout << "a = ";
  a.display();
  cout << "b = ";
```

```
  b.display();
  cout << "c = ";
  c.display();
  return 0;
}
```

Output

a = 1.2 + j3.4
b = 5.6 + j4.8
c = 6.8 + j8.2

Overloading Binary Operators using Friends

In many cases, whether you overload an operator by using a friend or a member function makes no functional difference. In those cases, it is usually best to overload by using member functions. However, there is one situation in which overloading by using a friend increases flexibility of an overloaded operator. Let's examine this case now. As you know, when you overload a binary operator by using a member function, the object on the left side of the operator generates the call to the operator function. Further, a pointer to that object is passed in **this** *(Refer 5)* pointer. Now assume some class that defines a member operator+() function that adds an object of the class to an integer. Given an object of that class called obstacle, the following expression is valid:

```
Ob + 100    //valid
```

In this case, Ob generates the call to the overloaded + function and the addition is performed. But what happens if the expression is written like this?

```
100 + Ob    //invalid
```

In this case, it is the integer that appears on the left. Since an integer is a built in type, no operation between an integer and an object of Ob's type is defined. Therefore, the compiler will not compile this expression. The solution to the preceding problem is to overload addition using a friend, not a member function. When this is done, both arguments are explicitly passed to the operator function. Therefore, to allow both object+integer and integer+object, simply overload the function twice – one version for each situation. Thus, when you overload an operator by using two friend functions, the object may appear on either the left or right side of the operator. This program illustrates how friend functions are used to define an operation that involves an object and built in type:

```
#include<iostream>
using namespace std;
class loc
{ int longitude, latitude;
  public:
    loc()   { }
    loc(int lg, int lt)
    { longitude   = lg;
      latitude = lt;   }
```

```
  void show()
  {
    cout << longitude << " ";
    cout << latitude << "\n";
  };
  friend loc operator +(loc op1, int op2);
  friend loc operator +(int op1, loc op2);
 // + is overloaded for loc + int
loc operator +(loc op1, int op2)
{ loc temp;
  temp.longitude= op1.longitude + op2;
  temp.latitude = op1.latitude + op2;
  return temp; }
// + is overloaded for int + loc
loc operator +(int op1, loc op2)
{ loc temp;
  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;
  return temp; }
int main()
{ loc ob1(10,20), ob2(5,30), ob3(7,14);
  ob1.show();
  ob2.show();
  ob3.show();
  ob1 = ob2 + 10; //both of these are valid
  ob3 = 10 + ob2;
  ob1.show();
  ob3.show();
  return 0;
 }
```

4. Limitations of Operator Overloading

i. The overloading of operators is only available for classes; you cannot redefine the operators for the predefined simple types. This would probably be very silly since the code could be very difficult to read if you changed some of them around.

ii. The logical and "&&" and the logical or "||" operators can be overloaded for the classes you define, but they will not operate as short circuit operators. All members of the logical construction will be evaluated with no regard concerning the outcome. Of course the normal predefined logical operators will continue to operate as short circuit operators as expected, but not the overloaded ones.

iii. If the increment "++" or decrement "--" operators are overloaded, the system has no way of telling whether the operators are used as preincrement or postincrement (or predecrement or postdecrement) operators. Which method is used is implementation dependent, so you should use them in such a way that it doesn't matter which is used.

5. "this" Pointer

When a member function is called, it is automatically passed as an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**.

It is a common knowledge that C++ keeps only one copy of each member function and the data members can allocate memory for all of their instances. These kinds of various instances of data are maintained using **this** pointer.

Syntax

```
class_name *this;
```

this pointer is initialized to point to the object for which the member function is invoked. **this** pointer is most useful when working with pointers and especially with a linked list when you need to reference a pointer to the object you are inserting into the list. The keyword **this** is available for this purpose and can be used in any object. Actually the proper way to refer to any variable within a list is through use of the predefined pointer **this**, by writing **this->variable_name**, but the compiler assumes the pointer is used, and we can simplify every reference by omitting the pointer.

Look at the following example to understand how 'this' pointer is used.

```
#include<iostream>
using namespace std;
class pwr{
   double b;
   int e;
   double val;
   public:
      pwr(double base, int exp);
      double get_pwr()
      {return val; } };
pwr :: pwr(double base, int exp)
{  b = base;
   e = exp;
   val = 1;
   if(exp == 0)
      return;
   for( ; exp > 0; exp--)
      val = val * b; }
int main ()
{  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
   cout << x.get_pwr() << " ";
   cout << y.get_pwr() << " ";
   cout << z.get_pwr() << " ";
   return 0;
 }
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside pwr(), the statement

```
b = base;
```

means that the copy of b associated with the invoking object will be assigned the value contained in base. However, the same statement can also be written like this:

```
this ->b = base;
```

The **this** pointer points to the object that invoked pwr(). Thus this–>b refers to that object's copy of b. *For example*: If pwr() had been invoked by x(as in x(4.0, 2)) then **this** in the preceding statement would have been pointing to x. Writing the statement without using **this** is really just shorthand.

Here is the entire pwr() constructor written using the **this** pointer:

```
pwr :: pwr(double base, int exp)
{ this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for(; exp>0;exp--)
    this->val = this->val * this->b; }
```

Actually, no C++ programmer would write pwr() as just shown because nothing is gained, and the standard form is easier. However, **this** pointer is very important when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

this pointer is automatically passed to all member functions. Therefore, get_pwr() could also be rewritten as shown below:

```
double get_pwr() {return this->val;}
```

In this case, if get_pwr() is invoked like this:

```
y.get_pwr();
```

then **this** will point to object **y**.

Some points regarding "this"

i. this pointer stores the address of the class instance, to enable pointer access of the members to the member functions of the class.
ii. this pointer is not counted for calculating the size of the object.
iii. this pointers are not accessible for static member functions.
iv. this pointers are not modifiable.
v. As friend functions are not members of a class, therefore are not passed as this pointer.

6. Overloading << and >> Operators

6.1 Overload the Insertion Operator

One of the features of the I/O stream classes provided in C++ is the capability to overload the insertion so that it can be used with instances of user-defined classes. In other words, the goal is to be able to output instances of user-defined types just as though they were instances of primitive types. This can be done because the insertion operator, which has already been overloaded by the ostream class, may be overloaded again by you so that you may then use the instance cout in the normal fashion. Certainly, if you failed to overload the operator, the compiler would have no idea what to do except generate a compilation error.

The general format for the function declaration and definition are:

```
// Declaration
friend  ostream& operator<<(ostream&, const X&);
//Definition
ostream& operator<<(ostream& stream, const X& obj)
{ //output fields of the object using 'obj;' and the dot operator. Then
. . .
  return stream; }
```

Since you want this function to be invoked by the instance cout and not by an instance of the user-defined class x, it must be declared as a binary friend of the class x. The first argument is a reference to the first argument of the function call (usually cout), and is returned by reference so that function calls can be chained together. The second argument must be an instance of the user-defined class x, and should be passed in by constant reference. Here is a complete example that uses the complex class to add two numbers together. Then the overloaded operator<<() friend function is used to display all three numbers.

```
#include<iostream>
using namespace std;
class complex
{ double real, img;
  public:
    complex(double = 0.0, double = 0.0);
    friend complex operator+(const complex&, const complex&)
    friend ostream& operator<<(ostream&, const complex&);
};
inline complex::complex(double r, double i) { real = r; img = i;}
complex operator+(const complex& c1, const complex& c2)
{complex c;
c.real = c1.real + c2.real;
c.img = c1.img + c2.img;
return c; }
ostream& operator<<(ostream& stream, complex& c)
{ return stream<<c.real<<(c.img < 0 ? " " : "+") << c.img <<"i";}
int main()
{ complex c1(3, -4), c2(2);
  complex c3 = c1 + c2;
  cout << "C1=" << c1 <<"\n";
  cout << "C2=" << c2 <<"\n";
  cout << "C3=" << c3 <<"\n";
  return 0;
}
```

Output

C1 = 3–4i

C2 = 2+0i

C3 = 5–4i

6.2 Overloading Extraction Operator

We have already seen how an insertion operator is overloaded for a user defined class. In a similar fashion, the extraction operator can also be overloaded so that instances of a class can be input just like the primitive types. The general formats for the function declaration and definition for some generic class x are:

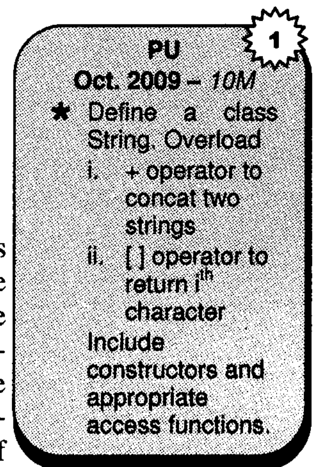
```
// Declaration
friend  istream& operator>>(istream&, X&);
//Definition
istream& operator>>(istream& stream, X& obj)
{ //input into the fields of the object using 'obj' and the dot
operator.then..
  return stream; }
```

The first argument must be a reference to the class istream, and the second must be a reference to an instance of the user defined class. In the following example, both the extraction and the insertion operators have been overloaded.

```
#include<iostream>
using namespace std;
class complex
{ double real, img;
  public:
    complex(double = 0.0, double = 0.0);
    friend complex operator+(const complex&, const complex&)
    friend istream& operator>>(istream&, complex&);
    friend ostream& operator<<(ostream&, complex&);};
inline complex :: complex(double r, double i) {real =r; img = i;}
complex operator+(const complex& c1, const complex& c2)
{complex c;
c.real = c1.real + c2.real;
c.img = c1.img + c2.img;
return c; }
istream& operator>>(istream& stream, complex& c)
{ return stream>>c.real>>c.img;  }
ostream& operator<<(ostream& stream, complex& c)
{ return stream<<c.real<<(c.img < 0 ? " " : "+") << c.img <<"i";}
int main()
{ cout<<"Enter the data for 2 complex numbers\n";
  complex c1, c2;
  cin >> c1>>c2;
  if(!cin)
    cerr<<"Input error\n";
  else
  { complex c3 = c1 + c2;
    cout << "C1=" << c1 <<"\n";
    cout << "C2=" << c2 <<"\n";
    cout << "C3=" << c3 <<"\n";  }
    return 0;
}
```

Output

Enter the data for 2 complex numbers
3 –4 2 1
C1 = 3–4i
C2 = 2+1i
C3 = 5–3i   Enter the data for 2 complex numbers
3 4 2 A
Input error



7. Manipulation of String

C++ has two methods for representing strings, C-style character arrays and the string class. C-style character arrays are a low-level, primitive representation of string data. Although the string class provides more functionality and is less error prone in use, it is not uncommon to see C-style character arrays in C++ code, so it is important to understand and be able to use this representation. This representation is also used in C++ code to handle command line arguments. The string class, which is part of the C++ standard library, provides methods for easy manipulation of string data.

7.1 C-Style Character Arrays

As implied by its name, C-style character arrays is the representation of string data used in the C programming language. In C, this is the only technique for storing and manipulating string data. Strings are stored as null, '\0', terminated character arrays. This representation has several weaknesses and should generally be avoided in C++ programming in favor of the use of the string class.

i. A character array of sufficient size must be defined or allocated to hold the string. The array must be at least the length of the string plus one. One byte is needed to hold the null terminator. It is the programmer's responsibility to be certain that the array is large enough. The compiler will not issue any warnings or errors if the size is too small. Errors in array size will result in run-time errors. The program may crash, behave erratically or operate incorrectly.
ii. At times, it is necessary to explicitly add the null terminator.
iii. Pointers are commonly needed and used to access and manipulate string data.
iv. When copying strings, the programmer must check that the destination array is large enough. When adding to strings, again, the array size must be considered.

Additionally, the string class was not part of the library of early versions of C++. Programmers typically built their own string classes or used C-style strings. In either case, knowledge of C-style strings in necessary. The final reason to learn this representation is that it is used in C++ to handle command line arguments.

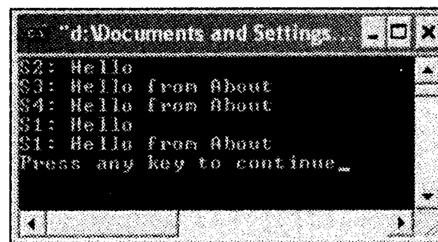
7.2 String Class

The C++ Standard Library provides a string class. To use this class you must include the string include file in your program. `#include<string>`

Unlike C-style strings, the internal representation of the string data is hidden by the string class. The data is set, accessed and manipulated using the methods of the string class. The programmer need not be concerned with how or where the string is stored. This is an example of encapsulation. Let's see how to create and initialize string objects by studying a simple program.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{   string S1;                  // Default constructor, no initialization
    string S2("Hello");         // Initialization with a C-Style string
    string S3("Hello from About"); // Initialization with a C-Style string
    string S4(S3);             // Initialization with another string object
    cout << "S2: " << S2 << endl;
    cout << "S3: " << S3 << endl;
    cout << "S4: " << S4 << endl;
    S1 = S2;                   // Assignment of one string to another
    cout << "S1: " << S1 << endl;
    S1 = S3;                   // S1 is resized as required.
                              // No explicit programming required.
    cout << "S1: " << S1 << endl;
    return 0;
}
```

Output



As can be seen by this example, there are several constructors for the string class. The default constructor creates an empty string. Other forms of the constructor take C-style character arrays or other string objects as arguments and initialize the new string with these values. Also, notice that it is possible to assign one string to another directly. With C-style strings, an assignment like this would need to be done in a for loop, or with the standard library function strcpy. **String Operations:** The string class supports relational operators, such as <, <=, ==, !=, >= and > for comparing strings. The "+" operator is used for string concatenation. The subscript operator, [], can be used to access or set individual elements of a string. The string class also has methods; empty, to test for empty strings, and size, to return the string length. Let's see how to use these in a simple program.

```
#include<iostream>
#include<string>
using namespace std;
int main() {   string S1;
    string S2("from ");
    string S3("About");
    string S4;
    if(S1.empty())     { cout << "S1 is empty" << endl;
        cout << "It has length " << S1.size() << endl;    }
    else   { cout << "No it's not" << endl;     }
    S1 = "Hello ";
    cout << "S1 now has length " << S1.size() << endl;
    if(S3 < S1)     // Relational Comparison    {
        cout << "A comes before C" << endl;     }
    S4 = S1 + S2 + S3;   // Concatenation
    cout << "S4: " << S4 << endl;
    S1 += S2;     // Concatenation
    S1 += S3;
    cout << "S1: " << S1 << endl;   //Subscripting
    cout << "S1[0] " << S1[0] << endl;
    cout << "S1[1] " << S1[1] << endl;
    cout << "S1[2] " << S1[2] << endl;
    cout << "S1[3] " << S1[3] << endl;
    cout << "S1[4] " << S1[4] << endl;
    cout << "S1[5] " << S1[5] << endl;
    return 0;
}
```

Output

```
"d:\Documents and Settings...
S1 is empty
It has length 0
S1 now has length 6
A comes before C
S4: Hello from About
S1: Hello from About
S1[0] H
S1[1] e
S1[2] l
S1[3] l
S1[4] o
S1[5]
Press any key to continue_
```

▶ String Class Methods

The string class has many methods for string manipulations such as appending, inserting, deleting, searching and replacing. Some of the more common methods are summarized in the following table:

Method	Use

append(char *pt);
append(char *pt, size_t count);
append(string &str, size_t offset,size_t count);
append(string &str);
append(size_t count, char ch);
append(InputIterator Start, InputIterator End);
	Appends characters to a string from C-style strings, char's or other string objects.

at(size_t offset);	Returns a reference to the character at the specified position. Differs from the subscript operator, [], in that bounds are checked.
begin();	Returns an iterator to the start of the string.
*c_str();	Returns a pointer to C-style string version of the contents of the string.
clear();	Erases the entire string.
copy(char *cstring, size_t count, size_t offset);	Copies "count" characters from a C-style string starting at offset.
empty();	Test whether a string is empty.
End();	Returns an iterator to one past the end of the string.

erase(iterator first, iterator last);
erase(iterator it);
erase(size_t pos, size_t count);
	Erases characters from the specified positions.

find(char ch,size_t offset = 0);
find(char *pt,size_t offset = 0);
find(string &str,size_t offset = 0);
	Returns the index of the first character of the substring when found. Otherwise, the special value "npos" is returned.

Find_first_not_of();	Same sets of arguments as find. Finds the index of the first character that is not in the search string.
Find_first_of();	Same sets of arguments as find. Finds the index of the first character that is in the search string.
Find_last_not_of();	Same sets of arguments as find. Finds the index of the last character that is not in the search string.
Find_last_of();	Same sets of arguments as find. Finds the index of the last character that is in the search string.

insert(size_t pos, char *ptr);
insert(size_t pos, string &str);
insert(size_t pos, size_t count, char ch);
insert(iterator it, InputIterator start,
InputIterator end);
	Inserts characters at the specified position.

push_back(char ch);	Inserts a character at the end of the string.

replace(size_t pos, size_t count, char *pt);
replace(size_t pos, size_t count, string &str);
replace(iterator first, iterator last, char *pt);
replace(iterator first, iterator last, string &str);
	Replaces elements in a string with the specified characters. The range can be specified by a start position and a number of elements to replace, or by using iterators.

Size();	Returns the number of elements in a string.
swap(string &str);	Swaps two strings.

Using String Methods: The following program illustrates the use of some of the methods of the string class.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{   string S1("Hello World");
    string S2("from About");
    string S3("Bye");
    char c1[] = "Universe";
    size_t index;
    cout << S1 << endl;
    index = S1.find('W');    //Returns an iterator to W in World
    S1.erase(index,1);       //S1 is now "Hello orld"
    cout << S1 << endl;
    string::iterator begin = S1.begin() + S1.find_last_of('o');
    string::iterator end = S1.end();
    S1.erase(begin,end);     //Erases "orld"
    cout << S1 << endl;
    S1.append(c1);           // Appends "Universe"
    cout << S1 << endl;
    S1.clear();              // Clears S1;
    if(S1.empty())
    {   cout << "S1 has been cleared" << endl;
        cout << "Its size is " << S1.size() << endl;   }
    S1.push_back('H');       //H is a char
    S1.insert(1,"ello");     //"ello" is a C-style string.
    cout << S1 << endl;
    S1 += ' ';               //Add a space
    S1 += S2;                // Concatenate S1 and S2
    cout << S1 << endl;
    begin = S1.begin() + S1.find("About");
    end = S1.end();
    S1.replace(begin,end,"John");
    cout << S1 << endl;
    S1.swap(S3);
    cout << S1 << endl;
    return 0;
}
```

8. Type Conversion

While dealing with assignment statements and calculations, data types can sometimes be converted to other data types through something that is called type conversion.

i. **Implicit Conversion:** Conversion that happens without the programmer specifically asking for it is called as implicit conversion. Programmers run into problems during implicit conversions because they can be unaware of what is actually happening during execution of their program until they examine the code very carefully. *For example*: If y and z are float, and a is int, when the statement z = a + y is executed, the value of a will be stored in a temporary memory location as float for use in the rest of the statement. Then, the addition is done (in float form) and finally, the result stored in z. So what happened? [a was implicitly converted into float during the execution of the statement]. Another form of implicit conversion may take place during assignment, but not necessarily from a smaller type to a larger type. **Converting by assignment:** It is a usual way of converting a value from one data type to another by using the assignment operator (=).

> 1
> **PU**
> **Oct. 2009 – *5M***
> ★ How do you achieve the following?
> i. Conversion between objects and basic types.
> ii. Conversion between objects of different classes. Illustrate with the help of an example.

For example:
```
int p;
float q;
```

In the above example, p is an integer type and q is a floating point data type and we have to show that whether p=q or not. This means that we can convert a value from one data type to another just by assigning a float variable's value to a double value variable, a char variable's constant to an int variable or an int variable to a float variable. In C++, converting by assignment operator is not recommended to the programmer, as it will truncate the fractional or real parts and one may not get the desired results. To avoid this, there is a special way of converting one data type to the other, by using the cast operator.

For example:
```
int a;
float b = 3.5;
float c = 5.0;
int d = 2;
a = b * c / d;
cout << a;
```

Output: 8

In the above example first of all, d was implicitly converted to 2.0 during b * c / d. The result of that multiplication and division is 8.75, but a was declared int, so a was implicitly converted to 8 during assignment.

ii. **Explicit Conversion:** Conversion requested by the programmer; sometimes called as typecasting or explicit conversion. While programmers are unaware of implicit conversions, they are completely in control of explicit conversions.

Typecasting can be easily accomplished and it has an easy to use form which is:

```
type(variable) Or type(expression)
```

where, type is whatever data type you choose and variable is the variable you are converting. It simply takes the value of variable and converts it into the type specified.

Example 1

```
    double d;
int k;
d = 12.78;
k = int(d);        --> k = 12
d = double(k);     --> d = 12.0
```

Example 2

```
   int m = 10, n = 4, r;
double q;
q = m / n;            --> q = 2.0
r = double(m) / n;--> r = 2
r = m / n;            --> r = 2
q = double(m) / n --> q = 2.5
```

Solved Programs

1. Create a class FLOAT that contains one float data member. Overload all four arithmetic operators so that they operate on the objects of FLOAT.

Solution

```
#include<iostream>
using namespace std;
class FLOAT { private:
   float number;
   public:
   FLOAT operator + (FLOAT n1) {   FLOAT t;
t.number = number + n1.number;
return t; }
FLOAT operator - (FLOAT n1) {   FLOAT t;
   t.number = number - n1.number;
   return t; }
   FLOAT operator * (FLOAT n) { FLOAT t;
   t.number = number * n.number;
   return t;}
   FLOAT operator / (FLOAT num){   FLOAT t;
   t.number = number / num.number;
   return t; }
   void show() {cout << number; };
   void in(){cout << "Enter a number:";
   cin>>number;} };
   void main(){ FLOAT ob1, ob2, ob3;
   ob1.in()
   ob2.in();
   ob3 = ob1 + ob2;
   ob3.show();
   ob3 = ob1-ob2;
   ob3.show();
   ob3 = ob1 * ob2;
   ob3.show();
   ob3 = ob1/ob2;
   ob3.show();
}
```

1 PU Oct. 2009 – *10M*

2. Define a class string. Overload

i. + operator to concat two strings   ii. [ ] operator to return i^{th} character

Include constructors and appropriate access functions.

Example 1

```
   double d;
int k;
d = 12.78;
k = int(d);          --> k = 12
d = double(k);       --> d = 12.0
```

Example 2

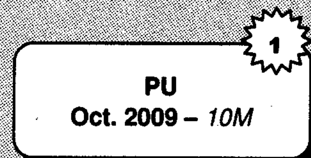
```
   int m = 10, n = 4, r;
double q;
q = m / n;          --> q = 2.0
r = double(m) / n;--> r = 2
r = m / n;          --> r = 2
q = double(m) / n --> q = 2.5
```

Solved Programs

1. Create a class FLOAT that contains one float data member. Overload all four arithmetic operators so that they operate on the objects of FLOAT.

Solution

```
#include<iostream>
using namespace std;
class FLOAT { private:
   float number;
   public:
   FLOAT operator + (FLOAT n1) {  FLOAT t;
t.number = number + n1.number;
return t; }
FLOAT operator - (FLOAT n1) {  FLOAT t;
   t.number = number - n1.number;
   return t; }
   FLOAT operator * (FLOAT n) { FLOAT t;
   t.number = number * n.number;
   return t;}
   FLOAT operator / (FLOAT num){  FLOAT t;
   t.number = number / num.number;
   return t; }
   void show() {cout << number; };
   void in(){cout << "Enter a number:";
   cin>>number;} };
   void main(){ FLOAT ob1, ob2, ob3;
   ob1.in()
   ob2.in();
   ob3 = ob1 + ob2;
   ob3.show();
   ob3 = ob1-ob2;
   ob3.show();
   ob3 = ob1 * ob2;
   ob3.show();
   ob3 = ob1/ob2;
   ob3.show();
}
```



2. Define a class string. Overload

i. + operator to concat two strings ii. [ ] operator to return i^{th} character

Include constructors and appropriate access functions.

Solution

i. + Operator to concat two strings

```
#include<iostream.h>
#include<conio.h>
class test1
{
  private:
    char str1[11];
    char str2[22];
  public:
    test1()
        {
    }
    test1(char c1, char c2)
    {
      str1 = c1;
      str2 = c2;
    }
    test1 operator +(test1);
    void display(void);
};
test1 test1 :: operator +(test1 c)
{
  test1 temp;
  temp.str1 = str1+c.str1;
  temp.str2 = str2 + c.str2;
  return(temp);
}
void test1 :: display(void)
{
  cout << str1 << " + Str2" << str2 << "\n";
}
void main()
{
  test1 c1, c2,c3;
  clrscr();
  c1 = test1("Sita", "Geeta");
  c2 = test1("Meena", "Teena");
  c3 = c1 + c2;
  cout << "\n Str1 ";
  c1.display();
  cout << "\n Str2 = ";
  c2.display();
  cout << "\n Combine ";
  c3.display();
  getch();
}
```

ii. [ ] operator to return i^th^ character

```
#include<iostream.h>
#include<conio.h>
class myclass
{
```

```
        int num;
        int a[10];
     public:
        myclass(int num);
        int operator [] (int);
  };
  myclass :: myclass(int n)
  {
     num = n;
     for(int i =0;i<num;i++)
     {
        cout << "\n Enter value for element" << i+1<< "  ";
        cin >> a[i];
     }
  }
  int myclass:: operator[](int index)
  {
     if(index < num)
        return a[index];
     else
        return NULL;
  }
  void main()
  {
     myclass a(3);
     clrscr();
     cout << a[0] << endl;
     cout << a[1] << endl;
     cout << a[3] << endl;
     getch();
  }
```

3. What is operator overloading? Write a program to demonstrate how insertion and extraction operators are overloaded.

PU Apr. 2010 – *10M*

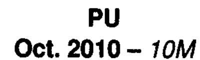
Solution

```
#include<iostream>
#include<string>
class PhoneNumber
{
   friend ostream &operator<<(ostream&, const PhoneNumber&);
   friend istream &operator>>(istream&, PhoneNumber&);
   private:
    string areaCode;
    string phno;
};

  ostream &operator<<(ostream &output,const PhoneNumber&num)
{
   output <<"("<< num.areaCode<<")"<<"-"<< num.phno;
   return output;     //enables cout<<a<<b;
}
  istream &operator>>(istream &input,PhoneNumber&num)
  {
```

```
    input.ignore();  //skip one char, the'('input >>
                          setw(4)>>num.areaCode;
                     //input area code input.ignore();// skip)
    input.ignore();  // skip dash(-)
    input >> setw(8) >> num.phno;//input phno
    return input;    // enables cin >> a >> b;
  }
int main()
{
   PhoneNumber p;
   Cout<<"Enter phone number in the form(0241)-12345678:\n";
   cin >> p;
   cout<<"The phone number entered was: "<<p<<endl<<endl;
   return 0;
}
```

4. Write a program to overload increment and decrement operators.



Solution

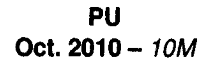
/* Program to overload increment and decrement operators */

/* In following program Digit class holds a number between 0 and 9. We have overloaded increment and decrement so they increment/decrement the digit, wrapping around if the digit is incremented/decremented out range. */

```
#include<iostream.h>
class Digit
{
  private:
       int m_nDigit;
  public:
       Digit(int nDigit=0)
       {
         m_nDigit = nDigit;
       }
       Digit& operator++(); // prefix
       Digit& operator--(); // prefix
       Digit operator++(int); // postfix
       Digit operator--(int); // postfix
      int GetDigit()const
     {
       return m_nDigit;
     }
};
Digit& Digit::operator++()
{
   //If our number is already at 9, wrap around to 0
   if(m_nDigit == 9)
       m_nDigit = 0;
   // otherwise just increment to next number
    else
```

```
    input.ignore();  //skip one char, the'('input >>
                           setw(4)>>num.areaCode;
                     //input area code input.ignore();// skip)
    input.ignore();  // skip dash(-)
    input >> setw(8) >> num.phno;//input phno
    return input;    // enables cin >> a >> b;
  }
int main()
{
   PhoneNumber p;
   Cout<<"Enter phone number in the form(0241)-12345678:\n";
   cin >> p;
   cout<<"The phone number entered was: "<<p<<endl<<endl;
   return 0;
}
```

4. Write a program to overload increment and decrement operators.



PU
Oct. 2010 – *10M*

Solution

/* Program to overload increment and decrement operators */

/* In following program Digit class holds a number between 0 and 9. We have overloaded increment and decrement so they increment/decrement the digit, wrapping around if the digit is incremented/decremented out range. */

```
#include<iostream.h>
class Digit
{
  private:
       int m_nDigit;
  public:
       Digit(int nDigit=0)
       {
         m_nDigit = nDigit;
       }
       Digit& operator++(); // prefix
       Digit& operator--(); // prefix
       Digit operator++(int); // postfix
       Digit operator--(int); // postfix
      int GetDigit()const
     {
       return m_nDigit;
     }
};
Digit& Digit::operator++()
{
   //If our number is already at 9, wrap around to 0
   if(m_nDigit == 9)
       m_nDigit = 0;
   // otherwise just increment to next number
    else
```

```
        ++m_nDigit;
    return *this;
}
Digit& Digit::operator--()
{
   //If our number is already at 0, wrap around to 9
   if(m_nDigit == 0)
      m_nDigit = 9;
   // otherwise just decrement to next number
     else
        --m_nDigit;
     return *this;
}
Digit Digit::operator++(int)
{
  //Create a temporary variable with our current digit
    Digit cResult(m_nDigit);

     // Use prefix operator to increment this digit
     ++(*this);  // apply operator

     // return temporary result
     return cResult; // return saved state
}
Digit Digit::operator--(int)
{
   //Create a temporary variable with our current digit
     Digit cResult(m_nDigit);
  // Use prefix operator to increment this digit
     --(*this);      // apply operator

     // return temporary result
     return cResult;  // return saved state
}
int main()
{
  Digit cDigit(5);
  ++cDigit; // calls Digit::operator++();
  cDigit++; // calls Digit::operator++(int);
  return 0;
}
/*    */
```

5. Write outputs with explanation:

PU
Oct. 2011 – *2M*

```
class Test
{
    public:
    int operator==(Test t)
    {
      if(*this == t)
      {
        cout<<"Both are same";
        return 1;
      }
      else
      {
        cout<<"Both are different";
        return 0;
      }
    }
  };
void main()
{
    Test t1,t2;
    t1 == t2;
}
```

Solution

The above code would go in an infinite loop. The above program is based on the operator overloading. The statement t1 == t2 will invoke the operator==(Test t) function. In this function the statement if(*this == t) will invoke the same function because *this is nothing but t1 and t is the copy of t2.

6. Write the stack class and overload + and – operators for push and pop operation. The stack class should throw an exception when the stack underflow and overflow takes place.

PU
Oct. 2011 – *10M*

Solution

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define MAX 5
class Stack
{
  int arr[MAX], top;
  public:
  Stack()
  { top = -1; }
void operator+()
{
  if(top == MAX)
  { cout<<"\n Stack Overflow..."; }
  else
  {
    int ele;
    cout<<"\n Enter an element : ";
    cin>>ele;
```

```
    top++;
    arr[top] = ele;
  }
}
  void operator-()
  {
    if(top < 0)
    { cout<<"\n Stack underflow..."; }
    else
    {
      cout<<"\nThe element "<<arr[top]<<" is removed.";
      top--;
    }
  }
  void display()
  {
    int i;
    cout<<"\n Stack elements are : \n";
    for(i = 0; i <= top; i++)
    { cout<<arr[i]<<"\n";   }
  }
};
int main()
{
  Stack s1;
  int ch;
  do
  {
    cout<<"\n------------------";
    cout<<"\n1.Push";
    cout<<"\n2.Pop";
    cout<<"\n3.Display";
    cout<<"\n4.Exit";
    cout<<"\n------------------";
    cout<<"\n Enter choice : ";
    cin>>ch;
    switch(ch)
    {
      case 1:
        +s1; break;
      case 2:
        -s1; break;
      case 3:
        s1.display(); break;
      case 4:
        cout<<"\nPress any key to exit.";
        getch();
        exit(0);
    }
  }while(ch < 5 && ch > 0);
  return 0;
}
```

EXERCISES

A. Review Questions

1. What is meant by operator overloading?
2. List the C++ operators that cannot be overloaded?
3. Explain the limitations of operator overloading.
4. Explain the rules of operator overloading.
5. Explain how the preincrement and postincrement operators are overloaded.

B. Programming Exercises

1. Write C++ program using operator overloading to find factorial of given number.
2. Write a C++ program to perform the following using operator overloading
   i. Area of circle ii. Circumference of a circle iii. Area of rectangle
   iv. Area of a triangle v. Perimeter of square

Collection of Questions asked in Previous Exams PU

1. How do you achieve the following? **[Oct. 2009 – *5M*]**
   i. Conversion between objects and basic types.
   ii. Conversion between objects of different classes. Illustrate with the help of an example.
2. Define a class String. Overload i. + operator to concat two strings
   ii. [ ] operator to return i^{th} character
   Include constructors and appropriate access functions. **[Oct. 2009 – *10M*]**
3. What is operator overloading? Write a program to demonstrate how insertion and extraction operators are overloaded. **[Apr. 2010 – *10M*]**
4. Write a program to overload increment and decrement operators. **[Oct. 2010 – *10M*]**
5. Write outputs with explanation: **[Oct. 2011 – *2M*]**

```
class Test
{
     public:
     int operator==(Test t)
     {
        if(*this == t)
        {
           cout<<"Both are same";
           return 1;
        }
        else
        {
           cout<<"Both are different";
           return 0;
        }
     }
  };
void main()
{
     Test t1,t2;
     t1 == t2;
}
```

6. Write the stack class and overload + and – operators for push and pop operation. The stack class should throw an exception when the stack underflow and overflow takes place. **[Oct. 2011 – *10M*]**

8 Inheritance

I. Introduction

Inheritance is said to be the feature that distinguishes object-oriented programming languages (OOP's) from other kinds of languages. Inheritance is often described in terms of "is-a" relationships and supports the concept of hierarchical classification. The mechanism of creating new classes from an existing class is called as **inheritance**. The old class or existing class is known as **base class or parent class** and the newly created class is called as **derived class or child class**. In some OOP languages such as Simula, the base and derived classes are called as super and subclasses respectively.

You can again inherit from a derived class, making this class the base class of the new derived class. This leads to a hierarchy of base class/derived class relationships. If you draw this hierarchy you get an *inheritance graph.*

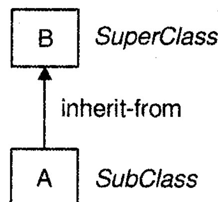


Figure 8.1: A simple (single) inheritance graph

In the *figure 8.1,* class A inherits from class B, so B is called **superclass** of A and A is called **subclass** of B.

A common drawing scheme is to use arrowed lines to indicate the inheritance relationship between two classes or objects. In our examples we have used "inherit-from". Consequently, the arrowed line starts from the subclass towards the superclass as illustrated in the above *figure 8.1* (some authors show the arrow in opposite direction meaning "inherited from". In C++ the word ``inherit-from" are replaced by a colon.

The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have all the features of the existing class (base class). It can also add some more features to this class.

The main advantages of inheritance are:

- reusability of the code,
- to increase the reliability of the code and
- to add some enhancements to the base class.

PU
Apr.10,Oct.10 – *10M*
★ Explain different types of inheritance with suitable examples of each type.

2. Single Inheritance

Single inheritance is the process of creating new classes from an existing base class. The existing class is known as the direct base class and the newly created class is called as a singly derived class. In case of single inheritance there is only one base class (*Refer figure 8.1*).

Single inheritance is the ability of a derived class to inherit the member functions and variables of the existing base class.

2.1 Defining the Derived Class

The declaration of a singly derived class is same as that of an ordinary class.

The general syntax of the derived class declaration is as follows:

```
class derived-class-name : private/public/protected base_class-
name
{  private:
      // data members
   public:
      // data members
      // methods
   protected:
      // data members
};
```

where, the colon indicates that the derived-class-name is derived from the base-class-name.

The base-class access specifier must be either public, private or protected. If no access specifier is present then the access specifier is **private** by default if the derived class is a **class**. If the derived class is **struct**, then **public** is default in the absence of an explicit access specifier.

For example: The following program illustrates the declaration of a single inherited class. The base class consists of two parts: data member and member function. The data members consist of name, rollno and sex and are defined as a private group. The member functions getdata() and putdata() are declared in the base class. The members of a base class can be referred as if they were members of the derived class.

```
class basic_info{
  private:
    char name[20];
    long int rollno;        // private; not inheritable
    char sex;
  public:
    void getdata();
    void display();              // public; ready for inheritance
}; // end of class definition
class physical_fit : public basic_info     // public derivation
{ private:
    float  height;
    float weight;
  public:
    void getdata();
    void display();
}; // end of class definition
```

The derived class inherits the properties of its base class including its data member and member functions. The *physical_fit* is a derived class which has two components– *private* and *public*. In addition to its new data members such as height and weight, it may inherit the data members of the base class. The derived class contains not only the methods of its own but also of its base class.

A program to read the derived class data members such as name, roll number, sex, height and weight from the keyboard and display the contents of the class on the screen.

Program using Single Inheritance

```
#include<iostream>
#include<iomanip>
using namespace std;
class basic_inco{
  private:
    char name[20];
    long int rollno;             // private; not inheritable
    char sex;
  public:
    void getdata();
    void display();              // public; ready for inheritance
}; // end of class definition
class physical_fit : public basic_info   // public derivation
{ private:
    float  height;
    float weight;
  public:
    void getdata();
    void display();
```

```
}; // end of class definition
void basic_info :: getdata()
{ cout << "Enter a name:\n";
  cin >> name;
  cout << "Enter roll no:\n";
  cin>>rollno;
  cout <<"Enter a sex: \n";
  cin>>sex;
}
void basic_info :: display()
{ cout << name   <<  " ";
  cout << rollno <<  " ";
  cout << sex    <<  " ";
}
void physical_fit :: getdata()
{ basic_info :: getdata();
  cout << "Enter height:\n";
  cin>> height;
  cout << "Enter weight:\n";
  cin >> weight;
}
void physical_fit :: display()
{ basic_info :: display();
  cout << setprecision(2);
  cout << height << " ";
  cout << weight << " ";
}
void main()
{ physical_fit a;
  cout << "Enter the following information\n";
  a.getdata();
  cout << " ________________________________________________________\n";
  cout << " Name         Rollno     Sex     Height       Weight \n";
  cout << " ________________________________________________________\n";
  a.display();
  cout <<endl;
  cout << " ________________________________________________________\n";
}
```

Output

Enter the following information
Enter a name : abc
Enter rollno : 13
Enter sex : m
Enter height : 134.54
Enter weight: 70

Name	Rollno	Sex	Height	Weight
abc	13	m	134.54	70

2.2 Types of Base Classes

A base class can be classified into two types, direct base class and indirect base class.

i. Direct Base Class

Types of Base Classes
i. Direct
ii. Indirect

A base class is called a direct base if it is mentioned in the base list.

For example: Following are valid derived class declarations:

1.
```
class base
{ ________
  ________ };
class derived : public base
{ ________
  ________ };
```
where class base is a direct base

2.
```
class baseA
{ ________
  ________ };
class baseB
{ ________
  ________ };
class derivedC : public
baseA, public baseB
{ ________
  ________ };
```
where both classes baseA and baseB are the direct bases.

3. A class may be derived form any number of base classes.
For example
```
class baseA
{ ________
  ________ };
class baseB
{ ________
  ________ };
class baseC
{ ________
  ________ };
class baseD
{ ________
  ________
};
class derivedE : public baseA, public baseB, public baseC,
public baseD
{ ________
  ________
};
```

The following are invalid declarations:

1. A class which has been named but not yet declared cannot be used as a base class.
```
class baseA
class derivedB : public baseA
{ ________
  ________ //error
};
```
The base class baseA is undeclared and an error message will be displayed by the compiler.

2. A class should not be specified as a direct base class of a derived class more than once.

```
class baseA
{ ________
  ________
};
class derivedB : public baseA, public baseA
{ ________
  ________
};
```

The base class baseA has been declared twice as the direct base of the derived class derivedB. It is an invalid way of constructing a derived class in C++.

ii. Indirect Base Class

A derived class can itself serve as a base class subject to access control. When a derived class is declared as a base of another class, the newly derived class inherits the properties of its base classes including its data members and member functions. A class is called as an indirect base if it is not a direct base, but is a base class of one of the classes mentioned in the base list.

For example: Following are the valid declarations:

1.
```
class baseA
{ ________
  ________
};
class derivedB : public baseA
{ ________
  ________
};
class derivedC : public derivedB
{ ________
  ________
};
```

Note that the class derivedB is the base of the class derivedC that is called as an indirect base.

2. A class may be specified as an indirect base more than once. *For example*

```
class baseA
{ ________
  ________
};
class baseB : public baseA
{ ________
  ________
};
class  baseC : public baseA
{ ________
  ________
};
class derivedD : public baseB, public baseC
{ ________
  ________
};
```

The class baseA has inherited both the derived classes baseB and baseC, in the sense that inheritance means building new abstractions from the existing ones, where one class inherits data and member functions from another.

2.3 Types of Derivation

We know that inheritance is a process of creating a new class form an existing class. While deriving the new classes, the access control specifier gives the total control over the data members and methods of the base classes. A derived class can be defined with one of the access specifiers, such as private, public and protected.

Types of Base Derivation
i. Public inheritance
ii. Private inheritance
iii. Protected inheritance

i. Public Inheritance

If the access specifier for a base class is public then

- All **public** members in the base class remain **public** in the derived classes.
- All **protected** members in the base class remain **protected** in the derived classes.
- In all cases, the base's **private** elements remain **private** to the base and are not accessible by members of the derived class.

The general **syntax** of the public derivation is

```
class base_class_name
{ .    .    .
  .    .    .
}
class derived_class_name : public base_class_name
{ .    .    .
  .    .    .
}
```

In the following program objects of type derived can directly access the public members of base:

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

```
    derived(int x) {k = x;}
  void showk()
  {
    cout << k << "\n";}
};

int main()
{
  derived ob(3);
  ob.set(1,2);      // access member of base
  ob.show();        // access member of base
  ob.showk;       // uses member of derived class
  return 0;
 }
```

ii. Private Inheritance

When the base class is inherited by using the private access specifier,

- All **public** members in the base class become **private** in the derived classes.
- All **protected** members in the base class become **private** in the derived classes
- All **private** members in the base class remain **private** in the base class and hence are visible only in the base class.

The general **syntax** of the private derivation is

```
class base_class_name
{
   . . .
   . . .
};
class derived_class_name : private base_class_name
{
   . . .
   . . .
};
```

For example: The following program will not even compile because both set() and show() are now private elements of derived:

```
// This program won't compile
#include<iostream>
using namespace std;
class base{
  int i, j;
  public:
    void set(int a, int b)
      { i=a; j=b; }
    void show()
      { cout << i << " " << j << "\n"; }
};
```

```
// public elements of base are private in derived.
class derived : private base{
  int k;
  public:
    derived(int x) {k = x;}
  void showk()
  { cout << k << "\n"; }
};
int main()
{ derived ob(3);
  ob.set(1,2);          // error, can't access set()
  ob.show();           // error, can't access show()
  return 0;
}
```

Note: When a base class access specifier is **private**, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.

iii. Protected Inheritance

If the access specifier for a base class is protected then

- All **public** members in the base class become **protected** in the derived classes.
- All **protected** members in the base class remain **protected** in the derived classes
- All **private** members in the base class remain **private** in the base class and hence it is visible only in the base class.

The general **syntax** of the protected derivation is

```
class base_class_name
{
   . . .
   . . . };
class derived_class_name: protected base_class_name
{
   . . .
   . . . };
```

```
#include<iostream>
using namespace std;
class base{
  protected:
    int i, j;     //private to base, but accessible by derived
  public:
    void setij(int a, int b)
      { i=a; j=b; }
    void showij()
      { cout << i << " " << j << "\n"; }
};
```

```
// inherit base as protected.
class derived : private base{
  int k;
  public:
  // derived may access base's i and j and setij()
  void setk() { setij(10, 12); k = i*j; }
  // may access showij() here
  void showall()
  { cout << k << " "; showij();}
};
int main()
{
  derived ob;
  // ob.setij(2,3);  //illegal, setij() is protected member of derived
  ob.setk();          // ok, public member of derived
  ob.showall();       // ok, public member of derived
  // ob.showij();     // illegal, showij() is protected member of derived
  return 0;
 }
```

As you can see by reading the comments, even though setij() and showij() are public members of base, they become protected members of derived when it is inherited using the protected access specifier. This means that they will not be accessible inside main().

The following table shows the summary of above rule:

Table 8.1: Access rights and Inheritance

Derivation type	Base class member	Access in derived class
private	private	(inaccessible)
	public	private
	protected	private
public	private	(inaccessible)
	public	public
	protected	protected
protected	private	(inaccessible)
	public	protected
	protected	protected

The various functions that can have access to the private and protected members of a class are:

a. A function that is a friend of the class.

b. A member function of a class that is a friend of the class.

c. A member function of a derived class.

2.4 Ambiguity in Single Inheritance

Whenever a data member and member function are defined with the same name in both the base and the derived classes, these names must be without ambiguity. The scope resolution operator (::) may be used to refer to any base member explicitly. This allows access to a name that has been redefined in the derived class.

For example: The following program segment illustrates how ambiguity occurs when the *getdata()* member function is accessed from the *main()* program.

```
class baseA
{ public:
     int i;
     getdata();
};
class baseB
{ public:
     int i;
     getdata();
};
class derivedC : public baseA, public baseB
{ public:
     int i;
     getdata();
}
void main()
{ derivedC obj;
  obj.getdata;
};
```

The members are ambiguous without scope operators. When the member function getdata() is accessed by the class object, naturally, the compiler cannot distinguish between the member function of the class baseA and the class baseB. Therefore it is essential to declare the scope operator explicitly to call a base class member as shown below:

```
obj.baseA :: getdata();
obj.baseB :: getdata();
```

A program to demonstrate how ambiguity is avoided in single inheritance using scope resolution operator.

```
// ambiguity in single inheritance
#include<iostream>
using namespace std;
class baseA {
  private:
     int i;
  public:
     void getdata(int x);
     void display();
};
class baseB{
  private:
     int j;
  public:
     void getdata(int y);
     void display();
};
class derivedC : public baseA, public baseB
{      };
```

```
void baseA :: getdata(int x)
{    i = x; }
void baseA :: display()
{ cout << "value of i =" << i << endl;  }
void baseB :: getdata(int y)
{ j = y; }
void baseB :: display()
{ cout << "value of j =" << j << endl;  }
void main()
{ derivedC objc;
  int x, y;
  cout << "Enter a value for i: \n";
  cin >> x;
  objc.baseA :: getdata(x);
  // member is ambiguous without scope
  cout << "Enter a value for j: \n";
  cin >> y;
  objc.baseB :: getdata(y);
  objc.baseA :: display();
  objc.baseB :: display();
 }
```

Output

Enter a value for i:
50
Enter a value for i:
60
value of i = 50
value of j = 60

3. Multiple Inheritance

Later versions of C++ introduced a "multiple inheritance" model for inheritance. In a multiple-inheritance graph, the derived classes may have a number of direct base classes, i.e., multiple inheritance is the process of creating a new class from more than one base classes.

Consider the following figure in which the derived class C inherits properties of the base classes A and B.

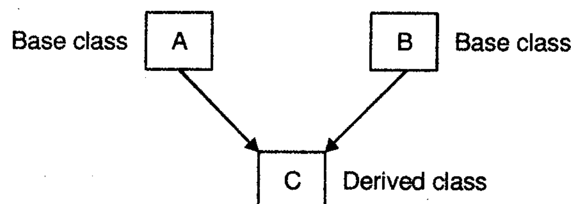


Figure 8.2 : Multiple Inheritance

The syntax of a derived class with multiple base classes is as follows:

```
class derived : access specifier base1, access specifier base2 .
. .
{
    . . .
    . . .    //body of derived class
}
```

For example

```
class point
{ . . .
  . . . };
class string
{ . . .
  . . . };
class drawablestring : public point, public string
{ . . .
  . . . };
```

The class *drawablestring* is derived from both classes *point* and *string*.

Multiple inheritance is a derived class declared to inherit properties of two or more parent classes (base classes). Multiple inheritance can combine the behaviour of multiple base classes in a single derived class. Multiple inheritance has many advantages over the single inheritance such as rich semantics and the ability to directly express complex structures. In C++, derived classes must be declared during the compilation of all possible combinations of derivations and the program can choose the appropriate class at run time and create object for the application.

Difference between Single and Multiple Inheritance is as follows:

In single inheritance, a derived class has only one base class while in multiple inheritance, a derived class has more than one base classes.

In single inheritance hierarchy, a derived class typically represents a specialization of its base class while in multiple inheritance hierarchy a derived class typically represents a combination of its base classes.

The rules of inheritance and access do not change from single to multiple inheritance hierarchy. A derived class inherits data members and methods from all its base classes, regardless of whether the inheritance links are private, protected or public.

Consider the following definition of two base classes

```
class A{    // base class 1
  . . .
  . . .};
class B{    // base class 2
  . . .
  . . .};
```

Form the above definition,

i. multiple inheritance with all public derivation can be defined as

```
class derivedC : public A, public B
{ . . .
  . . . };
```

ii. multiple inheritance with all private derivation can be defined as

```
class derivedC : private A, private B
{ . . .
  . . . };
```

iii. multiple inheritance with all mixed derivation can be defined as

```
class derivedC : private A, public B
{ . . .
  . . . };
```

A program to illustrate how a multiple inheritance can be declared and defined. This program consists of two base classes and one derived class. The base class basic_info contains the data members: name, rollno and sex. Another base class academic_fit contains the data members: course, semester and rank. The derived class financial_assit contains the data member amount besides the data members of the base classes. The derived class has been declared as public inheritance. The member functions are used to get information of the derived class from the keyboard and display the contents of class objects on the screen.

```
#include<iostream>
#include<iomanip>
using namespace std;
class basic_info
{ private:
    char name[30];
    long int rollno;
    char sex;
  public:
    void getdata();
    void display();    }; // end of class definition
class academic_fit
{ private:
    char course[30];
    char semester[10];
    int rank;
  public:
    void getdata();
    void display();    }; // end of class definition
class financial_assit : private basic_info, private academic_fit
{ private:
    float amount;
  public:
    void getdata();
    void display();    }; // end of class definition
void basic_info :: getdata()
{ cout << "Enter name: \n";
  cin >>  name;
  cout << "Enter roll no:\n";
  cin >> rollno;
  cout << "Enter sex:\n"
  cin >> sex;}
void basic_info :: display();
{ cout << name    << " ";
```

```
  cout << rollno   << " ";
  cout << sex      << " ";   }
void academic_fit :: getdata()
{ cout << "Course Name (MBA, MCM,MCS, MCA etc)\n";
  cin >> course;
  cout << "Semester (First/Second etc)\n";
  cin >> semester;
  cout << "Rank of the student \n";
  cin >> rank;     }
void academic_fit :: display();
{ cout << course     << " ";
  cout << semester   << " ";
  cout << rank       << " ";}
void financial_assit :: getdata()
{ basic_info :: getdata();
  academic_fit  :: getdata();
  cout << "Amount in rupees\n";
  cin >> amount;     }
void finanacial_assit :: display()
{
  basic_info :: display();
  academic_fit :: display();
  cout << setprecision(2);
  cout << amount << " ";     }
void main()
{ financial_assit f;
  cout << "Enter the following information for financial assistance";
  f.getdata();
  cout << endl;
  cout << "Academic Performance for Financial Assistance\n";
  cout << "__________________________________________________________\n";
  cout << "Name   Rollno   Sex   Course   Semester   Rank   Amount\n";
  cout << "__________________________________________________________\n";
  f.display();
  cout << endl;
  cout << "__________________________________________________________\n";
}
```

Output

Enter the following information for financial assistance
Enter name
ABC
Enter rollno
23
Enter sex
M
Course Name (MBA, MCM, MCS, MCA etc)

MCS
Semester (First, Second etc)
First
Rank of the student
20
Amount in rupees
1200
Academic Performance for Financial Assistance

Name	Rollno	Sex	Course	Semester	Rank	Amount
ABC	23	M	MCS	First	20	1.2e + 03

Ambiguity in the Multiple Inheritance

To avoid ambiguity between the derived class and one of the base classes or between the base class themselves, it is better to use the scope resolution operator :: along with the data members and methods.

For example: The following program segment illustrates how ambiguity occurs in both base classes and the derived class.

```
// Ambiguity in multiple inheritance
#include<iostream>
using namespace std;
class baseA
{
  public:
    int a;
};
class baseB
{
  public:
    int a;
};
class baseC
{
  public:
    int a;
};
class derivedD : public baseA, public baseB, public baseC
{
  public:
    int a;
};
void main()
{
  derivedD objd;
  objd.a = 10; //local to the derived class
}
```

Suppose one intends to access the data members of the base classes, then conflict occurs between the base classes themselves and the compiler cannot distinguish between the calls. It is upto programmer to avoid such conflicts and ambiguities. Therefore, it is better to use the scope operator to avoid such ambiguities.

```
void main()
{
  derivedD objd;
  objd.a = 10;
  objd.baseA::a = 40;        // accessing the base class A member
  objd.baseB::a = 60;        // accessing the base class B member
  objd.baseC::a = 80;        // accessing the base class C member
}
```

A program to demonstrate how ambiguity is avoided in multiple inheritance using scope resolution operator.

```
// ambiguity in multiple inheritance without scope resolution operator
#include<iostream>
using namespace std;
class baseA
{ public:
    int a;
};
class baseB
{
  public:
    int a;
};
class baseC
{ public:
    int a; };
class derivedD : public baseA, public baseB, public baseC
{ public:
    int a; };
void main()
{ derivedD objd;
  objd.a = 10;
  objd.baseA::a = 20;
  objd.baseB::a = 30;
  objd.baseC::a = 40;
  cout << "Value of a in the derived class =" << objd.a;
  cout << endl;
  cout << "Value of a in the baseA =" << objd.baseA::a;
  cout << endl;
  cout << "Value of a in the baseB =" << objd.baseB::a;
  cout << endl;
  cout << "Value of a in the baseC =" << objd.baseC::a;
  cout << endl;
  cout << endl;
}
```

Output

Value of a in the derived class = 10
Value of a in the baseA = 20
Value of a in the baseB = 30
Value of a in the baseC = 40

4. Multilevel Inheritance

Multilevel inheritance means a class is derived from another derived class. *Figure 8.3* shows class 'A' which serves as a base class for class 'B' which in turn serves as a base class for class 'C'. Class 'B' is called as intermediate base class as it acts as a link between A and C. Chain ABC is called as inheritance path.

Declaration of multilevel inheritance:

```
Class A {  . . . . . . } ;              // Base Class
Class B : public A { . . . . . .} ;       // B derived form A
Class C : public B { . . . . . .} ;     // C derived form B
```

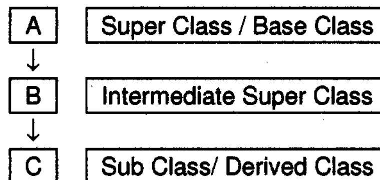


Figure 8.3 : Multilevel Inheritance

Let us consider the simple example. Assume that the test results of a batch of students are stored in three different classes. Class student stores the rollno, class marks stores the marks obtained in three subjects and class result contains the total marks and percentage obtained and access the rollno of students through multilevel inheritance.

```
#include<iostream>
using namespace std;
class stduent{
  protected:
    int rollno;
  public:
    void getno(int);
    void putno(void); };
void student :: getno(int a)
{ rollno = a; }
void student :: putno()
{ cout << "Roll Number: " << rollno << "\n";   }
class marks : public student     // first level derivation
{ protected:
    float m1, m2,m3;
```

```
  public:
     void getmarks(float, float, float)
     void putmarks(void); };
void marks :: getmarks(float x, float y, float z)
{ m1 = x;
  m2 = y;
  m3 = z; }
void marks :: putmarks()
{ cout << "Marks in m1:= " << m1 <<"\n";
  cout << "Marks in m2:= " << m2 <<"\n";
  cout << "Marks in m3:= " << m3 <<"\n";}
class result : public marks    // Second level derivation
{ private:
     float total;
     float percentage;
  public:
     void display(void)
};
void result :: display(void)
{ total = m1 + m2 + m3;
  percentage = total/3;
  putno();
  putmarks();
  cout << "Total:=" << total << "\n";
  cout << "Percentage:=" << percentage << "\n";
}
int main()
{
  result student1;   // object student1 is created
  student1.getno(30);
  student1.getmarks(40.5, 60.0, 80.0);
  student1.display();
  return 0;
 }
```

Output

```
Roll Number   := 30
Marks in m1   := 40.5
Marks in m2   := 60.0
Marks in m3   := 80.0
Total         := 180.5
Percentage    := 60.17
```

5. Hierarchical Inheritance

Hierarchical inheritance comes into picture when certain features of one level are shared by many other levels below that level. As an example, consider a hierarchical classification of accounts in a commercial bank as shown in *figure 8.4.* All accounts possess certain common features.

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the subclasses. A subclass is constructed by inheriting properties of the base class for the lower level classes and so on.

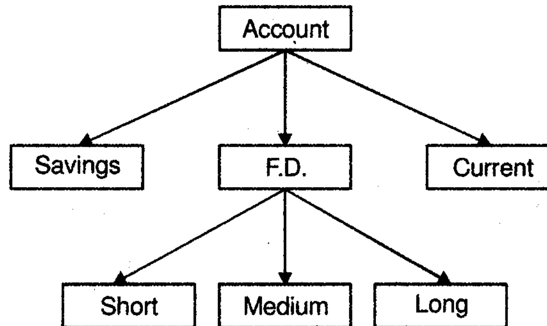


Figure 8.4: Classification of bank accounts (Hierarchical classification)

Consider the following program:

```
#include<iostream>
using namespace std;
class account
{ protected:
     int accno;
     char accname[20];
  public:
     void getdata();
     void displaydata();   };
void account :: getdata()
{ cin >> accno;
  cin >> accname; }
void account :: displaydata()
{ cout << accno << "\n";
  cout << accname << "\n";
};
class withdraw : public account
{ private:
     float amt;
  public:
     void get()
     { getdata();
       cin >> amt;      }
     void display()
     { displaydata();
       cout << amt;       }
};
class balance : public account
{ private:
```

```
      float balamt;
   public:
      void getbal()
      { getdata();
        cin >> balamt;    }
      void dispbal()
      { displaydata();
        cout << balamt;     }
};
void main()
{ withdraw w;
  balance b;
  w.get();
  w.display();
  b.getbal();
  b.dispbal();
}
```

Output

For w

100	Savings Account	1200	(Input)
100	Savings Account	1200	(output)

For b

300	Savings Account	25000	(Input)
300	Savings Account	25000	(output)

In the above program the accno and accname are the common properties which are shared by each new class, i.e., withdraw and balance.

6. Hybrid Inheritance

To design a program sometimes we need to apply two or more types of inheritance. *For example*: Consider the case of processing the student results assume that we have to give weightage for extracurricular activities like sports, drawings etc. before finalizing the results. The weightage for extracurricular activities is stored in separate class known as extra. The new inheritance relationship between the various classes would be as shown in the following figure.

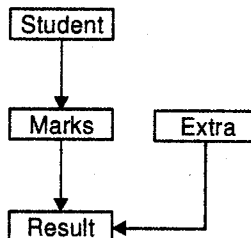


Figure 8.5 : Hybrid Inheritance (Multilevel, Multiple Inheritance)

Program for the implementation of both multilevel and multiple inheritance.

```
#include<iostream>
using namespace std;
class stduent{
  protected:
     int rollno;
  public:
     void getno(int);
     void putno(void); };
void student :: getno(int a)
{ rollno = a; }
void student :: putno()
{ cout << "Roll Number: " << rollno << "\n";    }
class marks : public student
{ protected:
     float m1, m2,m3;
  public:
     void getmarks(float, float, float);
     void putmarks(void); };
void marks :: getmarks(float x, float y, float z)
{ m1 = x;
  m2 = y;
  m3 = z;}
void marks :: putmarks()
{ cout << "Marks in m1:= " << m1 <<"\n";
  cout << "Marks in m2:= " << m2 <<"\n";
  cout << "Marks in m3:= " << m3 <<"\n";         }
class extra
{ protected:
     float m4;
  public:
     void getextra(float s)
     { m4 = s;}
     void putextra(void)
     { cout << "Extra Marks:=" << m4 << "\n\n";} };
class result : public marks, public extra
{ private:
     float total;
     float percentage;
  public:
     void display(void);  };
void result :: display(void)
{ total = m1 + m2 + m3 + m4;
  percentage = total/4;
  putno();
  putmarks();
  putextra();
  cout << "Total Marks:=" << total << "\n";
  cout << "Percentage:=" << percentage << "\n";}
int main()
{
```

```
  result student1;   // object student1 is created
  student1.getno(30);
  student1.getmarks(40.5, 60.0, 80.0);
  student1.getextra(50.5);
  student1.display();
  return 0;
}
```

Output

Roll Number := 30
Marks in m1 := 40.5
Marks in m2 := 60.0
Marks in m3 := 80.0
Extra Marks := 50.5
Total Marks := 231
Percentage := 57.75

7. Container Classes

C++ allows to declare an object of a class as a member of another class. When an object of a class is declared as a member of another class, it is called as a container class. Examples of container classes are arrays, linked lists, stacks and queues.

The general syntax for the declaration of container class is,

```
class user_defined_name1{
     . . .
     . . .
};
class user_defined_name2{
     . . .
     . . .
};
class user_defined_namen{
     . . .
     . . .
};
class derived_class
{
  user_defined_name1 obj1;     // object of the class one
  user_defined_name2 obj2;     // object of the class two
  . . .
  user_defined_namen objn;     // object of the class n
};
```

Program to demonstrate how a container class is declared and defined

```
#include<iostream>
#include<iomanip>
```

```
using namespace std;
class basic_info
{ private:
     char name[30];
     long int rollno;
     char sex;
  public:
     void getdata();
     void display();
}; // end of class definition
class academic_fit
{ private:
     char course[30];
     char semester[10];
     int rank;
  public:
     void getdata();
     void display();
}; // end of class definition
class financial_assit
{ private:
     float amount;
     basic_info bdata;     //object of class basic_info
     academic_fit acd;     //object of class academic_fit
  public:
     void getdata();
     void display();
}; // end of class definition
void basic_info :: getdata()
{ cout << "Enter name: \n";
  cin  >> name;
  cout << "Enter roll no:\n";
  cin  >>  rollno;
  cout << "Enter sex:\n";
  cin  >> sex;
}
void basic_info :: display()
{ cout << name     << " ";
  cout << rollno   << " ";
  cout << sex      << " ";
}
void academic_fit :: getdata()
{ cout << "Course Name (MBA, MCM,MCS, MCA etc)\n";
  cin  >> course;
  cout << "Semester (First/Second etc)\n";
  cin  >> semester;
  cout << "Rank of the student \n";
  cin  >> rank;
}
void academic_fit :: display()
{ cout << course     << " ";
  cout << semester   << " ";
  cout << rank       << " ";
}
```

```
void financial_assit :: getdata()
{  bdata.getdata();
   acd.getdata();
   cout << "Amount in rupees\n";
   cin  >> amount;
}
void finanacial_assit :: display()
{  bdata.display();
   acd.display();
   cout << setprecision(2);
   cout << amount << " ";
}
void main()
{  financial_assit f;
   cout << "Enter the following information for financial assistance";
   f.getdata();
   cout << endl;
   cout << "Academic Performance for Financial Assistance\n";
   cout << "___________________________________________________________\n";
   cout << "Name   Rollno   Sex   Course   Semester   Rank   Amount\n";
   cout << "___________________________________________________________\n";
   f.display();
   cout << endl;
   cout << "___________________________________________________________\n";
 }
```

8. Virtual Base Classes

PU Oct.09, Oct.10 – 5M
★ What is virtual base class? Explain with suitable example.

We know that multiple inheritance is a process of creating a new class which is derived from more than one base classes. Multiple inheritance hierarchies can be complex, which may lead to a situation in which a derived class inherits multiple times from the same indirect base class. In *figure 8.6*, 'abc' has two direct base classes 'derivedB' and 'derivedC' which themselves have a common base class 'baseA'. The 'abc' inherits the properties of 'baseA' via two separate paths. It can also inherit directly as shown by the broken line. The class 'baseA' is referred as indirect base class.

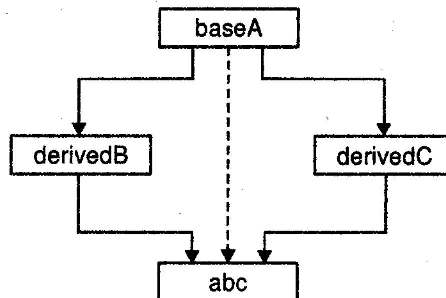


Figure 8.6

From the above situation we can write a program segment as follows:

```
class baseA {
  protected:
    int x;
    . . . };
class derivedB : public baseA {     //path1, through derivedB
  protected:
    . . . };
class derivedC : public baseA{    //path2, through derivedC
  protected:
    . . . };
class abc : public derivedB, public derivedC
{ // the data member x comes twice
   . . . };
```

In the above segment, the data member x is inherited twice in the derived class abc, once through the derived class derivedB and again through derivedC. This is wasteful and confusing. To avoid such multiple repetition of the data member we have to convert the derived class derivedB and derivedC into virtual base classes. Any base class which is declared using the keyword *virtual* is called as virtual base class. Virtual base class is a useful method to avoid unnecessary repetition of the same data member in the multiple inheritance hierarchies.

The following program segment shows how a base class is derived only once from the derived classes via virtual base classes.

```
class baseA {
  protected:
    int x;
    . . . };
class derivedB : public virtual baseA {    //path1, through derivedB
  protected:
    . . . };
class derivedC : virtual public baseA{     //path2, through derivedC
  protected:
    . . . };
class abc : public derivedB, public derivedC
{ // the data member x comes only once
   . . .    };
```

Note: The keywords *virtual* and *public* may be used in any order.

When we make derivedB and derivedC as virtual base classes for abc then only one copy of the data member x is available.

Consider again the student results processing system. Assume that the class *extra* derives the rollno from the class student. Then the inheritance relationship will be as shown in *figure 8.7*.

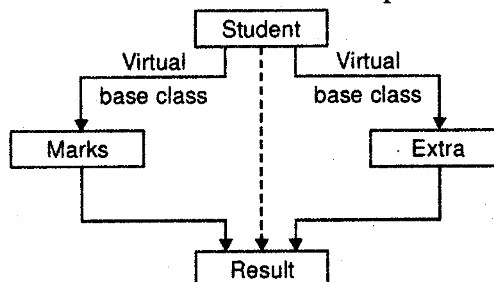


Figure 8.7: Virtual base class

Program to implement the concept of virtual base class

```
#include<iostream>
using namespace std;
class student{
  protected:
    int rollno;
  public:
    void getno(int);
    void putno(void);    };
void student :: getno(int a)
{ rollno = a; }
void student :: putno()
{ cout << "Roll Number: " << rollno << "\n";      }
class marks : virtual public student
{ protected:
    float m1, m2,m3;
  public:
    void getmarks(float, float, float);
    void putmarks(void);                        };
void marks :: getmarks(float x, float y, float z)
{ m1 = x;
  m2 = y;
  m3 = z;}
void marks :: putmarks()
{ cout << "Marks in m1:= " << m1 <<"\n";
  cout << "Marks in m2:= " << m2 <<"\n";
  cout << "Marks in m3:= " << m3 <<"\n";          }
class extra : public virtual student
{ protected:
    float m4;
  public:
    void getextra(float s)
    { m4 = s;}
    void putextra(void)
    { cout << "Extra Marks:=" << m4 << "\n\n";}  ;
class result : public marks, public extra
{ private:
    float total;
    float percentage;
  public:
    void display(void) };
void result :: display(void)
{ total = m1 + m2 + m3 + m4;
  percentage = total/4;
  putno();
  putmarks();
  putextra();
  cout << "Total Marks:=" << total << "\n";
  cout << "Percentage:=" << percentage << "\n";   }
int main()
{ result student1;   // object student1 is created
  student1.getno(30);
```

```
  student1.getmarks(40.5, 60.0, 80.0);
  student1.getextra(50.5);
  student1.display();
  return 0;
}
```

Output

Roll Number := 30
Marks in m1 := 40.5
Marks in m2 := 60.0
Marks in m3 := 80.0
Extra Marks := 50.5
Total Marks := 231
Percentage := 57.75

In the following example, `derived1a` and `derived1b` each contain a `base` object, but `derived2` contains just one `i` field.

```
#include<iostream>
using namespace std;
class base    {
  public:
    int i; };
class derived1a: virtual public base {
  public:
    int j; };
class derived1b: virtual public base {
  public:
    int k; };
class derived2: public derived1a, public derived1b{
  public:
    int product() {return i*j*k;}
};
int main()
{ derived2 obj;
  obj.i = 10;
  obj.j = 3;
  obj.k = 5;
  cout << "product is" << obj.product() << '\n';
  return 0;
}
```

The class which is not used to create objects, but is designed only to act as a base class, so as to be inherited by other classes, is called an abstract class. It is a design concept in program development and provides a base upon which other classes may be built. In the previous examples, class student and base are abstract classes.

9. Constructors in Derived Classes

We know that the constructor function is used to initialize the objects. Whenever an object of a class is created, a constructor member function is invoked automatically. As long as no base class constructor takes any arguments, the derived class need not have a constructor function. If any base class contains a constructor with one or more arguments then it is compulsory for the derived class to have a constructor and pass the arguments to the base class constructors.

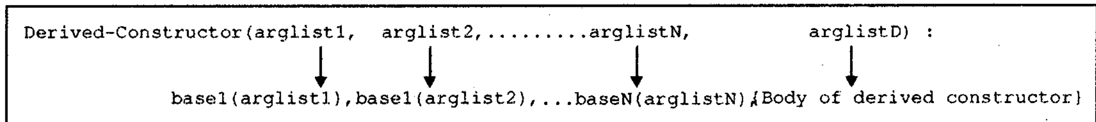
While applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. Similarly, in multilevel inheritance, the constructors will be executed in the order of inheritance.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. For passing the values to the base class constructors C++ supports a special argument passing mechanism for such situations.

The derived class constructor receives the entire list of values as its arguments and passes them on to the base class constructors in the order in which they are declared in the derived class. The base class constructors are called and executed before executing the statements in the body of the derived constructor.

The general form of defining a derived constructor is



where, **Derived-Constructor** function contain two parts separated by a colon. The first part contains the declaration of the arguments that are passed to the derived-constructor and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2), ...baseN(arglistN) are function calls to the base constructors base1(), base2(), . . . baseN() and therefore **arglist1, arglist2, ... arglistN** represent the actual parameters that are passed to the base constructors. arglist 1 to arglistN are the argument declarations for base constructors base1 to baseN. arglistD provides the parameters that are necessary to initialize the members of the derived class.

Example

```
alpha(int a, int b, int c, float d, int d1):
beta(a, b),    // call to the constructor beta
gama(c, d)     // call to the constructor gama
{
} alpha = d1;    // executes its own body
```

beta(a, b) invokes the beta() base constructor while gama(c, d) invokes gama() base constructor. The constructor alpha() supplies the values for these four arguments. In addition, it has one arguments of its own. The constructor alpha() has a total of five arguments. Alpha() may be invoked as follows:

```
alpha obj(4,  7,  8, 9, 23.5);
```

These values are assigned to various parameters by the constructor alpha() as follows:

4	7	8	9	23.5
↓	↓	↓	↓	↓
a	b	c	d	d1

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class construction is executed. Program for illustrating how constructors are implemented when the classes are inherited.

```
#include<iostream>
using namespace std;
class baseA
{ int x;
  public:
    baseA(int i)
    { x = i;
      cout << "baseA initialized \n";
    }
    void showx(void)
    { cout   << "x = " << x << "\n";}
};
class derivedD
{ float y;
  public:
    derivedD(float j)
    { y =j;
      cout <<"derivedD initialized \n";
    }
    void showy(void)
    { cout   << "y = " << y << "\n";}
};
class derivedE : public derivedD, public baseA
{ int m, n;
  public:
  derivedE(int a, float b, int c, int d): baseA(a), derivedD(b)
  { m = c;
    n = d;
    cout << "derivedE initialized \n";
  }
  void showmn(void)
  { cout <<"m = " << m <<"\n";
```

```
      cout <<"n = " << n <<"\n";
   }
};
int main()
{  derivedE e(5, 10.89, 40, 50);
   cout << "\n";
   e.showx();
   e.showy();
   e.showmn();
   return 0;
}
```

Output

baseA initialized

derivedD initialized

derivedE initialized

x = 5

y = 10.89

m = 40

n = 50

In the above program, derivedD is initialized first although it appears second in the derived constructor since it is declared first in the derived class header line. baseA(A) and derivedD(b) are the function calls. Therefore, the parameters should not include types.

10. Destructors in Derived Classes

It has been seen that destructor is a special member function. It is invoked automatically to free the memory space which is allocated by the constructor functions. Whenever an object of the class is getting destroyed, the destructors are used to free the heap area so that the free memory space may be used subsequently. In the previous section, it has been seen that constructors in hierarchy fire from a base class to a derived class. Destructors in hierarchy fire from a derived class to a base class order, i.e., they fire in the reverse order of that of the constructors.

A program to illustrate how the destructor member function gets fired from the derived class objects to the base class objects through pointers.

```
#include<iostream>
using namespace std;
class baseA{
   public:
      ~baseA();          //destructor
};
class derivedB : public baseA {
```

```
  public:
    ~derivedB();     // destructor
};
baseA :: ~baseA()
{ cout << "base class destructor \n";
}
derivedB :: ~derivedB()
{ cout << "derivedB class destructor \n";
}
void main()
{ derivedB objb;
 }
```

Output

derivedb class destructor

base class destructor

A program to display the message of both constructors and destructors of a base class and a derived class.

```
#include<iostream>
using namespace std;
class baseA{
  public:
    baseA() { //baseA's constructor
      cout << "base class constructor\n"; }
    ~baseA(); { // baseA's destructor
      cout << "base class destructor\n"; }
};
class derivedD : public baseA {
  public:
    derivedD() { //  derivedD's constructor
      cout << "derived class constructor\n"; }
    ~derivedD (); { // derivedD's destructor
      cout << "derived class destructor\n"; }
};
void main()
{ derivedD obj;
 }
```

Output

base class constructor

derived class constructor

derived class destructor

base class destructor

11. Nesting of Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called *nested classes*. Nested classes are used in situations where the nested class has a close conceptual relationship to its surrounding class. *For example*: With the class *string* a type *string::iterator* is available which will provide all characters that are stored in the *string*. This *string::iterator* type could be defined as an object *iterator*, defined as nested class in the class *string*.

A class can be nested in every part of the surrounding class: in the *public, protected* or *private* section. Such a nested class can be considered a member of the surrounding class. The normal access and rules in classes apply to nested classes. If a class is nested in the *public* section of a class, it is visible outside the surrounding class. If it is nested in the *protected* section it is visible in subclasses, derived from the surrounding class, if it is nested in the *private* section, it is only visible for the members of the surrounding class. The surrounding class has no special privileges with respect to the nested class. So, the nested class still has full control over the accessibility of its members by the surrounding class. *For example*: C`onsider the following class definition:

```
class Surround
{   public:
    class FirstWithin
    {  int d_variable;
       public:
       FirstWithin();
       int var() const
       { return d_variable;
       }
    };
    private:
    class SecondWithin
    {  int d_variable;
       public:
       SecondWithin();
       int var() const
       {  return d_variable;
       }
    };
};
```

In this definition access to the members is defined as follows:

The class *FirstWithin* is visible both outside and inside Surround. The class *FirstWithin* therefore has global scope.

The constructor *FirstWithin*() and the member function var() of the class *FirstWithin* are also globally visible.

The int d_variable datamember is only visible to the members of the class *FirstWithin*. Neither the members of Surround nor the members of *SecondWithin* can access d_variable of the class *FirstWithin* directly.

The class *SecondWithin* is only visible inside Surround. The public members of the class *SecondWithin* can also be used by the members of the class *FirstWithin*, as nested classes can be considered members of their surrounding class.

The constructor *SecondWithin*() and the member function var() of the class *SecondWithin* can also only be reached by the members of Surround (and by the members of its nested classes).

The int d_variable datamember of the class *SecondWithin* is only visible to the members of the class *SecondWithin*. Neither the members of Surround nor the members of *FirstWithin* can access d_variable of the class *SecondWithin* directly.

As always, an object of the class type is required before its members can be called. This also holds true for nested classes.

If the surrounding class should have access rights to the private members of its nested classes or if nested classes should have access rights to the private members of the surrounding class, the classes can be defined as friend classes.

The nested classes can be considered members of the surrounding class, but the members of nested classes are *not* members of the surrounding class. So, a member of the class Surround may not access *FirstWithin*::var() directly. This is understandable considering the fact that a Surround object is not also a *FirstWithin* or *SecondWithin* object. In fact, nested classes are just typenames. It is not implied that objects of such classes automatically exist in the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

For example: In the following class definition there is a surrounding class Outer and a nested class Inner. The class Outer contains a member function caller() which uses the inner object that is composed in Outer to call the infunction() member function of Inner:

```
class Outer
{  public:
   void caller()
   { d_inner.infunction();
   }
   private:
   class Inner
   {  public:
      void infunction();
   };
     Inner d_inner;      // class Inner must be known
};
```

The mentioned function Inner::infunction() can be called as part of the inline definition of Outer::caller(), even though the definition of the class Inner is yet to be seen by the compiler. On the other hand, the compiler must have seen the definition of the class Inner before a data member of that class can be defined.

11.1 Defining Nested Class Members

Member functions of nested classes may be defined as inline functions. Inline member functions can be defined as if they were functions defined outside of the class definition: if the function Outer::caller() would have been defined outside of the class Outer, the full class definition (including the definition of the class Inner) would have been available to the compiler. In that situation the function is perfectly compilable. Inline functions can be compiled accordingly: they can be defined and they can use any nested class, even if it appears later in the class interface.

Member functions of nested classes can also be defined outside of their surrounding class. Consider the constructor of the class *FirstWithin* in the example of the previous section. The constructor *FirstWithin*() is defined in the class *FirstWithin*, which is, in turn, defined within the class Surround. Consequently, the class scopes of the two classes must be used to define the constructor. e.g.

```
Surround::FirstWithin::FirstWithin()
{  variable = 0;
}
```

Static (data) members can be defined accordingly. If the class *FirstWithin* would have a static unsigned datamember epoch, it could be initialized as follows:

```
unsigned Surround::FirstWithin::epoch = 1970;
```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```
void showEpoch()
{ cout << Surround::FirstWithin::epoch = 1970;
}
```

Inside the members of the class Surround only the *FirstWithin*:: scope must be used; inside the members of the class *FirstWithin* there is no need to refer explicitly to the scope.

What about the members of the class *SecondWithin*? The classes *FirstWithin* and *SecondWithin* are both nested within Surround, and can be considered members of the surrounding class. Since members of a class may directly refer to each other, members of the class *SecondWithin* can refer to (public) members of the class *FirstWithin*. Consequently, members of the class *SecondWithin* could refer to the epoch member of *FirstWithin* as

```
FirstWithin::epoch
```

11.2 Declaring Nested Classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers, references, parameters or return values to objects of the other nested classes. *For example*: The following class Outer contains two nested classes Inner1 and Inner2. The class Inner1 contains a pointer to Inner2 objects, and Inner2 contains a pointer to Inner1 objects. Such cross references require forward declarations. These forward declarations must be specified in the same access-category as their actual definitions. In the following example the Inner2 forward declaration must be given in a private section, as its definition is also part of the class Outer's private interface:

```
class Outer
{ private:
  class Inner2;          // forward declaration
  class Inner1
  {  Inner2 *pi2;        // points to Inner2 objects
  };
     class Inner2
  { Inner1 *pi1;        // points to Inner1 objects
  };
};
```

11.3 Accessing Private Members in Nested Classes

To allow nested classes to access the private members of their surrounding class; to access the private members of other nested classes; or to allow the surrounding class to access the private members of its nested classes, the friend keyword must be used.

Consider the following situation, in which a class Surround has two nested classes *FirstWithin* and *SecondWithin*, while each class has a static data member int s_variable:

```
class Surround
{   static int s_variable;
    public:
    class FirstWithin
     {   static int s_variable;
         public:
         int value();
     };
         int value();
       private:
       class SecondWithin
     { static int s_variable;
       public:
       int value();
     };
};
```

If the class Surround should be able to access *FirstWithin* and *SecondWithin*'s private members, these latter two classes must declare Surround to be their friend.

The function Surround::value() can thereupon access the private members of its nested classes. *For example*: Note the friend declarations in the two nested classes:

```
class Surround
{   static int s_variable;
     public:
     class FirstWithin
     {   friend class Surround;
         static int s_variable;
         public:
         int value();
     };
     int value()
     {   FirstWithin::s_variable = SecondWithin::s_variable;
         return (s_variable);
     }
          private:
           class SecondWithin
     {   friend class Surround;
         static int s_variable;
          public:
          int value();
     };
};
```

Now, to allow the nested classes access to the private members of their surrounding class, the class Surround must declare its nested classes as friends. The friend keyword may only be used when the class that is to become a friend is already known as a class by the compiler, so either a forward declaration of the nested classes is required, which is followed by the friend declaration, or the friend declaration follows the definition of the nested classes. The forward declaration followed by the friend declaration looks like this:

```
class Surround
{   class FirstWithin;
    class SecondWithin;
    friend class FirstWithin;
    friend class SecondWithin;
    public:
    class FirstWithin;
      ...
```

Alternatively, the friend declaration may follow the definition of the classes. Note that a class can be declared a friend following its definition, while the inline code in the definition already uses the fact that it will be declared a friend of the outer class. Also note that the inline code of the nested class uses members of the surrounding class not yet seen by the compiler. Finally note that 's_variable' which is defined in the class Surround is accessed in the nested classes as Surround::s_variable:

```
class Surround
{   static int s_variable;
    public:
    class FirstWithin
    {   friend class Surround;
        static int s_variable;
        public:
        int value()
        {   Surround::s_variable = 4;
            Surround::classMember();
            return s_variable;
        }
    };
    friend class FirstWithin;
    int value()
    {   FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
    }
    private:
    class SecondWithin
    {   friend class Surround;
        static int s_variable;
        public:
        int value()
        {   Surround::s_variable = 40;
            return s_variable;
        }
    };
    static void classMember();
    friend class SecondWithin;
};
```

Finally, we want to allow the nested classes access to each other's private members. Again this requires some friend declarations. In order to allow *FirstWithin* to access *SecondWithin*'s private members nothing but a friend declaration in *SecondWithin* is required. However, to allow *SecondWithin* to access the private members of *FirstWithin* the friend class *SecondWithin* declaration cannot plainly be given in the class *FirstWithin,* as the definition of *SecondWithin* is as yet unknown. A forward declaration of *SecondWithin* is required, and this forward declaration must be provided by the class Surround, rather than by the class *FirstWithin.*

Clearly, the forward declaration class *SecondWithin* in the class *FirstWithin* itself makes no sense, as this would refer to an external (global) class *SecondWithin.* Likewise, it is impossible to provide the forward declaration of the nested class *SecondWithin* inside *FirstWithin* as class Surround::*SecondWithin,* with the compiler issuing a message like `Surround' does not have a nested type named 'SecondWithin'.

The proper procedure here is to declare the class SecondWithin in the class Surround, before the class *FirstWithin* is defined. Using this procedure, the friend declaration of SecondWithin is accepted inside the definition of FirstWithin. The following class definition allows full access of the private members of all classes by all other classes:

```
class Surround
{  class SecondWithin;
   static int s_variable;
   public:
   class FirstWithin
   {   friend class Surround;
       friend class SecondWithin;
       static int s_variable;
       public:
       int value()
       {  Surround::s_variable= SecondWithin::s_variable;
          return s_variable;
       }
   };
   friend class FirstWithin;
   int value()
   {   FirstWithin::s_variable = SecondWithin::s_variable;
       return s_variable;
   }
    private:
    class SecondWithin
    {   friend class Surround;
        friend class FirstWithin;
        static int s_variable;
        public:
        int value()
        {  Surround::s_variable = FirstWithin::s_variable;
           return s_variable;
        }
    };
  friend class SecondWithin;
};
```

12. Pointers to Derived Classes

C++ allows a pointer in a base class to point to either a base class object or to any derived class object. The following program segment illustrates how a pointer is assigned to point to the object of the derived class.

```
class baseA
{
   . . .
   . . . };
class derived : public baseA

{
   . . .
   . . . };
void main()
{  baseA *ptr;       //pointer to baseA
   derived objd;
   ptr =&objd;       // indirect reference objd to the pointer
   . . .
   . . .
}
```

The pointer *ptr* points to an object of the derived class *objd.*

But there is a problem in using **ptr** to access the public members of the derived class **derived.** Using ptr, we can access only those members which are inherited from baseA and not the members that originally belong to derived. In case a member of derived has the same name as one of the members of baseA, then any reference to that members by ptr will always access the base class member.

In contrast, a pointer to a derived class object may not point to a base class object with explicitly casting. *For example*: The following assignment statements are invalid:

```
void main()
{  baseA obj
   derived *ptr;
   ptr = &obj;    //invalid
}
```

Note that a derived class pointer cannot point to base class object. But, the above code can be corrected by using the explicit casting.

```
void main()
{  square sqobj;
   rectangle *ptr;    //pointer of the derived class
   ptr = (rectangle*) &sqobj;   //explicit casting
   ptr->display();
   . . . }
```

A program to illustrate how to assign the pointer of the derived class to the object of a base class using explicit casting.

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

```
{ public:
    int b;
  void show()
  { cout << "b =" << b <<"\n";}
};
class derivedD : public baseA
{ public:
    int d;
    void show()
    { cout << "b = " << b <<"\n" << "d=" << d <<"\n";   }
};
int main()
{ baseA *bptr;                  // base pointer
  baseA base;
  bptr = &base;                 // base address
  bptr->b = 100;                // access baseA via base pointer
  cout << "bptr points to base object \n";
  bptr->show();
  // derived class
  derivedD derived;
  bptr=&derived;              // address of derived object
  bptr->b = 200;              // access derivedD via base pointer
  cout << "bptr now points to derived object \n";
  bptr->show();             // bptr now points to derived object
  // accessing d using a pointer of type derived class derivedD
  derivedD *dptr;           // derived type pointer
  dptr = &derived;
  dptr->d = 300;
  cout << "dptr is derived type pointer \n";
  dptr->show();
  cout << "using ((derivedD *)bptr)\n";
  ((derivedD *)bptr)->d=400;
  ((derivedD *)bptr)->show();
  return 0;
 }
```

Output

bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((derivedD *) bptr)
b = 200
d = 400

The statement bptr-> show(); is used two times. First, when bptr points to the base object and second when bptr is made to point to the derived object. But both the times, it executed baseA::show() function and displayed the content of the base object. However the statements dptr->show();

```
((derivedD *) bptr)->show();    //cast bptr to derivedD type
```

display the contents of the derived object. This shows that although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.

13. Virtual Functions

A virtual function is one that does not really exist but it appears real in some parts of a program. Virtual function is an advanced feature of the Object-Oriented Programming. A virtual function is a function that is declared within a base class and redefined by a derived class. When we use the same function name in both the base and derived classes, the function in the base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration. Virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. Hence, the redefinition creates a specific method. 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. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. And this determination is made at run time. The general syntax of the virtual function declaration is:

```
class user_defined_name{
  private:
      . . .
      . . .
  public:
    virtual return_type function-name1(argument);
    virtual return_type function-name2(argument);
    virtual return_type function-name3(argument);
    . . .
    . . . };
```

To make a member function virtual, the keyword **virtual** is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual should be followed by a return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

```
#include<iostream>
using namespace std;
class Base
{
  public:
  void display()
  { cout << " \n Display Base ";
  }
  virtual void show()
```

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

    {
      cout << " \n Display Derived ";
    }
  void show()
  { cout << " \n Show Derived ";
  }
};
int main()
{ Base B;
  Derived D;
  Base *bptr;
  cout << " bptr points to Base \n ";
  bptr = &B;
  bptr -> display(); // calls Base version
  bptr -> show();      // calls Base version
  cout << " \n\n  bptr points to Derived\n ";
  bptr = &D;
  bptr -> display(); // calls Base version
  bptr -> show();      // calls Base version
  return 0;
 }
```

Output

bptr points to Base
Display Base
Show Base
bptr points to Derived
Display Base
Show Derived

Note: When *bptr* is made to point to the object *D*, the statement *bptr –> display();* calls only the function associated with the Base (i.e., *Base :: Display()*), whereas the statement *bptr –> show();* calls the Derived version of *show()*. This is because the function *display()* has not been made virtual in the Base class.

The most important point to note that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. We can use the object name with the dot operator the same way as any other member function to call the virtual functions but run time polymorphism is achieved only when a **virtual** function is accessed through a pointer to the base class.

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. Only a member function of a class can be declared as virtual. It is an error to declare a non member function of a class as virtual.

2. The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.
3. A virtual function cannot be static member because a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.
4. They are accessed by using object pointers.
5. A virtual function can be a friend of another class.
6. A virtual function in a base class must be defined, even though it may not be used.
7. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
8. We have only virtual destructors and cannot have virtual constructors.
9. A destructor member function does not take any argument and no return type can be specified for it not even void.
10. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
11. When a base pointer points to derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
12. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

14. Pure Virtual Functions

> 1 PU Apr. 2010 – 5M
> * What is pure virtual function? Explain with suitable example.

The functions which are only declared but not defined in the base class are called *pure virtual functions*. A function is made pure virtual by preceding its declaration with the keyword `virtual` and by postfixing it with `= 0`.

The general form of pure virtual function declaration is

```
virtual return_type function_name(argument list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile time error will result.

Consider the following example of a pure virtual function where the definition of a class `Sortable` requires that all subsequent classes have a function `compare()`:

```
class Sortable
{  public:
     virtual int compare(Sortable const &other) const = 0;
};
```

The function `compare()` must return an `int` and receives a reference to a second `Sortable` object. Possibly its action would be to compare the current object with the `other` one. The function is not allowed to alter the other object, as `other` is declared `const`. Furthermore, the function is

not allowed to alter the current object, as the function itself is declared `const`. A class containing one or more pure virtual functions cannot be used to define an object. The class is therefore only useful as a base class to be inherited into a useable derived class. It is called an abstract class. A program for illustrating how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class.

Program for Pure Virtual Functions

```
#include<iostream>
using namespace std;
class base
{ public:
   virtual void getdata() = 0;
   virtual void display() = 0; };
class derivedB : public base{
private:
   long int code;
   char name[20];
public:
   void getdata();
   void display(); };
void base :: getdata()     { }
void base :: display()     { }
void derivedB :: getdata()
{  cout << "Enter code of the employee:=";
   cin >> code;
   cout << "Enter name of the employee:=";
   cin >> name; }
void derivedB :: display()
{  cout << "---------------------------------------";
   cout << "Employee code  Employee name\n";
   cout << "---------------------------------------";
   cout<<  code   << "\t"<< name << endl; }
void main()
{  base *ptr;
   derivedB obj;
   ptr = &obj;
   ptr->getdata();
   ptr->display();
 }
```

Output

```
Enter code of the employee:= 001
Enter name of the employee:= sagar
---------------------------------------------
Employee code     Employee name
---------------------------------------------
   001                 sagar
```

15. Abstract Classes

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannót create an object of an abstract class type; however, you can use pointers and references to abstract class types. A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism. Program to illustrate how to define an abstract base class with pure virtual functions in which the function definition part has been defined without any statement. The members of the derived class objects are accessed through the base class objects through pointer technique.

Program for Abstract Classes

```
#include<iostream>
using namespace std;
class base{
public:
  virtual void getdata() = 0;
  virtual void display() = 0; };
class derivedB : public base{private:
  long int code;
  char name[20];
public:
  void getdata();
  void display(); };
class derivedC : public base{
private:
  float height;
  float weight;
public:
  void getdata();
  void display(); };
void base :: getdata()    //pure virtual function
{ }
void base :: display()    //pure virtual function
{ }
void derivedB :: getdata()
{ cout << "Enter code of the employee:=";
  cin >> code;
  cout << "Enter name of the employee:=";
  cin >> name;  }
void derivedB :: display()
{ cout << "Employee code  Employee name\n";
  cout <<  code << "\t"<< name << endl; }
void derivedC :: getdata()
{ cout << "Enter height of the employee:=";
  cin >> height;
  cout << "Enter weight of the employee:=";
  cin >> weight; }
void derivedC :: display()
```

```
{ cout << "Height and weight of Employee:= \n";
  cout << height << "\t"<< weight << endl; }
void main() { base *ptr[3];
  derivedB objb;
  derivedC objc;
  ptr[0] = &objb;
  ptr[1] = &objc;
  ptr[0]->getdata();
  ptr[1]->getdata();
  ptr[0]->display();
  ptr[1]->display();
 }
```

Output

Enter code of the employee:= 001
Enter name of the employee:= sagar
Enter height of the employee:=135.0
Enter weight of the employee:=70
Employee code Employee name
001 sagar
Height and weight of Employee:=
135.0 70

Restrictions for using Abstract Classes

Abstract classes cannot be used for:

- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions

Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.

Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the **syntax**

```
abstract class_name :: function_name()
```

This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object. Consider the following example:

```
// Declare an abstract base class with a pure virtual destructor.
class base
{ public:
    base() {}
    virtual ~base()=0; };
```

```
// Provide a definition for destructor.
base::~base()
{ }
class derived:public base
{ public:
     derived() {}
     ~derived(){} };
int main()
{  derived *pDerived = new derived;
   delete pDerived; }
```

When the object pointed to by `pDerived` is deleted, the destructor for class `derived` is called and then the destructor for class `base` is called. The empty implementation for the pure virtual function ensures that at least some implementation exists for the function.

▶ Early binding Vs. Late Binding

Early binding occurs when all information needed to call a function is known at compile time. The early binding is very efficient as all the necessary information to call a function is determined at compile time, making the function calls very fast. Examples are: standard library function calls, overloaded function calls, overloaded operators. Late binding refers to function calls that are not resolved until run time. Its main advantage is flexibility. It allows to create programs that can respond to events occurring while the program executes without having to create a large amount of code. Due to the same reason, it is the cause of slower execution times. Example of late binding is virtual functions.

Solved Programs

1. What will be the output of the following program?

PU
Oct. 2010 – *2M*

```
#include<iostream.h>
#include<conio.h>
public class A
{
        int x=4;
};
private class B::class A
{
        int x=20;
        cout<<"x="<<x;
};
```

Solution

The above code would result into following error:

Declaration terminated incorrectly.

This error would occur twice.

EXERCISES

A. Review Questions

1. What does inheritance mean in C++? Explain its advantages.
2. What are the different forms of inheritance? Given an example for each.
3. How is direct base class different from the indirect base class declaration in C++?
4. Define Multiple Inheritance.
5. List the merits and demerits of single inheritance over multiple inheritances.
6. Explain the merits and demerits of private derivation over the public derivation.
7. What is a container class?
8. What are the syntactic rules to be followed to avoid the ambiguity in single and multiple inheritance?
9. What is a virtual function and what are the advantages of declaring a virtual function in a program?
10. What is virtual base class and an abstract base class?
11. What is a pure virtual function? What are the merits and demerits of defining and declaring a pure virtual function in a program?
12. What are the syntactic rules to be observed while defining the keyword virtual?

B. Programming Exercises

1. Develop a program to prepare the marksheet of an university examination with the following items read from the keyboard.
   Name of the student, rollno, subject name, subject code, internal marks, external marks. Design a base class consisting of data members such as name of the student, rollno and subject name. The derived class consists of the data members, viz., subject code, internal marks and external marks.

Collection of Questions asked in Previous Exams PU

1. What is virtual base class? Explain with suitable example. [Oct. 2009, Oct. 2010 – 5M]
2. What is pure virtual function? Explain with suitable example. [Apr. 2010 – 5M]
3. Explain different types of inheritance with suitable examples of each type. [Apr. 2010, Oct. 2010 – 10M]
4. What will be the output of the following program? [Oct. 2010 – 2M]

```
#include<iostream.h>
#include<conio.h>
public class A
{
        int x=4;
};
private class B::class A
{
        int x=20;
        cout<<"x="<<x;
};
```

5. Why virtual constructors are not possible, but virtual destructor is a good idea? [Oct. 2011 – 5M]

9 The C++ I/O System Basics

1. Introduction

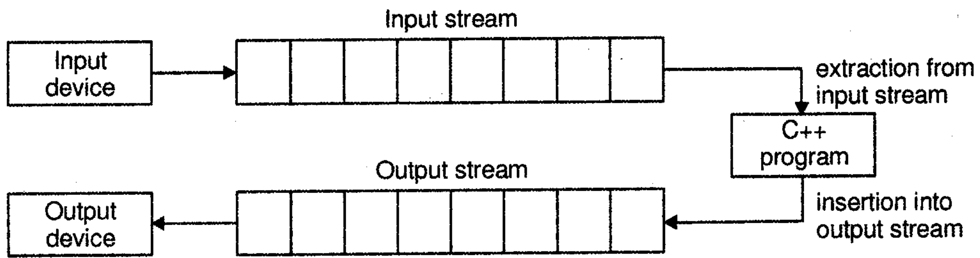
Every program takes some data as input, processes it and displays the output. Therefore it is essential to know how to provide the input data and how to display the result in the desired form. C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs. But we restrain from using them due to two reasons. First, I/O methods in C++ support the concepts of OOP and secondly, I/O methods in C cannot handle the user defined data types such as class object.

C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files. This chapter explains, how stream classes support the console oriented input-output operations. File-oriented I/O operations will be discussed in the next chapter.

Note: The console is the basic interface of computers, normally it is the set composed of the keyboard and the screen. The keyboard is generally the standard input device and the screen the standard output device.

2. C++ Stream

A stream is a sequence of bytes. The sequence of bytes flowing into a program is called as input stream and the one flowing out from the program is called as the output stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream.



The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout which have been used very often in our earlier programs. In short, we can say that stream is a general name given to a flow of data. In C++, a stream is represented by an object of a particular class. Different streams are used to represent different kinds of data flow. The advantage of streams in C++ is that streams are the best to write data to files and also to format data in memory for later use in text I/O windows and other GUI (Graphical User Interface) elements.

3. C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called Stream Classes.

Stream Class Hierarchy

The hierarchy of the stream classes used for input and output operations with the console unit is shown in *figure 9.1.* These classes are declared in the header file iostream. This file should be included in all the programs that communicate with the console unit.

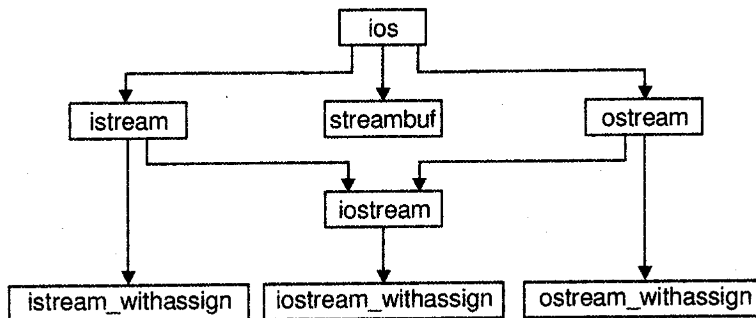


Figure 9.1: Stream class hierarchy

ios is the base class for istream (input stream) and ostream (output stream) which are in turn base classes for iostream (input/output stream).

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream (through inheritance) provides the facilities for formatted output. The class iostream provides the facilities for handling both input and output streams. Three classes, namely, istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.

i. **The streambuf Class:** The **streambuf** class provides memory for a buffer along with class methods for filling the buffer, accessing buffer contents, flushing the buffer, and managing the buffer memory. It handles the most primitive functions for streams on a first-in-first-out basis.

ii. The **filebuf** class is derived from class **streambuf** and extends it by providing basic file operations.

iii. The **strstreambuf** class is derived from class **streambuf** and is designed to handle memory buffers.

iv. **The ios Class (General input/output stream class):** This class contains basic facilities that are used by all other input and output classes, and also a pointer to a buffer object, i.e., streambuf object. It declares constants and functions that are necessary for handling formatted input and output operations.

v. **The istream Class (input stream class):** This class inherits the properties of *ios* class and provides input methods. That is, it accepts data from an input device in the way you expect. This class declares input functions such as get(), getline() and read() and contains overloaded extraction operator >>.

vi. **The ostream (output stream class) Class:** This class inherits the properties of *ios* and provides output methods. That is, it formats the data you send to an output device so that it appears in the way you expect. This class declares output functions such as put() and write() and contains overloaded insertion operator <<.

vii. **The iostream (input/output stream class) Class:** This class inherits the properties of istream and ostream through multiple inheritance and thus contains all the input and output functions.

The _withassign Classes: There are three _withassign classes. They are istream_withassign, ostream_withassign and iostream_withassign.

These classes are derived from istream, ostream and iostream respectively. The _withassign classes include the assignment operators. Using these operators the objects of the _withassign classes can be copied. istream, ostream and iostream classes are made uncopyable by making assignment operators private.

4. Unformatted I/O Operations

In the *iostream* C++ library, standard *input* and *output* operations for a program are supported by two data streams: **cin** for input and **cout** for output. Additionally, **cerr** and **clog** have also been implemented (these are two output streams specially designed to show error messages). The cout object representing the *standard output* stream, which is usually directed to the video display, is a predefined object of the ostream_withassign class, which is derived from the ostream class. Similarly, cin object representing the *standard input stream*, which is usually directed to the

keyboard, is a predefined object of the istream_withassign class which is derived from istream. By handling these two streams you will be able to interact with the user in your programs since you will be able to show messages on the screen and receive his/her input from the keyboard. As already known the general format for cin and cout are as follows:

```
cin>>variable 1 >>variable2>>--->>variableN;
cout<<item1<<item2<<---<<itemN;
```

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. When such a situation is encountered, the next data item remains in the input stream and will be input to the next cin statement or any other input statement (discussed later in the chapter).

4.1 Cerr and Clog

In addition to the function cout, C++ provides other functions of the class ostream called cerr and clog. The **cerr** object corresponds to the standard error stream, which can be used for displaying error messages. By default, this stream is associated with the standard output device, typically a monitor, and the stream is unbuffered. Unbuffered means that information is sent directly to the screen without waiting for a buffer to fill or for a newline character. The **clog** object also corresponds to the standard error stream which can be used for logging messages. By default, this stream is associated with the standard output device, typically a monitor, and the stream is buffered.

4.2 get() Function

The get() function is used with input streams.

There are three versions of get() function which are as follows:

i. **No arguments** returns the character being input. *For example:* `int get();`

ii. **One character-reference argument** inputs the next character from the input stream (even if this is a whitespace character) and stores it in the character argument.

```
istream &get(char&ch);
```

iii. **Three arguments**: A character array, a size limit and a delimiter (with default value '\n'). A null character is inserted to terminate the input string in the character array. The delimiter is not placed in the character array, but does remain in the input stream (the delimiter will be the next character read).

Thus, the result of a second consecutive **get** is an empty line, unless the delimiter character is removed from the input stream (possibly with **cin.ignore()**).

Program : Input of a String via cin vs. cin.get

```
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main() { // create two char arrays, each with 80 elements
   const int SIZE = 80;
   char buffer1[SIZE];
   char buffer2[SIZE];
   // use cin to input characters into buffer1
   cout << "Enter a sentence:" << endl;
   cin >> buffer1;
   // display buffer1 contents
   cout << "\nThe string read with cin was:"<< endl<< buffer1 <<  endl <<
endl;
   // use cin.get to input characters into buffer2
   cin.get(buffer2, SIZE);  //version 2 of get()
   // display buffer2 contents
   cout << "The string read with cin.get was:" << endl << buffer2 << endl;
   return 0;
 }  // end main
```

Output

Enter a Sentence:
The snow flakes fall as a winter calls, the time just seems to fly
The string read with cin was:
The
The string read with cin.get was:
snow flakes fall as a winter calls, the time just seems to fly
Press any key to continue

In the above program, cin statement reads characters till it encounters a white space, i.e., "The". remaining characters stay in the input stream and are assigned to buffer2 as soon as cin.get() is encountered, since get() can read white spaces.

4.3 put() Function

The function put() is used with output streams, and writes the character *ch* to the stream.

Syntax

```
ostream &put(char ch);
```

Examples

1.
```
cout.put(65);  // outputs "A"
```
2.
```
for(int i=0; i < 256; i++)
cout.put(i);
cout.put('\t');
```

Output

```
     ☺   ☻   ♥   ♦   ♣   ♠
     ♪   ☼   ►   ◄   ↕   ‼   ¶   §   ▬
↨    ↑   ↓   →   ←   ∟   ↔   ▲   ▼   •
!    "   #   $   %   &   '   (   )   4
+    ,   -   .   /   0   1   2   3   >
5    6   7   8   9   :   ;   <   =   H
?    @   A   B   C   D   E   F   G   R
I    J   K   L   M   N   O   P   Q   \
s    T   U   V   W   X   Y   Z   [   f
]    ^   _   `   a   b   c   d   e   p
g    h   i   j   k   l   m   n   o   z
q    r   s   t   u   v   w   x   y
<    ¦   >   ~   ∇   Ç
```

4.3 The getline() Function

Getline() is a line oriented input function. **getline()** operates similarly to the third version of the **get** member function. It reads a whole line of text that ends with a new line character. The **getline** function removes the *delimiter* from the stream (i.e., reads the character and discards it), but does not store it in the character array.

This function can be invoked as follows. `cin.getline(line, size);`

The above function call invokes the getline() function which reads character input into the variable 'line'. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read (whichever occurs first). The newline character is read but not saved; instead it is replaced by the null character.

```
#include<iostream>
using namespace std;
int main()
{  const int SIZE = 80;
   char buffer[SIZE];  // create array of 80 characters
   // input characters in buffer via cin function getline
   cout << "Enter a sentence:" << endl;
   cin.getline(buffer, SIZE);
   // display buffer contents
   cout << "\nThe sentence entered is:" << endl << buffer << endl;
   return 0;
}  // end main
```

Output

Enter a sentence:
Once there was a green field
The sentence entered is:
Once there was a green field
Press any key to continue

4.4 The write() Function

The write() function is a line oriented output function. It displays an entire line and has the following form:

```
cout.write(string, size);
```

The first argument 'string' is the name of the string to be displayed and the second argument 'size' is the number of characters to display. If size is greater than the length of string, write() does not stop displaying automatically (write() does not stop on encountering the null character), but it displays beyond the bounds of line.

Example:

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write(buffer, 10);
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

Program for unformatted I/O using read, gcount and write

```
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{  const int SIZE = 80;
   char buffer[SIZE];
   // use function read to input characters into buffer
   cout << "Enter a sentence:" << endl;
   cin.getline(buffer, 20);
   // use functions write and gcount to display buffer characters
   cout << endl << "The sentence entered was:" << endl;
   cout.write(buffer, SIZE);
   cout << endl;
   return 0;
 }  // end main
```

Output

Enter a sentence:
Today while the blossoms still cling to the vine
The sentence entered was:
Today while the blos
Press any key to continue

5. Formatted I/O Operations

There are a number of ways in which the output format of the fundamental types of C++ can be altered, and a few ways in which the requirements on the input format can be altered. *For example*: A field width can be set, also the alignment within that field. For integral types, the base can be set (decimal, octal, hexadecimal). For floating point types, the format can be fixed point or scientific. All of these, and yet some, are controlled with a few formatting flags, and a little data.

All flags are set or cleared with the member functions "os.setf()" and "os.unsetf()". The iostream objects maintain *format states* controlling the default formatting of values. The format states can be controlled by member functions and by manipulators. Manipulators are inserted into the stream, the member functions are used by themselves. The ios class contains a large number of member functions that helps us to format the output in a number of ways. The most important ones among them are listed in the following table:

Function name	Purpose
Width()	Read/set the field size.
Precision()	Read/set the number of digits to be displayed after the decimal point of a float value.
Fill()	Read/set a character that is used to fill the unused portion of a field.
Setf()	Set a formatting flag such as left justification and right justification, for output.
Unsetf()	Undo a flag specified.

i. **Width( ):** The function width() returns the current width. The optional *w* can be used to set the width. Width is defined as the minimum number of characters to display with each output.

Syntax

```
int width();
int width(int w);
```

For example:
```
cout.width(5);
    cout << "2";
```

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

Output: 2 (that's four spaces followed by a '2')

The width() can specify the field width for only one item, the one that follows immediately. After printing one item, it reverse back to the default.

ii. **Precision():** The function precision() is used to define the precision of the display of floating point numbers. The function expects the number of digits (***not*** counting the decimal point or the minus sign) that are to be displayed as its argument. The default precision is 6.

Syntax

```
streamsize precision();
streamsize precision(streamsize p);
```

For example

1. Output: 314.16
```
float num = 314.15926535;
  cout.precision(5);
cout << num;
```

2. Output: 1.414
```
cout.precision(4);
  cout << sqrt(2) << endl;
```

3. Output : -1.41421
```
  cout.precision(6);
cout << -sqrt(2) << endl;
```

when used without argument, precision() returns the actual precision value:

```
cout.precision(4);
cout << cout.precision() << ", " << sqrt(2) << endl;
Unlike width(), precision()retains the setting in effect until it is
reset.
```

iii. **Fill():** The function fill() either returns the current fill character, or sets the current fill character to *ch*. The fill character is defined as the character that is used for padding when a number is smaller than the specified width. The default fill character is the space character.

Syntax

```
char fill();
char fill(char ch);
```

```
#include<iostream>
using namespace std;
int main()
{   cout.width(10);
    cout.fill('.');
    cout << -5 << endl;
    cout.width(10);
    cout << -5 << endl;
    return 0;
}
```

Output: ....-5
........-5

fill() stays in effect until changed.

iv. **Setf():** The member-function setf() is used to define the way numbers are displayed. It expects one or two arguments, all **flags** are of the iostream class.

In the following examples, cout is used, but other ostream objects might have been used as well:

a. To display the numeric base of integral values, use

```
cout.setf(ios::showbase)
```

This results in *no* prefix for decimal values, 0x for hexadecimal values, 0 for octal values. *For example*

```
cout.setf(ios::showbase);
cout << 16 << ", " << hex << 16 << ", " << oct << 16 << endl;
```

result is 16, 0x10, 020

b. To display a trailing decimal point and trailing decimal zeros when real numbers are displayed, use

```
cout.setf(ios::showpoint)
```

For example:

```
cout.setf(ios::showpoint);
cout << 16.0 << ", " << 16.1 << ", " << 16 << endl;
```

result is 16.0000, 16.1000, 16

Note that the last 16 is an integer rather than a real number, and is not given a decimal point.

c. If ios::showpoint is not used, then trailing zeros are discarded. If the decimal part is zero, then the decimal point is discarded as well.

Comparable to the dec, hex and oct manipulators

```
cout.setf(ios::dec, ios::basefield);
cout.setf(ios::hex, ios::basefield);
```

or `cout.setf(ios::oct, ios::basefield);` can be used.

d. To control the way real numbers are displayed cout.setf(ios::fixed, ios::floatfield) or cout.setf(ios::scientific, ios::floatfield) can be used. These settings result in, respectively, a fixed value display or a scientific (power of 10) display of numbers.

For example:

```
cout.setf(ios::fixed, ios::floatfield);
cout << sqrt(200) << endl;
cout.setf(ios::scientific, ios::floatfield);
cout << sqrt(200) << endl;
```

result is 14.142136
1.414214e+01

```
#include<iostream>
using namespace std;
int main()
{ cout.setf(ios::right, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  cout.setf(ios::left, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  cout.setf(ios::internal, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  cout.width(10);
  cout.setf(ios::right, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  cout.width(10);
  cout.setf(ios::left, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  cout.width(10);
  cout.setf(ios::internal, ios::adjustfield);
  cout << '[' << -55 << ']' << endl;
  return 0;
 }
```

Output

```
[-55]
[-55]
[-55]
[   -55]
[-55   ]
[-   55]
```

v. **Unsetf():** To turn a flag off, use the **unsetf()** function:

```
cout.setf(ios::showpos); // turn on the ios::showpos flag
cout << 27 << endl;
cout.unsetf(ios::showpos); // turn off the ios::showpos flag
cout << 28 << endl;
```

This results in the following **output**:

+27
28

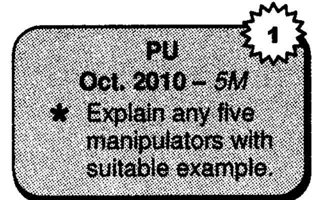
PU
Oct.09, Apr.10 – 5M
* Write short note on User Defined Manipulators.
* What are user defined manipulators? Illustrate with sample program.

6. Manipulators

You can use the manipulators for some simple formatted I/O. Manipulators are functions which are written in such a way that by placing a manipulator in the chain of **<<** operators, you can alter the state of the stream. C++ provides various stream manipulators that perform formatting tasks such as setting field widths, setting precision, setting and unsetting format *state,* setting the *fill character* in fields, flushing streams, inserting a *newline* into the output stream (and flushing the stream), inserting a *null character* into the output stream, *skipping whitespace* in the input stream. The following are some of the manipulators available in the **iostream** package. The manipulators that take arguments are declared in the file **iomanip.h** and the rest are in **iostream.h**.

Available manipulators in C++		
Manipulator	**Sample usage**	**Effect**
dec	cout << dec << intvar; or cin >> dec >> intvar;	Converts integers into decimal digits. Similar to the %d format in C.
hex	cout << hex << intvar; or cin >> hex >> intvar;	Hexadecimal conversion as in ANSI C's %x format.
oct	cout << oct << intvar; or cin >> oct >> intvar;	Octal conversion (%o in C).
ws	cin >> ws;	Discards whitespace characters in the input stream.
endl	cout << endl;	Sends newline to ostream and flushes buffer.
ends	cout << ends;	Outputs a null.
flush	cout << flush;	Flushes ostream's buffer.
resetiosflags(long)	cout << resetiosflags (ios::dec); or cin >>resetiosflags(ios::hex);	Resets the format bits specified by the long integer argument.
setbase(int)	cout << setbase(10); or cin >> setbase(8);	Sets base of conversion to integer (argument must be 0, 8, 10, or 16). Zero sets base to default.
setfill(int)	cout << setfill('.'); or cin >> setfill(' ');	Sets the fill character used to pad fields (width comes from setw).
setiosflags(long)	cout << setiosflags(ios::dec); or cin >> setiosflags(ios::hex);	Sets the format bits specified by the long integer argument.
setprecision(int)	cout << setprecision(6); or cin >> setprecision(15);	Sets the precision of floating-point conversions to the specified number of digits.
setw(int)	cout << setw(6) << var; or cin >> setw(24) >> buf;	Sets the width of a field to the specified number of characters.

i. **dec, hex, oct:** These manipulators enforce the input/output of integral numbers in, respectively, decimal, hexadecimal and octal format. The default conversion is decimal. The conversion takes effect on information inserted into the stream after processing the manipulators.




```
#include<iostream>
using namespace std;
int main()
{ int i = 100;
  cout << dec << i << endl;
  cout << hex << i << endl;
  cout << oct << i << endl;
  return 0;
}
```

Output

100
64
144

ii. **setbase(int b):** This manipulator can be used to display integral values using the base 8, 10 or 16. It can be used instead of oct, dec, hex in situations where the base of integral values is parameterized.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ cout << setbase(16);
  cout << 100 << endl;
  return 0;
}
```

This code uses setbase manipulator to set hexadecimal as the basefield. The output of this example is the hexadecimal value of 100, i.e., **64**.

iii. **setw(int width):** This manipulator expects as its argument the width of the field that is inserted or extracted next. It can be used as manipulator for insertion, where it defines the maximum number of characters that are displayed for the field, and it can be used with extraction, where it defines the maximum number of characters that are inserted into an array.

For example: To insert 20 characters into cout, use: `cout<<setw(20)<<8<< endl;`

To prevent array-bounds overflow when extracting from cin, setw() can be used as well:

`cin >> setw(sizeof(array)) >> array;`

A nice feature here is that a long string appearing at cin is split into substrings of at most sizeof(array) - 1 characters, and an ascii-z is appended.

Note

a. setw() is valid *only* for the next field. It does *not* act like, e.g., hex which changes the general state of the output stream for displaying numbers.

b. When setw(sizeof(someArray)) is used, make sure that someArray really is an array, and not a pointer to an array: the size of a pointer, being 2 or 4 bytes, is usually not the size of the array that it points to....

c. In order to use setw() the header file iomanip must be included.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{  int i = 100;
   cout << setw(6) << dec << i << endl;
   cout << setw(6) << hex << i << endl;
   cout << setw(6) << oct << i << endl;
   return 0;
 }
```

Output

```
   100
    64
   144
```

Here each variable is displayed in a six-character field aligned at the right and padded with blanks at the left.

iv. **setfill (int ch):** This manipulator defines the filling character in situations where the values of numbers are too small to fill the width that is used to display these values. By default the blank space is used.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{  int i = 100;
   cout << setfill('.');
   cout << setw(6) << dec << i << endl;
   cout << setw(6) << hex << i << endl;
   cout << setw(6) << oct << i << endl;
   return 0;
 }
```

Output

```
...100
....64
...144
```

The default alignment of fixed-width output fields is to pad on the left, resulting in right-justified output. The justification information is stored in a bit pattern called the **format bits** in a class named **ios**, which forms the basis of all stream classes.

v. **setprecision(int width):** This manipulator can be used to set the precision in which a float or double is displayed. In order to use manipulators requiring arguments the header file iomanip must be included.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main() {  double f =3.14159;
  cout << setprecision(5) << f << endl;
  cout << setprecision(9) << f << endl;
  return 0;
}
```

Output

3.1416
3.14159

vi. **resetiosflags(Reset format flags):** Unsets the format flags specified by parameter *mask*.Behaves as a call to stream's member: `setf(0,mask)`;

You must include <iomanip> to use this manipulator.

Parameters

Mask: Mask representing flags to be reset. This is object of type `ios_base::fmtflags`. This function should only be used as stream manipulator.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main() { cout << hex << setiosflags(ios_base::showbase);
  cout << 100 << endl;
  cout << resetiosflags(ios_base::showbase);
  cout << 100 << endl;
  return 0;
}
```

This code first sets flag showbase then resets it using resetiosflags manipulator.

Output

0x64
64

6.1 Correspondence between iostream.h and iomanip.h

Correspondence between **iostream.h** methods (functions) and **iomanip.h** manipulators is as follows:

Correspondence between Iomanip and Iostream	
iomanip.h	iostream.h
setiosflags(...)	setf(...)
resetiosflags(...)	unsetf(...)
setbase(10)	setf(ios::dec)
setbase(8)	setf(ios::oct)
setbase(16)	setf(ios::hex)
setfill('.')	fill('.')
setprecision(2)	precision(2)
setw(20)	width(20)

To use any of the format flags in the following table, insert the manipulator **setiosflags** with the name of the flag as the argument. Use **resetiosflags** with the same argument to revert to the format state before you use the **setiosflags** manipulator.

Name of flag	Meaning when flag is set
ios::skipws	Skips white space on input
ios::left	Left justifies output within the specified width of the field
ios::right	Right justifies output
ios::scientific	Uses scientific notation for floating point numbers (such as -1.23e+02)
ios::fixed	Uses decimal notation for floating-point numbers (such as -123.45)
ios::dec	Uses decimal notation for integers
ios::hex	Uses hexadecimal notation for integers
ios::oct	Uses octal notation for integers
ios::uppercase	Uses uppercase letters in output (such as F4 in hexadecimal, 1.23E+02)
ios::showbase	Indicates the base of the number system in the output (a 0x prefix for hexadecimal and a 0 prefix for octal)
ios::showpoint	Includes a decimal point for floating-point output (*For example:* –123.)
ios::showpos	Shows a positive sign when display positive values.
ios::internal	Padding after sign or base indicator
ios::unitbuf	Flush all streams after insertion
ios::stdio	Flush stdout, stderr after insertion

6.2 Difference between manipulators and ios functions [Oct. 11 5M]

	IOS functions	Manipulators
i.	These are like the normal functions which are invoked upon calling.	Manipulators are the instructions to the output stream to modify the output in various ways.
ii.	The setf() function is used to set the flags of IOS.	Manipulators directly insert the formatting instructions into the stream.
iii.	These functions use data members of IOS class only.	This is not the case with manipulators.
iv.	We cannot create user defined members or functions.	We can create user defined manipulators.
v.	The flags put on through the setf( ) function can be put off through unsetf( ) function.	Such flexibility is not available with manipulators.
vi.	*Example:* `setf(), unsetf(), precision(), width()`	*Example:* `setbase(), setfill(), setprecision(), setwidth(), dec, hex`

Solved Programs

1. Program prints out a table of square roots for the numbers 1 to 10.

Solution

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{ int number;
  cout.width(20);
  cout << "Number" << "Square Root\n\n";
  cout.setf(ios::fixed);
  cout.precision(2);
  for(number = 1 ; number <= 10 ; number = number + 1) {
    cout.width(20);
    cout << number << sqrt((double)number) << "\n"; }
  return 0;
}
```

2. What will be the output of following program?

PU Oct. 2009 – *2M*

i.
```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
void main()
{
  int i = 1350;
  float f = 302.250;
  cout<<setiosflags(ios::showbase|ios::uppercase);
  cout<<i<<endl;
  cout.precision(4);
  cout<<setiosflags(ios::showpoint|ios::scientific)<<f<<endl;
  cout<<resetiosflags(ios::scientific);
  cout<<setprecision(3)<<sqrt(2);
}
```

Solution

Output:

1350

3.0335E+02

1.414

ii.

```
#include<iostream.h>
void main()
{
        char s[]= "program";
        int i;
        for(i=0;s[i];i++)
                cout<< "\n"<<s[i++]<<*(s+i);
}
```

PU Oct. 2009 – *2M*

Solution

Output:

pp
oo
rr
mm

iii.

```
#include<conio.h>
#include<iostream.h>
#include<math.h>
void main()
{
   cout.setf(ios::left, ios::adjustfield);
   cout.width(9);
   cout.fill('#');
   cout.precision(5);
   cout <<-5.25;

   cout << "\n";
   cout.setf(ios::right, ios::adjustfield);
   cout.width(9);
   cout.fill('#');
   cout.precision(5);
   cout <<-5.25;
}
```

PU Apr. 2010 – *2M*

Solution

Output of the given code is:

-5.25####

####-5.25

Explanation: cout.setf(ios::left, ios::adjustfield); → causes the output to be left aligned and adjust the remaining fields (if any)

a. cout.width(9); → sets the field width to 9 spaces (characters)

b. cout.fill('#'); → causes the remaining fields to be filled with '#'

c. cout.precision(5); → sets the precision to 5

d. cout <<-5.25; → displays -5.25 (left aligned) and 4 times '#', i.e., **-5.25####**

e. cout.setf(ios::right, ios::adjustfield); → causes the output to be right aligned and adjust the remaining fields (if any)

f. cout <<-5.25; → displays 4 times '#' and -5.25 (right aligned), i.e., **####-5.25**

iv.

```
#include<conio.h>
#include<iostream.h>
#include<iomanip.h>
void main()
{
        int x=100;
        float f=56.75;
        cout<<hex<<x<<dec<<x<<endl;
        cout<<setw(8)<<setfill('0')<<f;
}
```

PU
Oct. 2010 – *2M*

Solution

Output of the given code is:

64100 (64 and 100, without any space in between them)

00056.75

Explanation: The code cout<<hex<<x would display the value of 'x' in hexadecimal number system. Here, x=100 is given and 64 is the hexadecimal equivalent of 100 (in decimal number system).

The code cout<<dec<<x would display the value of 'x' in decimal number system. So the output is 100.

The code cout<<setw(8)<<setfill('0')<<f; would set the width to 8 and set the fill character to '0'. The value of f (i.e., 56.75) needs 5 spaces to be displayed. Since width is set to 8 spaces, leading three spaces would be filled by '0'. So the output is 00056.75.

EXERCISES

A. Review Questions

1. What is a stream?
2. Explain the stream classes in C++.
3. Write a short note on unformatted I/O operations.
4. Write a short note on formatted I/O operations.
5. What is the role of fill() function?
6. Discuss the syntax of set() function?

B. Programming Exercises

1. Write a program to read a list containing item name, item code, and cost interactively and produce a three-column output as shown below.

Name	Code	Cost
Turbo C++	22389	300.00
C Primer	45646	95.00
. .	. .	. .

Note that the name and code are left justified and the cost is right justified with a precision of two digits.

2. State errors, if any, in the following statements:

i. `cout << (void*) amount;`

ii `cout << width();`

iii. `cout.width(10).precision(3);`

iv. `cout. Setf(ios::scientific, ios::left);`

Collection of Questions asked in Previous Exams PU

1. What will be the output of following program?

i. [Oct. 2009 – 2M]

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
void main()
{
  int i = 1350;
  float f = 302.250;
  cout<<setiosflags(ios::showbase|ios::uppercase);
  cout<<i<<endl;
  cout.precision(4);
  cout<<setiosflags(ios::showpoint|ios::scientific)<<f<<endl;
  cout<<resetiosflags(ios::scientific);
  cout<<setprecision(3)<<sqrt(2);
}
```

ii. [Oct. 2009 – 2M]

```
#include<iostream.h>
void main()
{
  char s[ ]= "program";
  int i;
  for(i=0;s[ i ];i++)
        cout<< "\n"<<s[i++]<<*(s+i);
}
```

iii. [Apr. 2010 – 2M]

```
#include<conio.h>
#include<iostream.h>
#include<math.h>
void main()
{
  cout.setf(ios::left, ios::adjustfield);
  cout.width(9);
  cout.fill('#');
  cout.precision(5);
  cout <<-5.25;
  cout << "\n";
  cout.setf(ios::right, ios::adjustfield);
  cout.width(9);
  cout.fill('#');
  cout.precision(5);
  cout <<-5.25;
}
```

iv. [Oct. 2010 – 2M]

```
#include<conio.h>
#include<iostream.h>
#include<iomanip.h>
void main()
{
     int x=100;
     float f=56.75;
     cout<<hex<<x<<dec<<x<<endl;
     cout<<setw(8)<<setfill('0')<<f;
}
```

2. Write Short note on User Defined Manipulators. [Oct. 2009 – 5M]
3. What are user defined manipulators? Illustrate with sample program. [Apr. 2010 – 5M]
4. Explain any five manipulators with suitable example. [Oct. 2010 – 5M]
5. Difference between manipulators and ios functions. [Oct. 2011 – 5M]

10 Working With Files

I. Introduction

All the programs we have seen so far use input only from the keyboard, and output only to the screen.

If we were restricted to use only the keyboard and screen as input and output devices, it would be difficult to handle large amount of input data, and output data would always be lost as soon as we turned off the computer.

To avoid these problems, we can store data in some secondary storage device, usually magnetic tapes or discs.

Data can be created by one program, stored on these devices, and then accessed or modified by other programs when necessary.

To achieve this, the data is packaged on the storage devices as data structures called *files*.

The easiest way to think about a file is as a linear sequence of characters.

There are two types of data files:

Types of Data Files
i. Sequential access file
ii. Random access file

i. **Sequential Access Files:** These files must be accessed in the same order in which they were written. This process is analogous to audio cassette tapes where you must fast forward or rewind through the songs sequentially to get to a specific song.

In order to access data from a sequential file, you must start at the beginning of the file and search through the entire file for the data that you want.

ii. **Random Access Files:** These files are analogous to audio compact disks where you can easily access any song, regardless of the order in which the songs were recorded. Random access files allow instant access to any data in the file.

Unfortunately, random access files often occupy more disk space than sequential access files.

File I/O Classes

There are 3 File I/O classes in C++ which are used for File Read/Write operations. They are:

- ifstream can be used for File read/input operations (derived from istream)
- ofstream can be used for File write/output operations (derived from ostream)
- fstream can be used for both read/write C++ file I/O operations (derived from iostream)

To use any of these classes, you must have the following include statement in your program:

`#include<fstream.h>` *or* `#include<fstream>`

This #include automatically includes the header file iostream.

2. Creating a Stream

In input/output of C++ handles file operations which are very much similar to the console input/output operations. A file stream is an interface between the programs and the files.

The stream that supplies data to the program is known as input and one that receives data from the program is called as output stream.

In short, the input stream reads data from file and the output stream writes data to the file. In C++, you open a file by linking it to a stream.

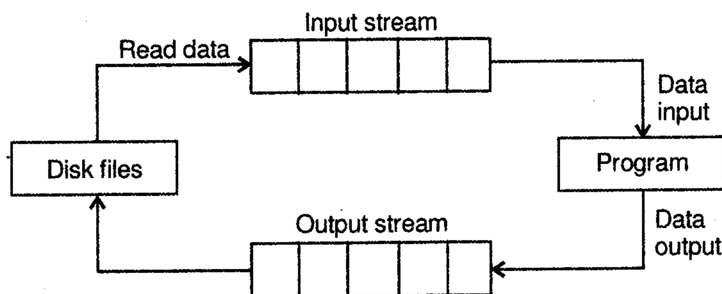


Figure 10.1: File input/ output stream

There are three types of streams: input, output and input/output. To create an input stream, you must declare the stream to be of class ifstream. To create an output stream, you must declare it as class ofstream.

Streams that will be performing both input and output operations must be declared as class fstream. *For example*: This fragment creates one input stream, one output stream and one stream capable of both input and output.

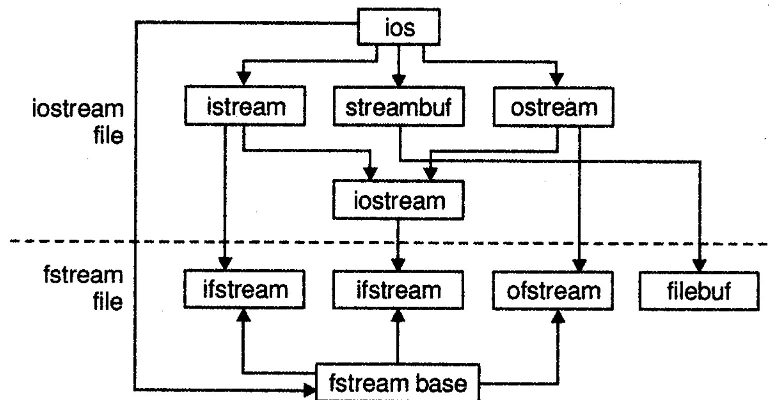


Figure 10.2

```
ifstream in; // input
ofstream out; // output
fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using open() function.

3. Opening a File

A file can be opened in two ways: by using the constructor function of the class and by using member function open() of the class. The first method is useful when we use only one file in the stream and the second method is used when we want to manage multiple files using one stream.

3.1 Opening Files using Constructor

Generally constructor is used to initialize an object while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

i. Create a file stream object to manage the stream using the appropriate class (as explained above).

ii. Initialize the file object with the desired filename.

For example: `ofstream out("outfile"); //output only`

The above statement will create a **out** as an ofstream object that manages the output stream.

This object can be any valid C++ names such as o_file, myfile or fout. This statement also opens the file **outfile** and attaches it to the output stream *out*.

Similarly,

```
ifstream in("infile");  //input only
```

this statement will create **in** as ifstream object and attaches it to the file **infile** for reading (input);

In the above method the functions for reading and writing a stream we have used only one argument i.e., filename. However these functions can take two arguments, the second one for specifying the file mode. The general form of the function open() with two arguments is:

```
stream-object.open("filename", mode);
```

The second argument *mode* (called file mode parameter) specifies the purpose for which the file is opened.

Modes for file open	
Mode name	**Operation**
ios::app	Appends data to the end of file.
ios::ate	When first opened, positions file at end-of-file (ate stand for at end).
ios::in	Opens file for reading.
ios::nocreate	Open fails if the file does not already exist.
ios::noreplace	If file exists, open for output fails unless ios::app or ios::ate is set.
ios::out	Opens file for writing.
ios::trunc	Truncates file if it already exists.
ios::binary	Opens file in binary mode.

We can combine two or more flags or (modes) by using bitwise operator OR(|).

For example: To open a file for output and position it at the end of existing data, we can write statement as follows:

```
ofstream outs("outfile",ios::out|ios::ate);
```

If we do not provide the value for the mode parameter then it will use the default values shown in the following table.

Class	Default mode to parameter
ofstream	ios::out | ios::trunc
ifstream	ios::in
fstream	ios::in | ios::out

That is for an ifstream (open for reading) the default mode is ios::in and for an ofstream (open for writing) the default mode is ios::out or ios::trunc etc.

3.2 Opening a File using Open()

As stated earlier, the function open() can be used to open multiple files that use the same stream object. *For example*: We may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

```
file_stream_class stream_object;
stream_object.open("filename");
```

Example:

```
ofstream file;                //create stream (for output)
file.open("example");         //connect stream to example
```

3.3 Points to remember when using Modes of File

i. Opening a file in ios::out mode also opens it in the ios::trunc mode by default. The ios:: trunc value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length. When creating an output stream using ofstream, any preexisting file by that name is automatically truncated.

ii. Including ios::ate causes a seek to the end of file to occur when the file is opened; but I/O operations can still occur anywhere within the file.

iii. The ios::binary value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations may take place but no such thing occurs in the binary mode.

4. Closing a File

The member function close() is used to close a file which has been opened for file processing such as to read, to write and for both. The close() member function is called automatically by the destructor functions. However one may call this member function to close the file explicitly. The close member function will not contain any arguments nor does it return any value.

The general **syntax** for close() is as follows:

```
#include<fstream>
void main()
{
   fstream infile;
   infile.open("datafile", ios::in || ios::out);
   . . .
   . . .
   infile.close();  // calling to close the file
}
```

5. Checking for Failure with File Commands

It may happen that there is an error while opening and closing file. So to avoid that we have to always include some check to make sure that file operations have completed successfully, and error handling routines in case they haven't. A simple checking mechanism is provided by the member function "fail()".

The function call

```
in_stream.fail();
```

returns True if the previous stream operation on "in_stream" was not successful (perhaps we tried to open a file which didn't exist).

If a failure has occurred, "in_stream" may be in a corrupted state, and it is best not to attempt any more operations with it.

The following example program fragment plays very safe by quitting the program entirely, using the "exit(1)" command from the library "cstdlib":

```
#include<iostream>
#include<fstream>
#include<cstdlib>
using namespace std;
int main()
{
  ifstream in_stream;
  in_stream.open("PQR.txt");
  if(in_stream.fail())
  {
    cout << "Sorry, the file couldn't be opened!\n";
    exit(1);
}
  ...
```

There is an alternate method. In the above example if open() fails, the stream will evaluate to false when used in a Boolean expression. This can be tested as follows:

```
if(!in_stream)
{
cout << "Sorry, Cannot open file.\n";
}
```

6. Detecting the End-of-File

While reading from a file, it is necessary to know whether the end-of-file is reached or not. The eof() member function is used to check whether a file pointer is reached at the end of a file or not . If it is successful, eof() function returns a nonzero, otherwise returns a zero.

Syntax

```
#include<fstream>
void main()
{
  ifstream infile;
  infile.open("text");
  while(!infile.eof())
  { . . .
      . . .
  }
}
```

In addition to **eof()**, other member functions exist to verify the state of the stream (all of them return a **bool** value):

i. **bad():** The bad() stream state member function is used to check whether any invalid file operations have been attempted or there is an unrecoverable error.

The bad() member function returns a nonzero if it is true; otherwise returns a zero.

Syntax

```
#include<fstream>
#include<cstdlib>
void main()
{  ifstream infile;
   infile.open("text");
   if(infile.bad())
   {
      cerr << "Open failure" << endl;
      exit (1);
   }
   . . .
   . . .}
```

ii. **fail():** The fail() stream state member function is used to check whether a file has been opened for input or output successfully, or any invalid operations are attempted or there is an unrecoverable error.

If it fails, it returns a nonzero character.

Syntax

```
#include<fstream>
void main()
{
   ifstream infile;
   infile.open("text");
   while(!infile.fail())
   {
      cout << "couldn't open a file" << endl;
      continue:
      . . .
      . . .
   }
}
```

iii. **good():** The good() stream state member function is used to check whether the previous file operation has been successful or not. The good() returns a nonzero if all stream state bits are zero.

Syntax

```
#include<fstream>
#include<cstdlib>
void main()
{
   ifstream infile;
   infile.open("text");
   if(infile.good())
   {
      . . .
      . . .
   }
}
```

7. File Pointers and their Manipulation

Each file has two associated pointers known as the file pointers. One of them is called the input pointer or get pointer and the other is called the output pointer or put pointer. We can use these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

All I/O streams objects have, at least, one stream pointer.

- `ifstream`, like `istream`, has a pointer known as *get pointer* that points to the next element to be read.
- `ofstream`, like `ostream`, has a pointer *put pointer* that points to the location where the next element has to be written.
- Finally `fstream`, like `iostream`, inherits both *get* and *put* pointers.

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

PU
Oct. 2011 – *5M*
★ Explain the role of seekg(), seekp(), tellp() and tellg() functions in the process of random access in a binary file.

i. **tellg() and tellp() :** These two member functions admit no parameters and return a value of type pos_type (according ANSI-C++ standard) that is an integer data type representing the current position of get stream (Input stream) pointer (in case of tellg) or put stream (Output stream) pointer (in case of tellp).

ii. **seekg() and seekp() :** This pair of functions serve respectively to change the position of stream pointers *get* and *put*. Both functions are overloaded with two different prototypes:

```
seekg(pos_type position);
seekp(pos_type position);
```

Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions tellg and tellp.

```
seekg(off_type offset, seekdir direction);
seekp(off_type offset, seekdir direction);
```

Using this prototype, an offset from a concrete point determined by parameter *direction* can be specified. It can be:

ios::beg	offset specified from the beginning of the stream
ios::cur	offset specified from the current position of the stream pointer
ios::end	offset specified from the end of the stream

The values of both stream pointers *get* and *put* are counted in different ways for text files than for binary files, since in text mode files some modifications to the appearance of some special characters can occur. For that reason it is advisable to use only the first prototype of **seekg** and **seekp** with files opened in text mode and always use non-modified values returned by **tellg** or **tellp**. With binary files, you can freely use all the implementations for these functions. They should not have any unexpected behavior. The following example uses the member functions just seen to obtain the size of a binary file:

```
// obtaining file size
#include<iostream.h>
#include<fstream.h>
const char * filename = "example.txt";
int main()
{
  long l,m;
  ifstream file;
  file.open(filename, ios::in|ios::binary);
  l = file.tellg();
  file.seekg(0, ios::end);
  m = file.tellg();
  file.close();
  cout << "size of " << filename;
  cout << " is " << (m-l) << " bytes.\n";
  return 0;
}
```

Output

size of example.txt is 40 bytes.

8. Reading / Writing a Character from a File

The following member functions are used for reading and writing a character from a specified file.

1. get()

This member function is used to read an alphanumeric character from a specified file.

Syntax

```
#include<fstream>
void main()
{
  ifstream infile;
  char ch;
  infile.open("text");
    . . .
    . . .
  while(!infile.eof())
  {
    ch = infile.get()
    . . .
    . . .
  } // end of while loop
}
```

ii. put()

This member function is used to write a character to a specified file or specified output stream.

Syntax

```
#include<fstream>
void main()
{
   ofstream outfile;
   char ch;
   outfile.open("text");
      . . .
      . . .
   while(!outfile.eof())
   {
      ch = outfile.get()
      cout.put(ch)  // display a character onto a screen
      . . .
      . . .
   }
}
```

Program using get() and put() function

```
#include<fstream>
#include<cstring>
using namespace std;
int main()
{
   char string[80];
   cout << "Enter a string \n";
   cin >> string;
   int len = strlen(string);
   fstream file;                         // input and output stream
   file.open("Text", ios::in | ios::out);
   for (int  i= 0; i<len; i++)
   file.put(string[i]);                  // put a character to file
   file.seekg(0);                        // goto the start
   char ch;
   while(file)
   {
      file.get(ch);                      // get a character from file
      cout << ch;                        // display it on screen
   }
   return 0;
}
```

Output

Enter a string :

Program using get() and put() // input

Program using get() and put() // output

9. write() and read() Functions

Another way to read and write blocks of binary data is to use C++'s read() and write() functions. The first one (**write**) is a member function of **ostream**, also inherited by **ofstream**. And **read** is member function of **istream** and it is inherited by **ifstream**. Objects of class **fstream** have both. Their prototypes are:

```
ostream &write(char * buffer, streamsize size);
istream &read(char * buffer, streamsize size);
```

where, *buffer* is the address of a memory block where the read data are stored or from where the data to be written are taken. The *size* parameter is an integer value that specifies the number of characters to be read/written from/to *the buffer.*

The following program writes a structure to disk and then reads it back in

```
#include<fstream>
#include<cstring>
using namespace std;
struct status{
  char name[80];
  double balance;
  unsigned long account_num; };
int main()
{
  struct status acc;
  strcpy(acc.name, "abc");
  acc.balance = 1123.23;
  acc.account_num = 3434;
  //write data
  ofstream outbal("balance", ios::out | ios::binary);
  if(!outbal)
  { cout << "cannot open file\n");
    return 1;
  }
outbal.write((char *) &acc, sizeof(struct status));
outbal.close();
//now read back;
ifstream inbal("balance", ios::in | ios::binary);
if(!inbal)
{ cout << "cannot open file\n");
  return 1;   }
inbal.read((char *) &acc, sizeof(struct status));
cout << acc.name << endl;
cout << "Account #" << acc.account_num;
cout.precision(2);
cout.setf(ios::fixed);
cout <<endl << "Balance: $" << acc.balance;
inbal.close();
return 0;
 }
```

Only a single call to read() or write() is necessary to read or write the entire structure. Each individual field need not be read or written separately.

If the end of the file is reached before **size** characters have been read, then read() simply stops, and the buffer contains as many characters as were available. To determine how many characters have been read, gcount() function can be used. Its prototype is:

```
streamsize gcount();
```

It returns the number of characters read by the last binary input operation.

10. Buffers and Synchronization

When we operate with file streams, these are associated to a *buffer* of type **streambuf**. This *buffer* is a memory block that acts as an intermediary between the stream and the physical file. *For example*: With an out stream, each time the member function **put** (writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in the *buffer* for that stream.

When the buffer is flushed, all data that it contains is written to the physic media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:

i. **When the file is closed:** Before closing a file all buffers that have not yet been completely written or read are synchronized.

ii. **When the *buffer* is full:** *Buffers* have a certain size. When the *buffer* is full it is automatically synchronized.

iii. **Explicitly with manipulators:** When certain manipulators are used on streams a synchronization takes place. These manipulators are: **flush** and **endl**.

iv. **Explicitly with function sync():** Calling member function **sync()** (no parameters) causes an immediate syncronization. This function returns an **int** value equal to **-1** if the stream has no associated *buffer* or in case of failure.

11. Other Functions

i. **ignore():** Used when reading a file. If you want to ignore certain amount of characters, just use this function. In fact, you can use seekg() instead, but the ignore() function has one advantage - you can specify a delimiter rule, where the ignore() function will stop. The prototype is:

```
istream& ignore(int nCount, delimiter);
```

where, *nCount* is the amount of characters to be ignored and *delimiter* is what its name says. It can be EOF if you want to stop at the end of the file. This way, this function is the same as seekg(). But it can also be '\n' *for example*: which will stop on the first new line. ignore() reads and discards characters until either nCount characters have been ignored (1 by default) or the character delimiter is encountered (EOF by default). Here is example:

```
#include<fstream.h>
void main()
{
    //if we have "Hello World" in test_file.txt
    ifstream File("test_file.txt");
    static char arr[10];
    //stop on the 6th symbol, if you don't meet "l"
    //in case you meet "l"- stop there
    File.ignore(6,'l');
    File.read(arr,10);
    cout << arr << endl; //it should display "lo World!"
    File.close();
}
```

ii. **getline():** This function can be used to read line-by-line, but it can be set to stop reading if it met a certain symbol. Here is how you should pass the parameters to it:

```
getline(array,array_size,delim);
```

The getline() function reads characters into the "array" until either array-size-1 characters have been read, the character specified by delim has been found, or the end of the file has been encountered. The array will be null terminated by getline(). If the delimiter counter is encountered in the input stream, it is extracted but not put into array.

And here is a code example:

```
#include<fstream.h>
void main()
{
  //if we have "Hello World" in test_file.txt
    ifstream File("test_file.txt");
    static char arr[10];
  /*read, until one of these happens:
    1) You have read 10
    2) You met the letter "o"
    3) There is new line
    */
    File.getline(arr,10,'o');
    cout << arr << endl; //it should display "Hell"
    File.close();
}
```

iii. **peek():** This function will return the next character from an input file stream, but it won't move the inside-pointer. I hope you remember, that get() *for example*: returns the next character in the stream, and after that, it moves the inside-pointer, so that the next time you call the get() function, it will return the next character, but not the same one. Well, using peek() will return a character, but it won't move the cursor. So, if you call the peek() function, two times in succession, it will return a same character. Consider the following code example:

```
#include<fstream.h>
void main()
{
    //if we have "Hello World" in test_file.txt
    ifstream File("test_file.txt");
    char ch;
    File.get(ch);
    cout << ch << endl; //should display "H"
    cout <<char(File.peek()) << endl; //should display "e"
    cout <<char(File.peek()) << endl; //should display "e" again
    File.get(ch);
    cout << ch << endl; //should display "e" again
    File.close();
}
```

The peek( ) function actually returns the ASCII code of the char, but not the char itself. So, if you want to see the character itself, you have to call it the way which is given in the above example.

iv. **_unlink():** Deletes a file. Include io.h in your program, if you are going to use this function. Here is a code example:

```
#include<fstream.h>
#include<io.h>
void main()
{   ofstream File;
    File.open("delete_test.txt"); //creates the file
    File.close();
    _unlink("delete_test.txt"); //deletes the file
    //tries to open the file, but if it does not exists
    //the function will return error ios::failbit
    File.open("delete_test.txt",ios::nocreate);
    //see if it returned it
    if(File.rdstate() == ios::failbit)
    cout << "Error...!\n"; //yup,it did
    File.close();
}
```

v. **putback():** This function will return the last read character, and will move the inside-pointer, one with –1 char. In other words, if you use get() to read a char, then use putback(), it will show you the same character, but it will set the inside-pointer with –1 char, so the next time you call get() again, it will again show you the same character. Here is a code example:

```
#include<fstream.h>
void main()
{
```

```
        //test_file.txt should have this text- "Hello World"
        ifstream File("test_file.txt");
        char ch;
        File.get(ch);
        cout << ch << endl; //it will display "H"
        File.putback(ch);
        cout << ch << endl; //it will again display "H"
    File.get(ch);
    cout << ch << endl; //it will display "H" again
    File.close();
}
```

vi. flush(): When dealing with the output file stream, the data you save in the file is not actually immediately saved in it. There is a buffer where it is kept, and when the buffer gets filled, then the data is put in the real file (on your disk). Then the buffer is emptied, and so on.

But if you want to save the data from the buffer, even if the buffer is still not full, use the flush() function. Just call it this way- FileHandle.flush(). And the data from the buffer will be put in the physical file, and the buffer will be emptied.And something in addition (advanced)- The flush() function calls the sync() function of the associated streambuf.

12. Random Access File Processing

Every open file has a position or a position indicator associated with it. This indicates the position where read and write operation takes place.

In all earlier programs, we read the file sequentially, i.e., incase of sequential access file, data are stored and retrieved one after another. The file pointer always moves from the starting of the file to the end of file. On the other hand, a random access file need not necessarily start from the beginning of the file and move towards the end of the file. Random access means moving the file pointer directly to any location in the file instead of moving it sequentially.

The random access approach is often used with the data base files. In order to perform both reading and modifying an object of a data base, a file should be opened with mode of access for both to read and to write. The header file <fstream> is required to declare a random access file. We know that fstream is a class which is based on both the classes of ifstream and ofstream. The fstream inherits two file pointers, one for input buffer and other for output buffer for handling a random access file both for reading and writing.

Declaring a Random Access File

The random access file must be opened with the following mode of access:

ios::in (to read a file), ios::out (to write a file), ios::ate(to append) and ios::binary (binary format).

The following program segment shows how a random access file is opened for both reading and writing.

```
#include<fstream>
```

```
void main()
{
  fstream file;
  file.open(fname, ios::in | ios::out|ios::ate|ios::binary);
  . . .
  . . .
}
```

It is essential to open a random access file with the above mode of access in order to perform read, write and append. The file should be declared, as a binary status as the data members of a class is stored in a binary format.

The fstream inherits the following member functions in order to move the file pointer in and around the data base.

ios::beg (moves the file pointer from the beginning of the file), ios::cur (moves the file pointer from the current file pointer position) and ios::end (moves the file pointer from the end of the file).

seekg(), seekp() , tellg() and tellp() member functions are also used to process a random access file.

For example: The following program segment shows the positioning of the file operation for a random access file.

```
#include<fstream>
  void main()
  {
    fstream infile
    _______________
    _______________
    infile.seekg(40; //goto byte number 40
    infile.seekg(40, ios::beg); //same as the above
    infile.seekg(0, ios::end); // goto end of file
    infile.seekg(0); // goto the start of the file
    infile.seekg(-1, ios:cur); // the file ptr is moved back end by one
  byte
  }
```

13. Updating a File: Random Access

Updating is a routine task in the maintenance of any data file. The updation would include one or more of the following tasks:

- Displaying the contents of a file
- Modifying an existing item
- Adding a new item
- Deleting an existing item

These actions require the file pointers to move to a particular location that corresponds to the item/ object under consideration. This can be easily implemented if the file contains a collection of items/ objects of equal lengths. In such cases, the size of each object can be obtained using the statement.

```
int object_length = sizeof(object);
```

Then, the location of a desired object, say the mth object, may be obtained as follows:

```
int location = m * object_length:
```

The **location** gives the byte number of the first byte of the mth object. Now, we can set the file pointer to reach this byte with the help of **seekg()** or **seekp().**

We can also find out the total number of objects in a file using the **object_length** as follows:

```
int n = file_size/object_length;
```

The file_size can be obtained using the function **tellg()** or **tellp()** when the file pointer is located at the end of the file.

Program illustrates how some of the task described above are carried out. The program uses the "STOCK.DAT" file which is already created for five items and performs the following operations on the file:

i. Adds a new item to the file.
ii. Modifies the details of an item.
iii. Displays the contents of the file.

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class INVENTORY
{
  char name[10];
  int code;
  float cost;
  public:
  void getdata(void)
  {
      cout << "Enter name: "; cin >> name;
      cout << "Enter code: "; cin >> code;
      cout << "Enter cost: "; cin >> cost;
  }
  void putdata(void)
  {
      cout << setw(10) << name
          << setw(10) << code
          << set precision(2) << setw(10) << cost
          << endl;
  }
}; // End of class definition
int main()
{
  INVENTORY item;
    fstream inoutfile;          // input/ output stream
  inoutfile.open("STOCK.DAT", ios::ate | ios:: in | ios::out |
ios::binary);
  inoutfile.seekg(0,ios::beg);    // go to start
  cout << "CURRENT CONTENTS OF THE STOCK" << "\n";
  while(inoutfile.read((char *) & item, sizeof(item)))
  {
      item.putdata();
  }
```

```
inoutfile.clear();              // turn of EOF flag

/* >>>>>>>>>>>>>>>>>> Add one more item <<<<<<<<<<<<<<<<<<<*/
cout << "\nADD AN ITEM\n";
item.getdata();
char ch;
cin.get(ch);
inoutfile.write((char *) & item, sizeof(item);
// Display the appended file
inoutfile.seekg(0);             // go to the start
cout << "CONTENTS OF APPENDED FILE \n";
while(inoutfile.read((char *) & item, sizeof(item))
{
  item.putdata();
}
// Find number of objects in the file
int last = inoutfile.tellg();
int n = last/sizeof(item);
cout << "Number of objects = " << n << "\n";
/* >>>>>>>>>>>>> MODIFY THE DETAILS OF AN ITEM <<<<<<<<<<< */
cout << "Enter object number to be updated \n";
int object;
cin >> object;
cin.get(ch);
int location = (object-1) * sizeof(item);
if(inoutfile.eof())
inoutfile.clear();
inoutfile.seekp(location);
cout << "Enter new values of the object \n";
item.getdata();
cin.get(ch);
inoutfile.write((char *) & item, sizeof(item)) << flush;
/* >>>>>>>>>>>>>> SHOW UPDATED FILE <<<<<<<<<<<<<<< */
inoutfile.seekg(0); //go to the start
cout <<"CONTENTS OF UPDATED FILE \n";
while(inoutfile.read((char *) & item, sizeof(item))
{
  item.putdata();
}
inoutfile.close();
return 0;
} // End of main
```

The output of program would be:

Current Contents of stock

AA	11	100
BB	22	200
CC	33	300
DD	44	400
XX	99	900

```
  Add an item
Enter name:   YY
Enter code:   10
Enter cost:   101

Contents of appended file
  AA      11      100
  BB      22      200
  CC      33      300
  DD      44      400
  XX      99      900
  YY      10      101
  Number of objects = 6
  Total bytes in the files = 96
  Enter object number to be updated
  6
  Enter new values of the object
  Enter name: ZZ
  Enter code: 20
  Enter cost: 201
Contents of updated file
  AA      11      100
  BB      22      200
  CC      33      300
  DD      44      400
  XX      99      900
  ZZ      20      201
```

We are using the fstream class to declare the file streams. The fstream class inherits two buffers, one for input and another for output and synchronizes the movement of the file, both the pointers move in tandem. Therefore, at any point of time, both the pointers point to the same byte.

Since we have to add new objects to the file as well as modify some of the existing objects, we open the file using **ios::ate** option for input and output operations. Remember, the option **ios::app** allows us to add data to the end of the file only. The **ios::ate** mode sets the file pointers at the end of the file when opening it. We must therefore move the 'get' pointer to the beginning of the file using the function **seekg()** to read the existing contents of the file.

At the end of reading the current contents of the file, the program sets the EOF flag on. This prevents any further reading from or writing to the file. The EOF flag is turned off by using the function **clear()**, which allows access to the file once again.

After appending a new item, the program displays the contents of the appended file and also the total number of objects in the file and the memory space occupied by them.

To modify an object, we should reach to the first byte of that object. This is achieved using the statements.

```
int location = (object-1) * sizeof(item);
inoutfile.seekp(location);
```

The program accepts the number and the new values of the object to be modified and updates it. Finally, the contents of the appended and modified file are displayed.

Remember, we are opening an existing file for reading and updating the values. It is, therefore, important that the data members are of the same type and declared in the same order as in the existing file. Since, the member functions are not stored, they can be different.

14. Command Line Arguments

Like C, C++ also supports a mechanism to pass arguments or parameters to main when it begins executing, i.e., at runtime.

These arguments are called as command line arguments because they are passed from the command line during run time.

The main() functions which we have been using up to now without any arguments can take two arguments as shown below:

```
main(int argc, char *argv[])
```

The first argument argc called as argument counter which is the number of arguments in the command-line. The second argument argv called as argument vector is an array of char type pointers that points to the command line arguments. The size of this array will be equal to the value of argc.

For example: For the command line

```
C> test marks results
```

The value of argc would be 3 and the argv would be an array of three pointers to strings as shown below:

argv[0] → test
argv[1] → marks
argv[2] → results

The first argument, i.e., argv[0] (or *argv) will always represent the command name that invokes the program, i.e., program name. Command line arguments begin with argv[1]. If the number of arguments will be fixed, the count, argc, should always be checked. Here, the second and third arguments, i.e., argv[1] (or *(argv+1)) and argv[2] (or *(argv+2)) will be used as file names in the file opening statement as shown below:

```
. . .
. . .
infile.open(argv[1]);      // open marks file for reading
. . .
. . .
outfile.open(argv[2]);     // open results file for writing
. . .
. . .
```

Following program illustrates the use of the command-line arguments for supplying the file names. The command line is

```
test POSIT NEGAT
```

The program creates two files called POSIT ans NEGAT using the command line arguments and a set of numbers stored in an array are written to these files. The positive numbers are written to the POSIT file and the negative numbers are written to the file NEGAT. The program then displays the contents of the files.

Program for Command-Line Arguments

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main(int argc, char* argv[])
{  int number[9]={1,2,-3,-4,5,-6,7,-8,-9};
   if(argc!=3)
   {  cout << "argc=" << argc << "\n";
      cout << "Error in arguments \n";
      exit(1);  }
   ofstream fout1, fout2;
   fout1.open(argv[1]);
   if(fout1.fail())
   {  cout << "Could not open the file" << argv[1] << "\n";
      exit(1);  }
   fout2.open(argv[2]);
   if(fout2.fail())
   {  cout << "Could not open the file" << argv[2] << "\n";
      exit(1);  }
   for(int i=0; i<9; i++)
   {  if(number[i] > 0)
         fout1 << number[i] << " ";   //write to POSIT file
      else
         fout2 << number[i] << " ";   //write to NEGAT file   }
   fout1.close();
   fout2.close();
   ifstream fin;
   char ch;
   for(i=1; i<argc; i++)
   { fin.open(argv[i]);
      cout << "Contents of" << argv[i] << "\n";
      do
      {     fin.get(ch);     //read a value
      cout << ch;      //display it
      }
      while(fin);
      cout << "\n\n";
      fin.close();
      }
  return 0;
}
```

Output

Contents of POSIT

1 2 5 7

Contents of NEGAT

-3 -4 -6 -8 -9

Solved Programs

1. Write a program which counts and displays the number of characters (including blanks) in its own source code file.

Solution

```
/* This program counts and displays the number of characters in itself.
*/
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ char character;
  char previous_character;
  int count = 0;
  ifstream in_stream;
  in_stream.open("Sheet4Ex3.cpp");
  in_stream.get(character);
  for( ; ! in_stream.fail() ; )/*alternative:'while(!in_stream.eof())'*/
  { count++;
    previous_character = character;
    in_stream.get(character);
  }
  in_stream.close();
  cout << "The total number of characters in 'Sheet4Ex3.cpp', ";
  cout << "is " << count << ".\n";
  if(count > 1)
  cout<<"The last character is a'" << previous_character<<"'.\n";
  return 0;
}
```

2. A program to illustrate the writing of a set of lines to a user defined file where name of the file is specified in the user defined mode.

Solution

```
// storing a text on a specified file
#include<fstream.h>
void main()
{ ofstream outfile;
  char fname[10];
  cout << "enter a file name to be opened ? \n";
  cin >> fname[10];
```

```
   outfile.open(fname);
   outfile << "this is a test \n";
   outfile << "program to store \n"
   outfile << "a set of lines on to a file \n";
   outfile.close();
}
```

3. A program to read a set of lines from the keyboard and to store it on a specified file.

Solution

```
// reading a text and store it on a specified file
#include<fstream.h>
#define MAX 2000
void main()
{  ofstream outfile;
   char fname[10], line[MAX];
   cout << "enter a file name to be opened ?\n";
   cin >> fname;
   outfile.open(fname);
   char ch;
   int i;
   cout << "enter a set of lines and terminate with @\n";
   cin.get(line MAX, '@');
   cout << "given input \n";
   cout << line;
   cout << "String onto a file \n";
   outfile << line;
   outfile.close();
}
```

4. A program to copy the contents of a text file into another.

Solution

```
// file copy
#include<fstream.h>
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
void main()
{  ofstream outfile;
   itstream infile;
   char fname1[10], fname2[10];
   char ch;
   cout << "enter a file name to be copied ? \n";
   cin >> fname1;
   cout << "new file name ? \n";
   cin >> fname2;
   infile.open(fname1);
   if(infile.fail())
   {  cerr << "No such file exists \n";
      exit(1);}
   outfile.open(fname2);
   if(outilfe.fail())
   {  cerr << "unable to create a file \n";
      exit(1);
```

```
   }
   while(!infile.eof())
   {  ch = (char)infile.get();
      outfile.put(ch);
   }
   infile.close();
   outfile.close();
} // End of main
```

5. A program to perform the deletion of white spaces such as horizontal tab, vertical tab, space, line feed, new line and carriage return, from a text file and to store the contents of the file without white spaces on another file.

Solution

```
// deleting white spaces from a file
#include<fstream.h>
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
void main()
{ ofstream outfile;
  ifstream infile;
  char fname1[10], fname2[10];
  char ch;
  cout << "enter a file name to be copied ? \n";
  cin >> fname1;
  cout << "new file name ? \n";
  cin >> fname2())
  {
    infile.open(fname1);
    if(infile.fail())
  {  cerr << "unable to create a file \n";
     exit(1);
  }
  while(!infinile.eof())
  { ch = (char)infile.get();
   if (ch == ' ' || ch == '\t' || ch == '\n')infile
  seekg(1,ios::cur);
   else
      outfile.put(ch);
}
infile.close();
outifile.close();
} // End of main
```

Output

input file
this is a test
program by ravic
of iti palghat
output file
thisisatesprogrambyravicofitipalghat

6. A program to convert a lower case character to an upper case character of a text file.

Solution

```
// converting a lower case to upper case letter
#include<stream.h>
#include<iostream.h>
#include<iomanip.h>
#include<stdlib.h>
#include<ctype.h>
void main()
{  ofstream outfile;
   ifstream infile;
   char fname1[10], fname2[10];
   char ch, uch;
   cout <<"enter a file name to be copied ? \n";
   cin >> fname1;
   cout <<"new file name ?\n";
   cin >> fname2;
   infile(infile.fail())
   {
     cerr <<"unable to create a file \n";
     exit(1);
   }
   outfile.open(fname2);
   if(outfile.fail())
   {  cerr <<"unable to create a file \n";
      exit(1);
   }
   while(!infile.eof())
   {
     ch = (char)infile.get();
     uch = toupper(ch);
     outfile.put(uch);
   }
   infile.close();
   outfile.close();
} // End of main
```

Output

input file
this is a test
program by raci
of iti palghat
output file
THIS IS A TEST
PROGRAM BY RAVIC
OF ITI PALGHAT

7. Design a class student having following data members.

- **Roll No**
- **Name**
- **Marks**

Write a menu driven program to do following task by using the concept of file I/O in C++

i. Add a student record ii. Display student record iii. Modify student record

Solution

PU
Oct. 2009 – *10M*

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include<conio.h>
class student
{
  private:
    int roll_no;
    char name[21];
    float marks;
  public:
    void getdata(void)
    {
      cout << "\n Enter Name  : ";
      cin >> name;
      cout << "\n Enter Roll No ;";
      cin >> roll_no;
      cout << "\n Enter the Marks : ";
      cin >> marks;
    }
    void putdata(void)
    {
      cout << setw(10) << name
        << set(10) << roll_no
        << setprecision(2) << setw(10) << marks
        << endl;
    }
};
void main()
{
  student rec;
  int ch;
  fstream infile;
  clrscr();
  infile.open("Students.dat",ios::ate | ios::in | iso::out |
ios::binary);
  infile.seekg(0,ios::beg);
cout << "\n Students Data Based Management System";
  cout << "\n 1 for Addition of Record";
  cout <<"\n 2 for Display Record";
  cout << "\n 3 Modify the Record";
  cout << "\n You can select any one option at a time";
  cout << "\n What is your option";
  cin >> ch;
  switch(ch)
```

```
    {
      case 1:
        rec.getdata();
        infile.write((char*) & rec, sizeof(rec));
        infile.seekg(0);
        break;
      case 2:
        while(infile.read((char*) & rec,sizeof(rec))
        {
          rec.putdata();
        }
        break;
        infile.clear();
      case 3:
        cout << "\n Enter object number to be updated ";
        int num;
        cin >> num;
        int location = (num-1) *sizeof(rec);
        if(infile.eof())
        infile.clear();
        infile.seekp(num);
        cout <<"\n Enter new values of the Object ";
        rec.getdata();
        cin.get(num);
        infile.write((char*) & rec, sizeof(rec)<< flush)
        break;
    }
}
```

8. A company has following details of their employees in the file "emp.dat".

1.	**EmpID**	**2.**	**Emp Name**
3.	**Emp Address**	**4.**	**Emp Dept (Admin/Sales/Production/IT)**
5.	**Emp phone**	**6.**	**Emp Age**

Write a program to read the above file. Create new files such as Adm.dat, Sal.dat, Pro.dat, IT.dat respectively, to store the employee details according to their department.

1 PU Apr. 2010 – *10M*

Solution

```
/* Header file inclusion*/
#include<iostream.h>
#include<fstream.h>
/* Declaration of Employee Class */
class Employee
{
  char empId[10];
  char ename[30], address[40], dept[15], phone[15];
  float age;
};
int main()
{
  /* File object declaration */
```

```
  fstream mainFile, admFile, salFile, proFile, itFile;

  /* Opening File */
  mainFile.open("emp.dat",ios::in);  //reading only
  admFile.open("Adm.dat",ios::out); // writing only
  salFile.open("Sal.dat",ios::out); // writing only
  proFile.open("Pro.dat",ios::out); // writing only
  itFile.open("IT.dat",ios::out); // writing only
  /* reading Emp.dat record by record.*/
  while(mainFile)
  {
    Employee emp;
    mainFile.read((char *) &emp, sizeof(emp));
    /*Checking the department and writing record to the corresponding
        file*/
    if(emp.dept == "Adm")
      admFile.write((char *) &emp,sizeof(emp));
    else if(emp.dept == "Sal")
      salFile.write((char *) &emp,sizeof(emp));
    else if(emp.dept == "Pro")
      proFile.write((char *) &emp,sizeof(emp));
    else
      itFile.write((char *) &emp,sizeof(emp));
  }
  return 0;
}
```

9. Write a program that reads the text file and replace all the occurrences of vowels with '*' and copy the contents in new file. Also display the no. of replacements made.

PU
Oct. 2010 – *10M*

Solution

/* ASSUMPTIONS:

i. Name of the file to be read → source.txt

ii. Name of the file in which data would be copied → dest.txt

iii. source.txt is stored on the disk with data

*/

```
/* header file inclusion */
#include<iostream.h>
#include<fstream.h>
int main()
{
  char ch;
  int replacement=0;
  /* file object declaration */
  fstream srcFile, destFile;
    /* opening files */
  srcFile.open("source.txt", ios :: in);//reading only
  destFile("dest.txt", ios :: out);//writing only
  /* reading source.txt character by character */
  while(srcFile)
  {
```

```
    srcFile.get(ch) //reads single character  at a time from
  source file
      /*check for vowel */
    if(ch == 'a' || ch =='A'||ch =='e'||ch =='E'||ch
=='i'||ch =='I' ||
      ch =='o'||ch == 'O'||ch =='u'||ch =='U')
        {
          destFile.put('*'); //if character is vowel, then
  //replace it by *
          replacement++; //increment replacement counter
        }
        else
        {
          destFile.put(ch); //if character is not vowel
  //then don't replace it by*
        }
    } // end of while loop
      /* close files */
      srcFile.close();
      destFile.close();
    /* display number of replacements made */
    cout<<"Number of vowels replaced by*are:"<< replacement;
}//end of main function
```

EXERCISES

Programming Exercises

1. Write a program in C++ to read a file and to
   i. Display the contents of the file on to the screen,
   ii. Display the number of characters and
   iii. The number of lines in the file.
2. Write a program in C++ to read a file and to display the contents of the file.
3. Write a program in C++ to merge two files into a one file heading.
4. Write a program in C++ to read students record such as name, sex, roll number, height and weight from the specified file and to display in a sorted order (name is the key for sorting).
5. There are 100 records present in a file with each record containing a 6-character item code, a 20-character item name and an integer quantity. Write a program to read these records, arrange them in the ascending order and write them in the same file overwriting the earlier records.

6. Write a C++ program that compares two text files and print the lines where they first differ. The program reads one line each from two files until a differing line is found or one of the files is exhausted, in which case the line read from the other file is the first differing line. If both files are exhausted simultaneously and no differing line has been found, the two files are identical. The program must take into account that two files may be the same in the text, but only the number of spaces or blanks on a line are different. Therefore, each line must be squeezed to eliminate blanks or spaces before they are compared. This method will give a true comparison of two files.

Collection of Questions asked in Previous Exams PU

1. Design a class student having following data members. **[Oct. 2009 – *10M*]**
   - Roll No
   - Name
   - Marks

   Write a menu driven program to do following task by using the concept of file I/O in C++
   i. Add a student record ii. Display student record iii. Modify student record
2. A company has following details of their employees in the file "emp.dat". **[Apr. 2010 – *10M*]**
   1. EmpID
   2. Emp Name
   3. Emp Address
   4. Emp Dept (Admin/Sales/Production/IT)
   5. Emp phone
   6. Emp Age

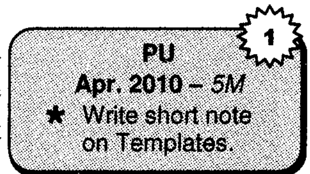
   Write a program to read the above file. Create new files such as Adm.dat, Sal.dat, Pro.dat, IT.dat respectively, to store the employee details according to their department.
3. Write a program that reads the text file and replace all the occurrences of vowels with '*' and copy the contents in new file. Also display the no. of replacements made. **[Oct. 2010 – *10M*]**
4. Explain the role of seekg(), seekp(), tellp() and tellg() functions in the process of random access in a binary file. **[Oct. 2011 – *5M*]**

VISION
Innovation Throughout

11 Template

I. Introduction

The template is one of C++'s most sophisticated and high powered features. Templates are a way of making your classes or functions more abstract by letting you define the behavior of class or function without actually knowing what datatype will be handled by the operations of a class or function.



In essence, this is what known as generic programming; this term is a useful way to think about templates because it helps the programmer to remind that a template class or function does not depend on the datatype (or types) it deals with. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data.

Templates are very useful for implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type. C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code.

C++ provides two kinds of templates: Function Templates and Class Templates.

The templates declared for functions are called as *Function Templates* and the templates those are declared for classes are called as *class templates*.

2. Generic Functions

i. Generic Functions define the general set of operations that will be applied to various data types.

ii. The specific data type the function will operate upon is passed to it as a parameter.

iii. The compiler will automatically generate the correct specific code for the type of data, i.e., actually used when the function is called.

iv. A generic function can be thought of as a function that "overloads itself".

v. A specific instance of a generic function (i.e., compiler-generated for a specific data type) is called a generated function.

vi. When you create a generic function, you are creating a function that can automatically overload itself.

Generic functions are created using the keyword **template**.

Syntax

```
template<class DataType> return_type func_name(arguments)
{
   //body
}
   or
template<typename DataType> return_type func_name(arguments)
{
   //body
}
```

Note: The only difference between both these prototypes is the use of keyword class or typename, its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way.

The Datatype is a placeholder name for a data type used by the function. The syntax defines the template of a function implementation and the compiler will automatically fill the correct data type wherever the DataType placeholder appears.

The keyword class means the parameter can be of any type. It can even be a class.

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. Hence, writing separate function for each data type will end up with 4 to 5 different functions, which can be complicated for maintenance.

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

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

This C++ function template definition will be enough. Now with the integer version of the function, the compiler generates an Add function compatible for integer data type and if float is called it generates float type and so on.

Thus a function template is used in such situation where we have to use same function for different data types.

Consider another example of function template:

Program for function template

```
#include<iostream>
using standard std;
template<class T>
T GetMax(T a, T b)
{ T result;
  result = (a > b)? a : b;
  return (result);
}
int main() {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
 }// end of main
```

Output

6
10

In the above example, we used the function **GetMax()** with arguments of type **int** and **long** having written a single implementation of the function. That is to say, we have written a function template and called it with two different patterns.

In **GetMax()** template function the type **T** can be used to declare new objects:

```
T result;
```

result is an object of type **T**, like **a** and **b**, i.e., it is to say, of the type that we enclose between angle-brackets <> when calling our template function.

In this case the generic **T** type is used as a parameter for function **GetMax** the compiler can find out automatically which data type is passed to it without having to specify it with patterns **<int>** or **<long>**. So we could have written:

```
int i,j;
GetMax(i,j);
```

since both **i** and **j** are of type **int** the compiler would assume automatically that the wished function is for type **int**. This implicit method is more usual and would produce the same result:

```
// function template II
#include<iostream>
using namespace std;
template <class T>
T GetMax(T a, T b)
{ return (a>b?a:b);   }
int main() {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Output

6
10

In this case, within function **main()** we called our template function **GetMax()** without explicitly specifying the type between angle-brackets **<>**. The compiler automatically determines what type is needed on each call.

Because our template function includes only one data type (**class T**) and both arguments it admits are both of that same type, we cannot call our template function with two objects of different types as parameters:

```
int i;
long l;
k = GetMax(i,l);
```

This would be incorrect, since our function waits for two arguments of the same type (or class).

3. A Function with Two Generic Data Types

More than one generic data type can be defined in the template statement using a comma separated list.

Syntax

```
template<class T1, class T2>
return_type  function_name(arguments of type T1,T2,. . .)
{
   //Body of function
}
```

For example

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

In this case, our template function **GetMin()** admits two parameters of different types and returns an object of the same type as the first parameter (**T**) that is passed. *For example:* After that declaration we could call the function by writing:

```
int i,j;
long l;
i = GetMin<int,long>(j,l);
```

or simply

```
i = GetMin(j,l);
```

even though **j** and **l** are of different types.

Following program demonstrates the use of a Function Template with two generic data types:

Program for a function template with two generic data types

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

template<class T, class U>
int GetMin(T a, U b){
  return(a<b?a:b);
}
int main()
{
  int i, j=10;
  long l=3444;
  i=GetMin(j,l);
  cout<<i;
}
```

Output

10

4. Explicitly Overloading a Generic Function

Although a generic function "overloads itself" you can still manually overload it explicitly.

If you overload a generic function, that overloaded function overrides (or hides) the generic function only relative to that specific datatype version.

Hence you can accommodate exception for which the general algorithm provided in the generic function needs to do something slightly different.

Program for explicitly overloading a template function

```
#include<iostream>
using namespace std;
template<class p> void swapargs(p &a, p &b)
{
  p temp;
  temp=a;
  a=b;
  b=temp;
  cout<< "Inside template swap argument \n";
}
// the following function overrides the general version of swapargs()
//for integers
void swapargs(int &a, int &b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
  cout<< "\n Inside swapargs integer specialization";
}
int main()
{
  int i=30, j=40;
  double p=20.5, q=30.7;
  char a= 'x', b= 'z';
  cout << "Original i,j:" << i << ' '<< j << "\n";
  cout << "Original p,q:" << p << ' '<< q << "\n";
  cout << "Original a,b:" << a << ' '<< b << "\n";
  swapargs(i,j);   //calls explicitly overloaded swapargs()
  swapargs(p,q);   //calls generic swapargs()
  swapargs(a,b);   //calls generic swapargs()
  cout<< "Swapped i,j:" << i << ' ' << j << "\n";
  cout<< "Swapped p,q:" << p << ' ' << q << "\n";
  cout<< "Swapped a,b:" << a << ' ' << b << "\n";
  return 0;
}
```

Output

```
Original i,j : 30     40
Original p,q : 20.5 30.7
Original a,b :  x      z
Inside swapargs integer specialization
Inside template swap argument
Inside template swap argument
Swapped i,j :  40        30
Swapped p,q   :  30.7      20.5
Swapped a,b:   z          x
```

When swapargs(i,j) is called, it invokes the explicitly overloaded version of swapargs() defined in the program. Thus, the compiler does not generate this version of the generic swapargs() function, because the generic function is overridden by the explicit overloading.

Recently, a new-style syntax was introduced to denote the explicit specialization of a function. In this method template keyword is used.

For example: The overloaded swapargs() function from the above program will look like as follows:

```
// By using the template keyword
template <> void swapargs<int> (int &a, int &b)
{
   int temp;
   temp=a;
   a=b;
   b=temp;
   cout<< "Inside swapargs int specialization\n";
}
```

In the above code, the template<> construct is used to indicate specialization. The type of data for which the specialization is being created is placed inside the angle brackets following the function name. This same syntax is used to specialize any type of generic function. The new-style approach is better for a long term.

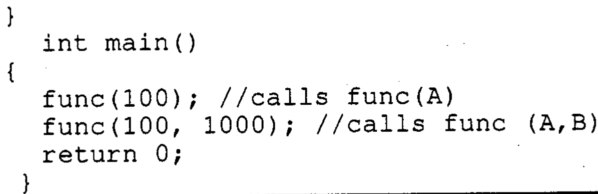
Explicit specialization of a template allows you to tailor a version of a generic function to accommodate a unique situation perhaps to take advantage of some performance boost that applies to only one type of data. However, as a general rule, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.

5. Overloading Function Templates

Sometimes, a function template just can't handle all of the possible instantiations that you might want to do. Besides creating explicit, overloaded versions of a generic function, the template specification itself can be overloaded. This is done by simply creating another version of the template that differs from any other version in its parameter list.

Example

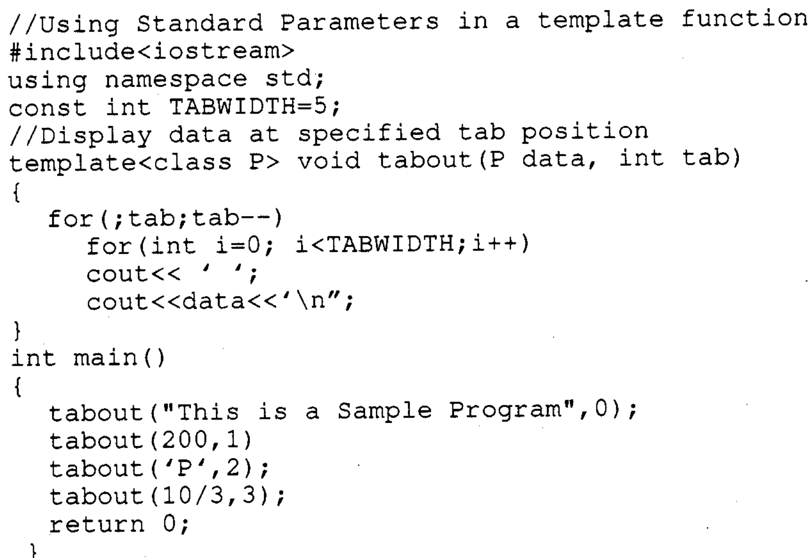
```
#include<iostream>
using namespace std;
//First version of func() template
template<classA> void func(A a)
{ cout <<"Inside func(A a) \n";
}
// Second version of func() template
template<class A, class B> void func(A a, B b)
{
  cout <<"Inside func(A a, B b)\n";
```



```
}
  int main()
{
  func(100); //calls func(A)
  func(100, 1000); //calls func (A,B)
  return 0;
 }
```

6. Using Standard Parameters with Template Functions

The standard parameters can be mixed with generic type parameters in a template function. These non-generic parameters work just like they do with any other function. *For example*



```
//Using Standard Parameters in a template function
#include<iostream>
using namespace std;
const int TABWIDTH=5;
//Display data at specified tab position
template<class P> void tabout(P data, int tab)
{
   for(;tab;tab--)
      for(int i=0; i<TABWIDTH;i++)
      cout<< ' ';
      cout<<data<<'\n";
}
int main()
{
   tabout("This is a Sample Program",0);
   tabout(200,1)
   tabout('P',2);
   tabout(10/3,3);
   return 0;
 }
```

Output

```
This is a Sample Program
     200
          P
               3
```

In the above program, the function tabout( ) displays its first argument at the tab position requested by its second argument. The first argument is a generic type, tabout( ) can be used to display any type of data. The tab parameter is a standard, call-by-value parameter. The mixing of generic and non-generic parameters causes no trouble and is indeed, both common and useful.

Advantage of C++ Function Templates

C++ function templates can be used wherever the same functionality has to be performed with a number of data types. Though very useful, lots of care should be taken to test the C++ template functions during development. A well-written C++ template will go a long way in saving time for programmers.

7. Generic Functions Restrictions

Generic functions are similar to overloaded functions except that they are more restrictive.

- All generated versions of a generic function need to perform the exact same action only the data type may differ.
- If you want to have different actions performed in different versions of a function, use overloading instead.

Applying Generic Function: Generic sort

We know that, one of C++'s most useful feature is Generic Functions. Generic Functions can be applied to all types of situations. Whenever you have a function that defines a generalizable algorithm, you can make it into a template function. Once you have done so, you may use it with any type of data without having to record it. A Generic Sort is an example which illustrates how easy it is to take advantage of this powerful C++ feature.

▶ Generic Sort

Sorting of any type of data is exactly the type of operation for which generic functions were designed. Let us consider an example of creating generic bubble sort. One of the characteristics of this sort is that it is easy to understand and program. But as compared to other sorting techniques it is less efficient.

The basic idea underlying the bubble sort is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (x[i] with x[i+1]) and interchanging the two elements if they are not in proper order.

The bubble() function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

Program for a generic bubble sort

```
#include<iostream>
using namespace std;
template<class T> void bubble
( T *items,    // pointer to array to be sorted
  int count)   // number of items in array
{
```

```
   register int a,b;
   T t;
   for(a=1;a<count;a++)
      for(b=count-1;b>=a;b--)
         if(items[b-1]>items[b]){
            //exchange elements
            t=items[b-1];
            items[b-1]= items[b];
            items[b]=t;
         } }
int main()
{
   int iarray[8]={25  57  48  37  12  92  86  33}
double darray[8]={7.5  10.5  4.7  9.8  3.0  100.2  2.5  -0.9}
   int i;
   cout<<"The Unsorted integer array is:";
   for(i=0;i<8;i++)
   cout<<iarray[i]<< ' ';
   cout<<endl;
   cout<<"The Unsorted double array is:";
   for(i=0;i<8;i++)
   cout<<darray[i]<< ' ';
   cout<<endl;
   bubble(iarray,8);
   bubble(darray,8);
   cout<< "The sorted integer array is:";
   for(i=0;i<8;i++)
   cout<<iarray[i]<< ' ';
   cout<<endl;
   cout<< "The sorted double array is:";
   for(i=0;i<8;i++)
   cout<<darray[i]<< ' ';
   cout<<endl;
   return 0;
}
```

Output

The Unsorted integer array is: 25 57 48 37 12 92 86 33

The Unsorted double array is:7.5 10.5 4.7 9.8 3.0 100.2 2.5 –0.9

The sorted integer array is: 12 25 33 37 48 57 8692

The sorted double array is:–0.9 2.53 4.7 7.5 9.8 10.5 100.2

In the above program, we create two arrays one is integer and another is double. You may also try to sort other types of data, including classes. The bubble() function sorts each. Since it is a template function and is automatically overloaded to accommodate the two different types of data. But in each case, the compiler will create the right version of the function for you.

8. Generic Classes

In addition to generic functions, you can also define a generic class. The created class defines all the algorithms used by that class; however the actual type of data being manipulated will be specified as a parameter when an object of that class is created.

Generic classes are useful when a class uses logic that can be generalized. A generic class can perform the defined operation like maintaining a queue, on a linked list, for any type of data.

Generic Classes are declared similar to generic functions.

The **syntax** for Generic Classes is as follows:

```
template<class DataType> class class_name{
     //class declaration
}
```

Declaration of the C++ class template should start with the keyword template. The parameter should be included inside angular brackets. The parameter inside the angular brackets can be either the keyword class or typename. After that the DataType is a placeholder where the compiler will automatically fill the correct datatype. This is followed by the class body declaration with the member data and member functions.

Once you define a generic class, you create a specific instance of that class using the following general form:

```
class_name<type> Object_name;
```

Here, type is the type name of the data that the class will be operating upon.

All member functions of a generic class are automatically generic. No need to use the template keyword while declaring them.

If needed, more than one generic data type can be declared using a comma-separated list *(for more detail refer 9.9).*

The desired data type is specified in angular brackets when an object of a generic class is instantiated.

The compiler automatically creates the data type specific versions of all member functions and variables for you.

For example: The following is the declaration for a sample Queue class.

```
template<class T>class stack{
T st[size]; // holds the stack
int top; // index of top of stack
public:
stack(){t = 0;} //initialize
void push (T obj); // push object on stack
T pop(); //pop object from stack
};
```

Defining Member Functions

If the functions are defined outside the template class body, they should always be defined with the full template definition. Other conventions of writing the function in C++ class templates are the same as writing normal C++ functions.

```
//Push object on stack
template<class T> void stack<T>::push(T obj)
{
   if(top==size)
   {
      cout<<"Stack complete.\n";
      return;
   }
st[top] = ob;
top++;
}
//Pop object from stack
template<class T> T stack<T>::pop()
{
   if(top==0)
   {
      cout<<"Empty stack.\n";
      return 0;
   }
top--;
return st[top];
}
int main()
{
   stack<char>S1, S2; //character stacks
   int i;
   S1.push('a');
   S2.push('x');
   S1.push('b');
   S2.push('y');
   S1.push('c');
   S2.push('z');
   for(i = 0; i<3; i++) cout<<"Pop S1:"<<S1.pop()<<"\n";
   for(i = 0; i<3; i++) cout<<"Pop S2:"<<S2.pop()<<"\n";
   stack<int> is1, is2; //integer stacks
   is1.push(1);
   is2.push(5);
   is1.push(2);
   is2.push(6);
   is1.push(3);
   is2.push(7);
   for(i=0; i<3; i++) cout<<"Pop S1:"<<S1.pop()<<"\n";
   for(i=0; i<3; i++) cout<<"Pop S2:"<<S2.pop()<<"\n";
   return 0;
}
```

Consider another example,

```
template<class T>
class pair {
    T values[2];
  public:
    pair(T first, T second)
    {  values[0]=first; values[1]=second;
     }
};
```

The class that we have just defined serves to store two elements of any valid type. *For example*: If we wanted to declare an object of this class to store two integer values of type **int** with the values **115** and **36** we would write:

```
pair<int> myobject(115, 36);
```

this same class would also serve to create an object to store any other type:

```
pair<float> myfloats(3.0, 2.18);
```

The only member function has been defined *inline* within the class declaration. If we define a function member outside the declaration we must always precede the definition with the prefix **template <... >**.

Program for class templates

```
#include<iostream>
using namespace std;
template<class T>
class pair{
    T value1, value2;
  public:
     pair(T first, T second)
       {value1=first; value2=second;}
     T getmax(); };
template<class T>
T pair<T>::getmax()
{  T retval;
retval = (value1>value2? value1:value2);
   return retval;
}
int main() {
  pair <int> myobject(100, 75);
  cout << myobject.getmax();
  return 0;
 }
```

Output

100

Note, how the definition of member function **getmax** begins:

```
template <class T>
T pair<T>::getmax()
```

All **T**s that appear are necessary because whenever you declare member functions you have to follow a format similar to this (the second **T** makes reference to the type returned by the function, so this may vary).

PU
Oct. 2009 – 10M
* What is the advantage of using a template class? Write a class template VECTOR with a series of float values and perform the following operations:
  i. To create a vector
  ii. To add two vectors

Advantages of C++ Class Templates

i. One C++ Class Template can handle different types of parameters.

ii. Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the C++ template class.

iii. Templates reduce the effort on coding for different data types to a single set of code.

iv. Testing and debugging efforts are reduced.

9. An example with two generic data types

A class template can have more than one generic data type. Just declare all the data types required by the class in a comma-separated list within the template specification.

The **syntax** is as follows:

```
template<class T1, class T2, . . .>
class classname
{
  //Body of class
};
```

The following program demonstrates the use of template class with two generic data types.

Program demonstrating the use of class template with two generic data types

```
#include<iostream>
using namespace std;
template<class T, class U>
class Test
{
  T a;
  U b;
  public:
    Test(T x, U y)
    { a=x;
      b=y;
    }
    void display()
    {
      cout<<a<< "and" <<b<< "\n";
    }
  }
```

```
};
int main()
{
  Test<int,char> obj1(10, 'X');
  Test<double,char*> obj2(0.25, "Templates add power");
  obj1.display();    //display int,char
  obj2.display();    //display double,char*
  return 0;
}
```

Output

10 X

0.25 Templates add power

Two types of objects are used in the program, i.e., obj1 uses int and char data and obj2 uses a double and a character pointer. For both cases, the compiler automatically generates the appropriate data and functions to accommodate the way the objects are created.

10. Using Non-type Arguments with Generic Class

In the template specification for a generic class you may also specify non-type arguments. That is besides the template arguments that are preceded by the class or typename keywords which represents type argument, we can also use other arguments such as an integer, enumeration, pointer, reference, or pointer to member type. Non-type template argument/parameter must be constant at compile time. Non-type template parameters can be qualified as const or volatile types. Floating point values, string literals, Objects of class, struct or union type are not allowed as non-type template parameters, although pointers to such objects are allowed. Arrays passed as non-type template parameters are converted into pointers. Functions passed as non-type parameters are treated as function pointers.

As an example consider a template that is used to contain sequences of elements.

Program for illustrating the non-type template parameters

```
// sequence template
#include<iostream>
using namespace std;

// Here, int N is a non-type argument
template<class T, int N>
class sequence {
    T memblock[N];
  public:
    void setmember(int x, T value);
    T getmember(int x);
};
template <class T, int N>
void sequence<T,N>::setmember(int x, T value) {
```

```
  memblock[x]=value;
}

template <class T, int N>
T sequence<T,N>::getmember(int x) {
  return memblock[x];
}
int main() {
  sequence<int,5> myints;
  sequence<double,5> myfloats;
  myints.setmember(0,100);
  myfloats.setmember(3,3.1416);
  cout << myints.getmember(0) << "\n";
  cout << myfloats.getmember(3) << "\n";
  return 0;
}
```

Output

100
3.1416

11. Using Default Arguments With Template classes

You can provide default arguments for template parameters in class templates but not in function templates.

A *default template-argument* is a *template-argument* specified with the equal (=) sign followed by the type name or value in a *template-parameter*.

For example

```
template <class T = long> class A;
```

Here, the type *long* will be used if no other data type is specified when an object of type A is instantiated.

As with default function arguments they should only be defined once the first time a class template declaration or definition is seen by the compiler and may be specified for any kind of *template-parameter* (type, non-type, template). They shall not be specified in a function template declaration or a function template definition, nor in the *template-parameter-list* of the definition of a member of a class template.

Once you introduce a default argument all the subsequent template parameters must also have defaults.

You can choose to provide defaults for all arguments but you must use an empty set of brackets when declaring an instance so that the compiler knows that a class template is involved:

```
template<class T = int size_t N = 100>  // Both default
class Stack {
  T data[N];  // Fixed capacity is N
  size_t count;
public:
  void push(const T& t);
  // Etc.
```

```
};
Stack<> myStack;   // Same as Stack<int 100>
```

When declaring a template class object with default arguments, omit the arguments to accept the default argument. If there are no non-default arguments, do not omit the empty angle brackets. A template that is multiply declared cannot specify a default argument more than once. The following code demonstrates an error: *For example*

```
template<class T = long> class A;
template<class T = long> class A { /* . . . */ }; // Generates Error
```

In the following example, an array class template is defined with a default type **int** for the array element and a default value for the template parameter specifying the size.

Program for default template arguments

```
#include<iostream>
using namespace std;
template<class T = int, int size = 10> class Array
{  T* array;
public:
   Array()
   {  array = new T[size];
      memset(array,0, size * sizeof(T));  }
   T& operator[](int i)
   {   return *(array + i);  }
   const int Length() { return size; }
   void print()
   { for(int i = 0; i < size; i++) {cout << (*this)[i] << " ";  }
      cout << endl; } };
int main()
{  // Explicitly specify the template arguments:
   Array<char, 26> ac;
   for(int i = 0; i < ac.Length(); i++)
   {  ac[i] = 'A' + i; }
   ac.print();
   // Accept the default template arguments:
   Array<> a; // You must include the angle brackets.
   for(int i = 0; i < a.Length(); i++)
   { a[i] = i*10; }
   a.print();
 }
```

Output

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 10 20 30 40 50 60 70 80 90

The use of default arguments especially default types adds versatility to your template classes. You can provide a default for the type of data most commonly used while still allowing the user of your classes to specialize them as needed.

12. Template Parameters

The third type of parameter a template can accept is another class template. If you are going to use a template type parameter itself as a template in your code the compiler needs to know that the parameter is a template in the first place. The following example illustrates a template parameter:

Program for a template parameter

```
#include<iostream>
using namespace std;
template<class T>
class Array { // A simple expandable sequence
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  Array() {
    count = 0;
    data = new T[capacity = INIT];
  }
  ~Array() { delete [] data; }
  void push_back(const T& t) {
    if(count == capacity) {
      // Grow underlying array
      size_t newCap = 2 * capacity;
      T* newData = new T[newCap];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCap;
    }
    data[count++] = t;
  }
  void pop_back() {
    if(count > 0)
      --count;
  }
  T* begin() { return data; }
  T* end() { return data + count; }
};
template<class T template<class> class Seq>
class container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};
int main() {
   container<int Array> container;
   container.append(1);
   container.append(2);
```

```
  int* p = container.begin();
  while(p != container.end())
    cout << *p++ << endl;
}
```

The **Array** class template is a trivial sequence container. The **Container** template takes two parameters: the type that it holds and a sequence data structure to do the holding. The following line in the implementation of the **Container** class requires that we inform the compiler that **Seq** is a template:

```
Seq<T> seq;
```

If we hadn't declared **Seq** to be a template parameter the compiler would complain here that **Seq** is not a template since we're using it as such. In **main( )**, a **container** is instantiated to use an **Array** to hold integers so **Seq** stands for **Array** in this example.

Note that, it is not necessary in this case to name the parameter for **Seq** inside **Container**'s declaration. The line in question is:

```
template<class T template<class> class Seq>
```

Although we could have written

```
template<class T template<class U> class Seq>
```

the parameter **U** is not needed anywhere. All that matters is that **Seq** is a class template that takes a single type parameter. This is analogous to omitting the names of function parameters when they're not needed such as when you overload the post-increment operator:

```
T operator++(int);
```

The **int** here is merely a placeholder and so needs no name.

13. Template Specializations

The term *specialization* has a specific template-related meaning in C++. A template definition is by its very nature a *generalization* because it describes a family of functions or classes in general terms. When template arguments are supplied the result is a specialization of the template because it determines a unique instance out of the many possible instances of the family of functions or classes. Consider the following definition of **min( )** function template it is a generalization of a minimum-finding function because the type of its parameters is not specified.

```
template<typename T> const T& min(const T& a const T& b) {
  return ((a < b) ? a : b);}
```

When you supply the type for the template parameter whether explicitly or implicitly via argument deduction the resultant code generated by the compiler (*for example*: **min<int>( )**) is a specialization of the template. The code generated is also considered an *instantiation* of the template as are all code bodies generated by the template facility.

Explicit Specialization

An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by `template<>`.

Program illustrating the specialization of a class template

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

template<class T> class myclass{
T a;
public:
myclass(T x) {
  cout << "Inside generic myclass\n";
  a=x;
}
T geta()
{return a;}};
// Explicit specialization for double.
template <> class myclass<double> {
double a;
public:
  myclass(double x){
  cout<<"Inside myclass<double> specialization\n";
  a=x*x;
}
int geta()
{ return a;}
};

int main()
{
  myclass<int> i(20);
  cout<< "Integer:" <<i.geta() << "\n\n";
  myclass<double>i(5.0);
  cout<<"Double:"<<d.geta()<<"\n";
  return 0;
 }
```

Output

Inside generic myclass
Integer:20
Inside myclass<double> specialization
Double: 25.0

The "**template<>**" prefix tells the compiler that what follows is a specialization of a template. The type for the specialization must appear in angle brackets immediately following the function name as it normally would in an explicitly specified call.

In the program, the line

```
template<> class myclass<double> {
```

tells the compiler that an explicit double specialization of *myclass* is being created. This same syntax is used for any type of class specialization. Explicit class specialization expands the utility of generic classes because it lets you easily handle one or two special cases while allowing all others to be automatically processed by the compiler. If you find that you are creating too many specializations, you are probably better off not using a template class in the first place. Explicit specializations tend to be more useful for class templates than for function templates. When you provide a full specialization for a class template though you may need to implement all the member functions. This is because you are providing a separate class and client code may expect the complete interface to be implemented.

14. The Typename and Export Keywords

The two keywords typename and export are added to C++ recently. Both are related specifically to templates and plays specialized roles in C++ programming. The typename and export keywords are briefly explained below.

Typename Keyword

As we already know that we can use the typename keyword in place of the keyword *class* in template parameter declaration.

For example: `template<class T> class xyz{};`

could have been written as:

```
template<typename T> class xyz{};
```

these two definitions of xyz are considered equivalent since template parameters using class or typename are interchangeable. Although the typename and class are synonymous at the start of the template definition, but sometimes you need the typename to tell the compiler that a symbol in a template represents a type name and not an object. *For example:* Consider the template shown below:

```
template<class T> class test{
  void X(){
    T::Y*P;
    . . .
        }
}
```

What does the T::Y*P mean in this code sample? Is this is a multiplication expression or is it supposed to be a pointer declaration? Is T::Y a member or a type? You can solve these types of questions by using typename.

```
template<class T> class test{
  void X(){ typename T::Y*P;
    . . . } }
  The format is: typename T::Name someobject;
```

Now the compiler knows T::Y is a type, not a member name. The export keyword can precede a template declaration. It allows other files to use a template declared in a different file by specifying only its declaration rather than duplicating its entire definitions.

Solved Programs

1. What is the advantage of using a template class? Write a class template VECTOR with a series of float values and perform the following operations:

i. To create a vector ii. To add two vectors

PU
Oct. 2009 – *10M*

Solution

```
#include<iostream.h>
#include<conio.h>
const size = 3;
Template <class T>
class vector
{
   T* v;
   public:
   vector()
   {
      v = new T[size];
      for(int i = 0;i<size;i++)
         v[i] =0;
   }
   vector(T* a)
   {
for(int i = 0;i<size;i++)
      v[i] = a[i];
   }
   T operator *(vector &y)
   {
      T sum = 0;
      for(int i =0;i<size; i++)
         sum += this-> v[i] * y.v[i];
      return sum;
   }
};
void main()
{
   float x[3] ={1.1,2.2,3.3};
   float y[3] = {4.4,5.5,6.6};
   clrscr();
   vector <float> v1;
   vector <float> v2;
   v1 = x;
   v2 = y;
   float r= v1* v2;
   cout << "R= " << r << "\n";
   getch();
}
```

2. Write a function template to calculate area of circle.

Solution

PU
Oct. 2010 – *5M*

/* Function template to calculate area of circle */

```
#include<iostream.h>
template <class T>//template declaration
void areaOfCircle(T radius)//function to calculate area for integer and
//radius value
{
  cout<<"Area of circle="<<3.142 *radius*radius;
}
int main()
{
  int intRadius;
  cout<<"Enter radius value:";
  cin >>intRadius;
  areaOfCircle(intRadius);//calculate area of circle for integer value
                          //of radius
  float floatRadius;
  cout<<"Enter radius value:";
  cin >>floatRadius;
  areaOfCircle(floatRadius);//calculates area of circle for float value
                            //of radius
  return 0;
}
```

3. Write a template function for sort. Also write a person class which having name and age as data members. Write a main function in which create an integer array and person object array. Store n values in both arrays and call the sort function to sort integer array and also the person object array. The person object array sorted by according to person name.

Solution

PU
Oct. 2011 – *10M*

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class Person
{
  private:
    char name[20];
    int age;
  public:
    void readData()
    {
      cout<<"\n Enter name : ";
      cin>>name;
      cout<<"\n Enter age : ";
      cin>>age;
    }
    void display()
    {
      cout<<"\nName : "<<name;
```

```
        cout<<"\nAge : "<<age;
      }
      int operator>(Person p)
      {
        int flag;
        //cout<<"\n\nOperator:"<<name<<"\n"<<p.name;
        flag = stricmp(name,p.name);
        if(flag > 0)
        {
          return 1;
        }
        else
        {
          return 0;
        }
      }
      Person operator=(Person &obj)
      {
        strcpy(name,obj.name);
        age = obj.age;
        return *this;
      }
};
template <class T>
void sort(T x[])
{
  int i,j;
  T temp;
  for(i=0;i<5;i++)
  {
    for(j=i+1;j<5;j++)
    {
      if(x[i] > x[j])
      {
        temp = x[i];
        x[i] = x[j];
        x[j] = temp;
      }
    }
  }
}
void main()
{
  Person obj[5];
  int arr[5];
  int i;
  cout<<"\nEnter 5 numbers : \n";
  for(i = 0; i < 5; i++)
  {
    cin>>arr[i];
  }
  cout<<"\n Before sorting, array elements are : \n";
  for(i = 0; i < 5; i++)
  {
```

```
    cout<<arr[i]<<"\n";
  }
  sort(arr);
  cout<<"\n After sorting, array elements are : \n";
  for(i = 0; i < 5; i++)
  {
    cout<<arr[i]<<"\n";
  }
  cout<<"\nEnter name and age of 5 persons : ";
  for(i = 0; i < 5; i++)
  {
    obj[i].readData();
  }
  cout<<"\n Before sorting, person data is : \n";
  for(i = 0; i < 5; i++)
  {
    obj[i].display();
  }
  sort(obj);
  cout<<"\n After sorting, person data is : \n";
  for(i = 0; i < 5; i++)
  {
    obj[i].display();
  }
  getch();
}
```

EXERCISES

A. Review Questions

1. What is a template? List the merits and demerits of using a template in C++.
2. Define a function template.
3. Explain how a function template is defined and declared in a program.
4. What is a class template?

B. Programming Exercise

1. Write a program in C++ to perform the following using the function template concepts:
   a. to read a set of integers
   b. to read a set of floating point numbers
   c. to read a set of double members individually.

   Find out the average of the nonnegative integers and also calculate the deviation of the numbers.
2. Write a C++ program using a class template to read any five parameterized data type such as float and integer and print the average.

3. Identify which of the following function template definitions are illegal.

i.
```
template<class A,B>
void fun(A, B)
{. . .};
```

ii.
```
template<class A, class A>
void fun(A, A)
{. . .};
```

iii.
```
template<class A>
void fun(A, A)
{. . .};
```

iv.
```
template<class T, typename R>
T fun(T, R)
{. . .};
```

v.
```
template<class A>
A fun(int *A)
{. . .};
```

4. Find error if any, in the following code segment.

```
Template <class T>
T max(T, T)
{ . . .};
unsigned int m;
int main()
{ max(m, 100); }
```

Collection of Questions asked in Previous Exams PU

1. What is the advantage of using a template class? Write a class template VECTOR with a series of float values and perform the following operations: **[Oct. 2009 – *10M*]**
   i. To create a vector ii. To add two vectors
2. Write short note on Templates. **[Apr. 2010 – *5M*]**
3. Write a function template to calculate area of circle. **[Oct. 2010 – *5M*]**
4. Write a template function for sort. Also write a person class which having name and age as data members. Write a main function in which create an integer array and person object array. Store n values in both arrays and call the sort function to sort integer array and also the person object array.
   The person object array sorted by according to person name. **[Oct. 2011 – *10M*]**

12 Exception Handling

1. Introduction

We know that it is very rare that a program will work correctly first time. It might have some errors. An error typically refers to a problem that exists in a program when it is written and compiled. It could be a logic error or syntax error. Logic error will result in incorrect results. In this case, the code is correct but the algorithm has an error. This type of error will only be found during program testing or during design reviews of the program. Another type of error is a syntax error. Typically, the compiler finds errors of this type and they are corrected during the coding of the program.

We often come across some peculiar problems other than the logic or syntax error. They are known as exceptions. Exceptions are errors or run-time anomalies that occur during the execution of a program.

They can be the result of unavailability of system resources, such as memory, file space or from data dependent conditions such as division by zero, or numerical overflow or access to an array outside of its bounds. Such exceptions exist outside the normal functioning of the program and require immediate handling by the program. ANSI C++ provides built-in language features to raise and handle exceptions. These language features activate a run-time mechanism used to communicate exceptions between two unrelated (often separately developed) portions of a C++ program.

One of the most powerful features of exception handling is that an error can be thrown over function boundaries.

This means that if one of the deepest functions on the stack has an error, this error can propagate up to the upper function if there is a *trying*-block of code there. This allows programmers to put the error handling code in one place, such as the *main*-function of your program.

Exception handling is a new feature added to ANSI C++ and it was not part of the original C++. Today, almost all compilers support this feature. C++ exception handling provides a type-safe, integrated approach, for coping with the unusual predictable problems that arise while executing a program.

1.1 Why Exception Handling?

Exceptions are anomalies that occur during the normal flow of a program and prevent it from continuing. These anomalies--user, logic, or system errors--can be detected by a function. If the detecting function cannot deal with the anomaly, it throws, an exception. A function that handles that kind of exception catches it.

In C++, when an exception is thrown, it cannot be ignored--there must be some kind of notification or termination of the program. If no user-provided exception handler is present, the compiler provides a default mechanism to terminate the program.

1.2 Synchronous and Asynchronous Exceptions

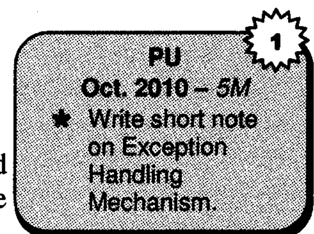
Exceptions are of two kinds, namely synchronous exception and asynchronous exception.

i. **Synchronous Exception Handling:** Exception handling is designed to support only synchronous exceptions, such as array range checks. The term *synchronous exception* means that exceptions can only be originated from throw expressions.

   The C++ standard supports synchronous exception handling with termination model. Termination means that once an exception is thrown, control never returns to the throw point.

ii. **Asynchronous Exception Handling:** These are caused by event that are beyond the control of the program. Exception handling is not designed to directly handle asynchronous exceptions such as keyboard interrupts. However, exception handling can be made to work in the presence of asynchronous events if care is taken. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, have another routine that polls the value of that variable at regular intervals, and throws an exception when the value changes.

2. Exception Mechanism



Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

So far we have handled error conditions by using the if statement to test some expressions and then executing specific code to deal with the error. C++ Language provides a good mechanism to tackle these conditions.

The exception mechanism uses three keywords, i.e., try, throw and catch.

The exception handling is done through the following steps:

i. Hit the exception. In other words, find the problem.

ii. Throw the exception. It suggests, report the error.

iii. Catch the exception. It means, receive the error information.

iv. Handle the exception. Lastly, perform corrective measures against the problem.

2.1 The try Block

A try block is a group of C++ statements, normally enclosed in braces { }, which may cause an exception (i.e., an error). The general form of the try is as follows:

```
try
{
  // statements in which exceptions may be thrown
}
```

When an exceptional circumstance arises within try block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored. An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block.

A try block can contain any C++ statement such as expressions as well as declarations. A try block introduces a local scope, and variables declared within a try block cannot be referred to outside the try block, including within the catch clauses.

2.2 The throw Statement

The throw statement is used to throw an exception inside of a try block to a subsequent exception handler. A throw statement is specified with:

- the keyword throw
- an assignment expression;

The throw statement is one of the following form:

```
throw(exception);
throw exception;
throw;          // used to rethrow an exception
```

where exception is object used to transmit information about a problem. The object being thrown can be Specific object or Anonymous object.

For example

Throw expression	Effect
throw 4;	The constant value 4 is thrown.
throw num;	The value of the variable num is thrown.
throw str;	The object str is thrown.
throw string("Exception found!");	An anonymous string object with the string "Exception found!" is thrown.

The type of the exception is used to determine which catch block to execute. The exception is passed as an argument to the catch block so that it can be used in handling the exception.

2.3 The catch Exception Handler

A catch block is a group of C++ statements that are used to handle a specific raised exception. Catch blocks, or handlers, should be placed after each try block. A catch block is specified by:

- The keyword catch
- A catch expression, which corresponds to a specific type of exception that may be thrown by the try block
- A group of statements, enclosed in braces { }, whose purpose is to handle the exception

For Example: A catch-block receiving char * exceptions:

```
catch(char *message)
  {
    //statements in which the thrown char * exceptions are handled
  }
```

The general form of the try and catch block are shown below:

```
try {
    ...
    ...
    throw Exception;   //Group of statements which detects and throws
                       // and exception
    ...
    ...
} catch(type arg)      // Catches exception
   {
    ...                // Group of statements that handles the
                       exception
    ...
   }
```

▶ Implementing Exception Handlers

Following steps are involved in implementing exception handlers:

i. When a function is called by many other functions, you should code it so an exception is thrown whenever an error is detected. The throw expression throws an object. This object is used to identify the types of exceptions and to pass specific information about the exception that has been thrown.

ii. Use the try statement in a client program to anticipate exceptions. Precede function calls that you anticipate may produce an exception with the keyword try and enclose the calls in braces.

iii. Code one or more catch blocks immediately after the try block. Each catch block identifies what type or class of objects it is capable of catching. When an object is thrown by the exception, this is what takes place:

- If the type of the object thrown by the exception matches the type of catch expression, control passes to that catch block.
- If the type of object thrown by the exception does not match the first catch block, subsequent catch blocks are searched for a matching type.
- If try blocks are nested, and there is no match, control passes from the innermost catch block to the outermost catch block.
- If there is no match in any of the catch blocks, the program is normally terminated with a call to the predefined function terminate(). By default, terminate() calls abort(), which destroys all remaining objects and exits from the program. This default behavior can be changed by calling the set_terminate() function.

iv. If no exception is thrown (that is no error occurs within the try block) during execution of the statements inside the try block, the **catch** clauses that follow the **try** block are not executed and control goes to the statement immediately after the catch block.

Figure 12.1 shows the execution of try and catch.

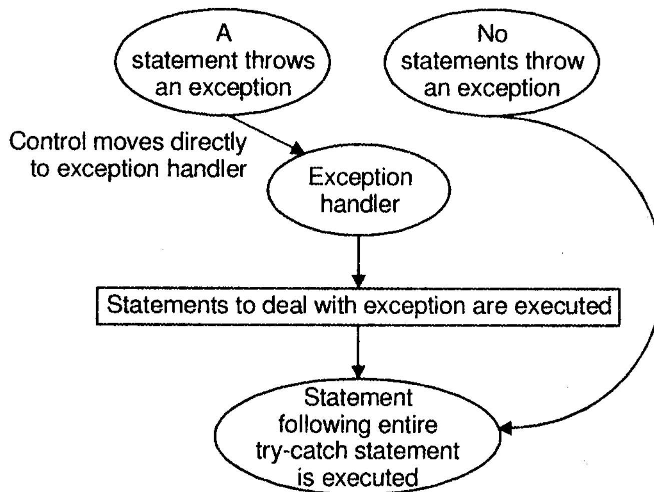


Figure 12.1

Following is a simple example of a **try** block and its associated **catch** handler.

Program for try-catch block

```
// exceptions
#include<iostream.h>
int main()
{
  char myarray[10];
```

```
  try
  {
    for(int n=0; n<=10; n++)
    {
      if(n>9) throw "Out of range";
      myarray[n]='z';
    }
  }
  catch(char * str)
  {
    cout << "Exception: " << str << endl;
  }
  return 0;
}
```

Output

Exception: Out of range

In this example, if within the **n** loop, **n** gets to be more than 9 an exception is thrown, since **myarray[n]** would be in that case point to a non-trustworthy memory address. When **throw** is executed, the **try** block finalizes right away and every object created within the **try** block is destroyed. After that, the control is passed to the corresponding **catch** block (that is only executed in these cases). Finally the program continues right after the **catch** block, in this case **return 0;**.

The syntax used by **throw** is similar to that of **return.** Only the parameter does not need to be enclosed between parenthesis.

The **catch** block must go right after the **try** block without including any code line between them. The parameter that **catch** accepts can be of any valid type. Even more, **catch** can be overloaded so that it can accept different types as parameters. In that case the **catch** block executed is the one that matches the type of the exception sent (the parameter of **throw**).

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block.

The point at which the throw is executed is called as the throw point. Once an exception is thrown to the catch block control cannot return to the throw point. Following program illustrates how a try block invokes a function that generates an exception.

```
// Throwing an exception from a function outside the try block
#include<iostream>
using namespace std;
void Xtest(int test)
{ // function
  cout << "Inside Xtest:" << test << endl;
  if(test) throw test;
}
main()
{
  cout << "Start\n";
  try { // start a try block
  Xtest(0); // call function inside try block
  Xtest(1);
  Xtest(2);
```

```
}
catch(int i)
{ // catch block for int exceptions
  cout << "Caught exception" << i << "\n";
}
cout << "End\n";
return 0;
 }
```

Output

Start
Inside Xtest: 0
Inside Xtest: 1
Caught exception 1
End

The try block can be localized to the function. In this case, each time the function is entered, the exception handling relative to that function is reset. *For example*: Consider the following program:

```
// Try-catch can be localized to the function.
#include<iostream>
using namespace std;
void Xtest(int test)
{ // function
  cout << "Inside Xtest:" << test << endl;
  try
  { // try block is now inside the function
    if(test) throw test;
  }
  catch(int i) { // then also catch needs to be here
    cout << "Caught exception" << i << "\n";
  }
}
main()
{
  cout << "Start\n";
  Xtest(0);     // simply call the function
  Xtest(1);
  Xtest(2);
  cout << "End\n";
  return 0;
 }
```

Output

Start
Inside Xtest: 0
Inside Xtest: 1
Caught exception 1
Inside Xtest: 2

Caught exception 2
End

Consider the output of the above two programs, i.e., output of the program when function within try block and output of the program when try block within function: the difference between them is that when the try block is within the function after each exception, the function returns and when the function is called again, the exception handling is reset.

▶ Catching Class Types

An exception can be of any type, including class types that you create. In real world programs, most exceptions will be class types rather than built-in types.

The most common reason for defining a class type for an exception is to create an object that describes the error that occurred.

Exception handler can use this information to process the error. The following example illustrates this:

Program for using class type exception

```
#include<iostream>
#include<cstring>
using namespace std;
class test{
public:
char str[30];
int a;
test(){*str=0;a=0;}
test(char *s, int e)
{
  strcpy(str,s);
  a=e;
}
};
int main()
{
  int i;
  try{
      cout<< "Enter a positive number:";
      cin>>i;
      if(i<0)
        throw test("Entered number is not positive",i);
      }
  catch(test e) {    //catch an error
      cout<< e.str << ":";
      cout<<e.a<<"\n";
      }
      return 0;
 }
```

Output

Enter a positive number: -2
Entered number is not positive: -2

In the above program, we have to enter a positive number. If we enter a negative number, an object of the class test is created that describes the error. Thus test encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

3. Using Multiple Catch Statements

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

The general form of multiple catch statement is as follows:

```
try
{
     //C++ statements
}
catch(type1 arg)
{
     //catch block1
}
catch(type2 arg)
{
     //catch block2
}
. . .
. . .
catch(typeN arg)
{
     //catch blockN
}
```

Note that each catch statement responds only to its own type.

When an exception is thrown, the exception handler is searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated.

It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Program for using multiple catch statements

```
#include<iostream>
using namespace std;
void Ehandler(int i)
{
  try{
      if(i==1)
        throw i;     //int
    else
     if(i==0)
```

```
        throw 'i';    //char
      else
       if(i==-1)
         throw 3.14; //float
      }
}
catch(char c)       //catch1
{   cout << "Caught a character\n"; }
catch(int x)            //catch2
{   cout << "Caught an integer\n"; }
catch (float n)     //catch3
{   cout << "Caught a float\n"; }
int main()
{
  cout<< "Multiple Catch Statements\n";
  cout<< "i==1\n";
  Ehandler(1);
  cout<< "i==0\n";
  Ehandler(0);
  cout<< "i==-1\n";
  Ehandler(-1);
  cout<<"i==2\n";
  Ehandler(2);
  return 0;
 }
```

Output

Multiple Catch Statements

i==1

Caught an integer

i==0

Caught a character

i==-1

Caught a float

i==2

While executing a program, it will first execute the function Ehandler() with i==1 so it will throw i an int exception. This matches the type of the parameter i in catch2 and therefore catch2 handler is executed. Next the function Ehandler() with i==0 is invoked.

This time, the function throws 'i', a character type exception and therefore the first handler is executed. Lastly, the catch3 handler is executed when a float type exception is thrown.

Try block does not throw any exception, when the test() is invoked with i==2 . Note that every time only the handler, which catches the exception, is executed and all other handlers are bypassed.

When the try block does not throw any exceptions and it completes normal execution, control passes to the first statement after the last catch handler associated with that try block.

4. Catch-All Exception Handler

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is.

Its form is as follows:

```
catch(. . .)
{
   //process all exceptions
}
```

Here, the ellipsis matches any type of data.

Program for catching all exceptions

```
#include<iostream>
using namespace std;
void Ehandler(int i)
{
  try{
    if(i==0) throw i;      // throw int
    if(i==1) throw 'i';    // throw char
    if(i==2) throw 1.2;    // throw double
  }
catch(...)
 {   // catch all exceptions
    cout << "Caught an exception\n";
  }
}
int main()
{
  cout << "Begin\n";
  Ehandler(2);
  Ehandler(1);
  Ehandler(0);
  cout << "End";
  return 0;
 }
```

Output

Begin
Caught an exception
Caught an exception
Caught an exception
End

Note that all throws were caught by the one catch, i.e., (catch(. . .)) statement no matter what the type of it.

We can also use catch(. . .) as a default handler that catches all exceptions not caught by other handlers if it is specified at last in the list of handlers. If we specified it before other catch blocks would prevent those blocks from catching exceptions.

Program for using catch(. . .) as default handler

```
#include<iostream>
using namespace std;
void Ehandler(int i)
{  try{
      if(i==0) throw i; // throw int
      if(i==1) throw 'i'; // throw char
      if(i==2) throw 1.2; // throw double
   }
   catch(int x)
   {   // catch an int exception
       cout << "An Integer is Caught " << x << '\n';
   }
   catch(...)
  {   // catch all exceptions
      cout << "Caught an exception\n";
   }
}
int main()
{  cout << "Begin\n";
    Ehandler(0);
    Ehandler(1);
    Ehandler(2);
    cout << "End";
    return 0;
 }
```

Output

Begin
An Integer is Caught
Caught an exception
Caught an exception
End

From this example, we can conclude that using catch(...) as a default is a good way to catch all exceptions that you don't want to handle explicitly.

Also, by catching all exceptions, you can prevent an unhandled exceptions from causing an abnormal program termination.

5. Nesting Try-catch Blocks

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. *For example*

```
try {
  try {
      // code here
  }
  catch(int n) {
      throw;  //throw with no arguments
```

```
    }
}
catch(...) {
    cout << "Exception occurred";
}
```

6. Exception Specifications

C++ allows a function declaration to specify exactly what set of exceptions the function can throw via an exception specification. The exception specification is used as a suffix to a function declarator and is of the general syntax.

```
return-type function-name(arguments-list) throw(type-list)
{
    // Function body
}
```

An exception specification is not part of a function's type. Both argument-list and type-list are optional.

An empty exception specification (i.e., thow()) indicates that a function cannot throw any exception. A function with no exception specification is capable of throwing any type of exception and is the default function declarator. Each of the different types or classes that can be thrown by a function is separated by a comma-operator in the type-list argument. A function that attempts to throw an exception that is not listed in its exception specification will result in a call to the unexpected() function (which is explained later). The default behaviour of the unexpected() is to call abort(), which in turn causes abnormal program termination.

In type-list you cannot specify an incomplete type, a pointer or a reference to an incomplete type, other than a pointer to void, optionally qualified with const and/or volatile. You cannot define a type in an exception specification.

An exception specification may only appear at the end of a function declarator of a function, pointer to function, reference to function, pointer to member function declaration, or pointer to member function definition. An exception specification cannot appear in a typedef declaration. The following declarations demonstrate this:

```
void f() throw(int);
void (*g)() throw(int);
void h(void i() throw(int));
// typedef int (*j)() throw(int);   This is an error.
```

The compiler would not allow the last declaration, typedef int (*j)() throw(int).

A function can only be restricted in what types of exceptions it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function and not within it.

Following program illustrates exception specification.

Program for using exception specification

```
#include<iostream>
using namespace std;
// this function can only throw int and double exceptions
void FThrowIntDouble(int i) throw(int, double)
{
```

```
    if(i == 1) throw 12;           // throw int
    if(i == 2) throw 1234.2;       // throw double
}
// throw no exception
void FThrowNone(int i) throw()
{
    // these statements cannot be thrown!
    if(i == 1) throw 12;       // throw int
    if(i == 2) throw 1234.2;   // throw double
}
// throw any exception
void FThrowAny(int i)
{
    // these statements can be thrown!
    if(i == 1) throw 12;       // throw int
    if(i == 2) throw 1234.2;   // throw double
}
void main()
{
    try
    {
        FThrowIntDouble(1);     // ok
        FThrowIntDouble(2);     // ok
        FThrowAny(2);           // ok
        FThrowNone(1);          // abnormal program termination
    }
    catch(int i)
    {
        cout << "caught integer:" << i << "endl";}
    catch(double d)
    {
        cout << "caught double:" << d << "endl"; }
}
```

The function FThrowIntDouble() can throw either int or double exceptions, but if an attempt is made to throw any other type of exception abnormal program termination will occur and the unexpected() function will be called. FThrowNone() cannot throw any exceptions and an attempt to do so will result in an abnormal program termination.

FThrowAny() illustrates that a function without an exception specification can throw any exception.

7. Unexpected Exception

If the function throws an exception of a type not listed in the exception specification, or of a type not derived from one of the listed types, the standard function **unexpected()** is called. The default behavior for this function is to terminate the program.

The **syntax** is

```
void unexpected();
```

The unexpected() function simply calls the other functions (i.e., by default terminate()) to actually handle the error. However you can change the functions that are called by unexpected() by using set_unexpected() function.

set_unexpected()

You can install your own handler or change the unexpected handler by calling set_unexpected. To use **set_unexpected( )**, you must include the header file **<exception>**.

Its general form is as shown below.

```
unexpected_handler set_unexpected(unexpected_handler newhandler) throw();
```

Here, *newhandler* is a pointer to the new unexpected handler. The function returns a pointer to the old unexpected handler. Also, it returns the previous value of the **unexpected( )** pointer so you can save it and restore it later. The new unexpected handler must be of type unexpected_handler, which is defined as follows:

```
typedef void(*unexpected_handler)();
```

This handler should take any actions necessary for unexpected exceptions, and then terminate execution of the program. It can also throw exceptions of its own. However, it must not return to the program.

Here's an example that shows a simple use of unexpected specification.

Program for using unexpected specification

```
// Basic exceptions
// Exception specifications & unexpected()
#include<exception>
#include<iostream>
#include<cstdlib>
#include<cstring>
using namespace std;
class Up {};
class Fit {};
void g();
void f(int i) throw(Up, Fit) {
  switch(i) {
    case 1: throw Up();
    case 2: throw Fit();   }
  g();}
// void g() {}          // Version 1
void g() { throw 47; }  // Version 2
// Can throw built-in types
void my_unexpected() {
  cout << "unexpected exception thrown";
  exit(1);}
int main() {
  set_unexpected(my_unexpected);
  // (ignores return value)
  for(int i=1; i<=3; i++)
    try {  f(i);  }
       catch(Up u) {
      cout << "Up caught" << endl;
```

```
    } catch(Fit f) {
      cout << "Fit caught" << endl;     }
}
```

The classes **Up** and **Fit** are created solely to throw as exceptions. Often exception classes will be small, but sometimes they contain additional information so that the handlers can query them. **f( )** is a function that promises in its exception specification to throw only exceptions of type **Up** and **Fit**, and from looking at the function definition this seems possible. Version one of **g( )**, called by **f( )**, doesn't throw any exceptions so this is true. But then someone changes **g( )** so it throws exceptions and the new **g( )** is linked with **f( )**. Now **f( )** begins to throw a new exception, unknown to the creator of **f( )**. Thus the exception specification is violated. The **my_unexpected( )** function has no arguments or return value, following the proper form for a custom **unexpected( )** function. It simply prints a message so you can see it has been called, then exits the program. Your new **unexpected( )** function must not return (that is, you can write the code that way but it's an error). However, it can throw another exception (you can even rethrow the same exception), or call **exit( )** or **abort( )**. If **unexpected( )** throws an exception, the search for the handler starts at the function call that threw the unexpected exception. (This behavior is unique to **unexpected( )**.) In **main( )**, the **try** block is within a **for** loop so all the possibilities are exercised. Note that this is a way to achieve something like resumption – nest the **try** block inside a **for**, **while**, **do**, or **if** and cause any exceptions to attempt to repair the problem; then attempt the **try** block again. Only the **Up** and **Fit** exceptions are caught because those are the only ones the programmer of **f( )** said would be thrown. Version two of **g( )** causes **my_unexpected( )** to be called because **f( )** then throws an **int**. (You can throw any type, including a built-in type.) In the call to **set_unexpected( )**, the return value is ignored, but it can also be saved in a pointer to function and restored later.

8. Throwing an Exception from Handler

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression causes the originally thrown object to be rethrown. *The rethrow expression has following form:*

```
throw; // throw with no argument
```

This causes the current exception to be passed to the next outer try/catch sequence. This allows multiple handlers to manage the same exception. Any function called from within a catch block can also re-throw exceptions. Re-thrown exception will not be re-caught by the same catch block, but will propagate to the next higher level in the try/catch hierarchy.

Program for exception re-throwing

```
#include<iostream>
using namespace std;
void Ehandler() {
try
{ throw "Welcome";          // throw a string }
catch(char *str)
{ // inner-most handler
  cout << "Caught string inside Ehandler\n";
```

```
   throw;      // re-throw the same exception } }
main() { cout << "Begin\n";
try
{ Ehandler(); } // call function inside try
   catch(char *str) { // next outer handler
   cout << "Caught string inside main\n";}
cout << "End\n";
return 0;
 }
```

Output

Begin
Caught string inside Ehandler
Caught string inside main
End

9. Uncaught Exception

If an exception is not caught by any catch statement because there is no catch statement with a matching type, then it is said to be "uncaught" or "unhandled" exception. An uncaught exception also occurs if a new exception is thrown before an existing exception reaches its handler – the most common reason for this is that the constructor for the exception object itself causes a new exception. If an exception is uncaught, the special function terminate( ) is automatically called. Like unexpected( ), terminate is actually a pointer to a function. Its default value is the Standard C library function abort( ), which immediately exits the program with no calls to the normal termination functions (which means that destructors for global and static objects might not be called). The **syntax** is:

```
void terminate();
```

No cleanups occur for an uncaught exception; that is, no destructors are called. If you don't wrap your code (including, if necessary, all the code in **main( )**) in a try block followed by handlers and ending with a default handler (**catch(...)**) to catch all exceptions, then you will take your lump. An uncaught exception should be thought of as a programming error. In general, terminate() is the handler of last resort when no other handlers for an exception are available.

set_terminate()

You can install your own **terminate( )** function using the standard **set_terminate( )** function, which returns a pointer to the **terminate( )** function you are replacing, so you can restore it later if you want. In addition, any **terminate( )** handler you install must not return or throw an exception, but instead must call some sort of program-termination function. If **terminate( )** is called, it means the problem is unrecoverable. Like **unexpected( )**, the **terminate( )** function pointer should never be null. To change or install the terminate handler, use set_terminate() shown below:

```
terminate_handler set_terminate(terminate_handler newhandler) throw();
```

Here, newhandler is a pointer to the new terminate handler. The function returns a pointer to the old terminate handler. The new terminate handler must be of type terminate_handler, which is defined like this:

```
typedef void(*terminate_handler)();
```

The only thing that your terminate handler must do is stop program execution. It must not return to the program or resume it in any way.

Like set_unexpected() the set_terminate() also require the header <exception>.

Here's an example showing the use of **set_terminate( )**. Here, the return value is saved and restored so the **terminate( )** function can be used to help isolate the section of code where the uncaught exception is occurring.

Program illustrating the use of set_terminate() and uncaught exceptions

```
// Use of set_terminate() Also shows uncaught exceptions
#include<exception>
#include<iostream>
#include<cstdlib>
using namespace std;
void terminator()
{  cout << "I'll be back!" << endl;
  abort();}
void (*old_terminate)()
=set_terminate(terminator);
class Botch
{ public:
  class Fruit {};
  void f() {    cout << "Botch::f()" << endl;
    throw Fruit();  }
  ~Botch() { throw 'c'; }};
int main()
{ try
   { Botch b;
     b.f(); } catch(...)
   {  cout << "inside catch(...)" << endl; }
 }
```

The definition of **old_terminate** looks a bit confusing at first. It not only creates a pointer to a function, but it initializes that pointer to the return value of **set_terminate( )**. Even though you may be familiar with seeing a semicolon right after a pointer-to-function definition, it's just another kind of variable and may be initialized when it is defined. The class **Botch** not only throws an exception inside **f( )**, but also in its destructor. This is one of the situations that causes a call to **terminate( )**, as you can see in **main( )**. Even though the exception handler says **catch(...)**, which would seem to catch everything and leave no cause for **terminate( )** to be called, **terminate( )** is called anyway, because in the process of cleaning up the objects on the stack to handle one exception, the **Botch** destructor is called, and that generates a second exception, forcing a call to **terminate( )**. Thus, a destructor that throws an exception or causes one to be thrown is a design error. The C++ exception handling subsystem supplies one other a brand-new library function: uncaught_exception(). Its general form is as shown below:

```
bool uncaught_exception();
```

This function returns true if an exception has been thrown but not yet caught. Once caught, the function returns false.

Solved Programs

1. Write a program to accept a number. Raise an exception if the number is prime.

PU
Oct. 2009 – *5M*

Solution

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a,c=0,i,n;
    clrscr();
      cout <<"enter the number to be checked";
    cin >>n;
    for(i=1;i<=n; i++)
    {
        a=n%i;
        try
        {
          if(a=0)
          {
            c=c+1;
          }
          if(c=2)
          {
            cout << "the given number is prime";
          }
          else
            cout << "the given number is not prime";
        }
        catch(int i)
        {   cout << "Exception Caught";
        }
    }
    getch();
}
```

2. Write a program that demonstrates how certain exceptions types are not allowed to be thrown.

PU
Apr. 2010 – *5M*

Solution

C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function's caller that the function will throw only the exceptions contained in the exception specification.

```
#include<iostream.h>
void myFun(int v)throw(char,float)
if(v==1)
{ throw v; }
else if(v == 2)
{ throw 'c'; }
else if(v == 3)
{ throw 2.0f; }
int main()
{
```

```
    try
    { myFun(1); // throw int
      myFun(2); // throw char
      myFun(3); // throw float
      myFun(4); // Exception not allowed to be thrown
    }
    catch(int i)
    { cout<< "\n Integer related exception found"; }
    catch(char ch)
    { cout<< "\n Character related exception found"; }
    catch(float f)
    { cout<< "\n Float related exception found"; }
    return 0;
}
```

EXERCISES

A. Review Questions

1. What is an exception?
2. How is an exception handled in C++?
3. What is a try block?
4. What is a catch statement?
5. What information can an exception contain?
6. When are exception objects created?
7. What does catch (...) mean?
8. What is an exception specification? When it is used?
9. When do we use multiple catch handlers?
10. Explain terminate and unexpected function.

B. Programming Exercises

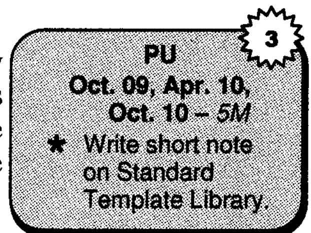
1. Write a program containing a possible exception. Use a try block to throw an exception and catch block to handle it properly.
2. Write a program, which illustrates the use of multiple catch statements.
3. Write a program using catch(. . .) handler.
4. Write a program for calculators which perform simple arithmetic operations using exception handling mechanism.

Collection of Questions asked in Previous Exams PU

1. Write a program to accept a number. Raise an exception if the number is prime. **[Oct. 2009 – 5M]**
2. Write a program that demonstrates how certain exceptions types are not allowed to be thrown. **[Apr. 2010 – 5M]**
3. Write short note on Exception Handling Mechanism. **[Oct. 2010 – 5M]**

13 Introduction To Standard Template Library

I. Introduction



The Standard Template Library (STL) is a general-purpose C++ library and it is a collection of algorithms and data structures. The data structures used in the algorithms are abstract in the sense that the algorithms can be used on (practically) every data type. The algorithms can work on these abstract data types due to the fact that they are template based algorithms.

STL was originated by Alexander Stepanov and Meng Lee. The STL, based on a concept known as generic programming, is part of the standard ANSI/ISO C++ library. The STL is implemented by means of the C++ template mechanism, hence its name. While some aspects of the library are very complex, it can often be applied in a very straightforward way, facilitating reuse of the sophisticated data structures and algorithms it contains.

The STL provides some nice features such as handling memory for you (no memory leaks), it is also safer (no buffer overflow issues when using vectors or similar data structures).

All elements or components of the STL are defined in the standard namespace. Therefore, a "using namespace std" or comparable directive is required unless it is preferred to specify the required namespace explicitly.

The using namespace directive informs the compiler that we intend to use Standard C++ Library.

VISION

2. The STL Programming Model

The STL would not have been possible without the use of C++ templates; class and function templates are used throughout the STL. Templates provide not only the efficiency needed for a generic component library, but also make the library extensible. Templates allow the STL to work with built-in types and user-defined types in a seamless way.

Templates are still a recent addition to C++ and many compiler vendors do not provide all of the features as suggested in the April 1995 ANSI draft. The STL depends heavily on many of these advanced features and, in some cases, relies on workarounds to accommodate the current generation of compilers.

There are six components in the STL organization. Three components, in particular, can be considered the core components of the library:

i. **Containers** are data structures that manage a set of memory locations.

ii. **Algorithms** are computational procedures.

iii. **Iterators** provide a mechanism for traversing and examining the elements in a container.

An STL data structure or container, does not contain many member functions. STL containers contain a minimal set of operations for creating, copying, and destroying the container along with operations for adding and removing elements. You will not find container member functions for examining the elements in a container or sorting them. Instead, algorithms have been decoupled from the container and can only interact with a container via traversal by an iterator. This relationship is explained in the *figure 13.1.*

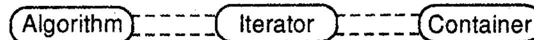


Figure 13.1: Containers, algorithms, and iterators form an orthogonal component space in the STL

This orthogonal structure is what brings the STL its power, flexibility, and extensibility. Developers that implement new algorithms utilizing one of the STL iterators are guaranteed that their algorithms will work with existing container types as well as those that have not yet been developed.

The remaining three components of the STL are also fundamental to the library and contribute to its flexibility and portability:

iv. **Function objects** encapsulate a function as an object.

v. **Allocators** encapsulate the memory model of the machine.

vi. **Adaptors** provide an existing component with a different interface.

Let's take a look at each of the STL components in a little more detail.

3. Containers

Containers are STL objects that actually store data. They use certain basic properties of the objects (ability to copy, etc.) but otherwise do not depend on the type of object they contain. STL containers may contain pointers to objects, though in this case you will need to do a little extra work. There are several different types of containers, which are grouped into three categories namely, Sequence Container, Associative Container and Derived Containers or Container Adapters.

To give you a brief idea of the containers that are available, here is the hierarchy:

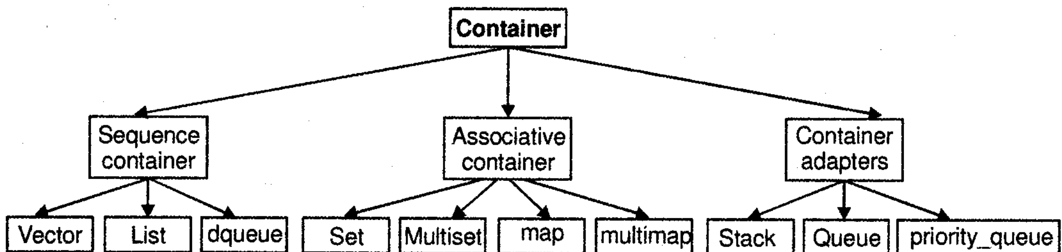


Figure 13.2: Hierarchy of three major categories of containers

Following *table 13.1* describes the details of the containers defined by the STL as well as the header file required to use each container.

Table 13.1

Container	Description	Required Header File	Iterator type
Sequential Containers			
vector	A dynamic array of variables, structures or objects. Insertion / deletion of element at the end and allows direct access to any element.	<vector>	Random access
list	A linked list of variables, structures or objects. Insertion / deletion of element anywhere.	<list>	Bidirectional
deque	A double-ended queue, it is an array which supports insertion / removal of elements at beginning or end of array. It allows direct access to any element.	<deque>	Random access
Associative Containers			
set	A set in which each element is unique, i.e., duplicate data is not allowed.	<set>	Bidirectional
multiset	A set in which each element is not necessarily unique, i.e., duplication allowed.	<set>	Bidirectional
map	A map stores unique key/value pairs. Each key value is associated with only one value.	<map>	Bidirectional
multimap	A multimap stores key/value pairs in which one key may be associated with two or more values, i.e., duplicate keys allowed.	<map>	Bidirectional

Derived Containers / Container Adapters / Sequence Adapters			
stack	A stack works in LIFO, i.e., Last In First Out manner	<stack>	No iterator
queue	A queue works in FIFO, i.e., First In First Out manner	<queue>	No iterator
priority queue	A priority queue. The element which is first out is always the highest priority element	<queue>	No iterator

Each container class defines a set of functions that may be used to manipulate its contents. *For example*: A vector container defines functions for inserting and erasing an element and swapping the contents of two vectors. A list includes functions for inserting, deleting and merging elements.

▶ Sequence Containers

A sequence is a container that stores a finite set of objects of the same type in a linear organization. An array name is a sequence. You use one of the following three sequence types for a particular application depending on its retrieval requirements.

Following are the three types of sequence containers:

i. **vector**: A vector is a sequence that you can access at random. Insertion/deletion of element is fast at the end of the vector but it takes more time (i.e., slow) at the beginning or in the middle of the vector because they involve shifting the remaining element to make room or to close the deleted element space, Vector allows fast random access. It supports a dynamic array.

ii. **list**: A list is sequence that you can access bi-directionally, i.e., at both end. Insertion/deletion of element is fast and we can insert/delete element anywhere, but provide slow sequential access.

iii. **deque**: A deque is like a vector, except that deque allows fast insertion/deletion at beginning as well as at the end of the container. Insertion/deletion of the element in the middle takes more time (i.e., slow). It allows fast random access.

▶ Associative Containers

Associative containers are a generalization of sequences. Sequences are indexed by integers; where as associative containers can be indexed by any type.

Associative containers provide efficient retrieval of values based on keys. We can use associative containers for large dynamic tables that you can search sequentially or at random. Associative containers use tree-structures to store data rather than using contiguous arrays or linked lists. These structuressupport fast random retrievals and updates.

The set, multiset, map and multimap are called associative containers because they associate keys with values.

i. **set:** Set allows you to add and delete elements, query for membership, and iterate through the set.

ii. **multisets:** Multisets are just like sets, except that you can have several copies of the same element (these are often called bags).

iii. **maps**: Maps represent a mapping from one type (the key type) to another type (the value type).
You can associate a value with a key, or find the value associated with a key, very efficiently; you can also iterate through all the keys.

iv. **multimaps**: Multimaps are just like maps except that a key can be associated with several values.

The most important basic operations with associative containers are putting things in and in case of set seeing if something is in the set. In case of a map you want to first see if a key is in the map and if it exists you want the associated value for that key. There are many variations on this theme but that's the fundamental concept.

▶ Container Adapters

Container adapters are created from the existing sequence containers. The derived containers do not support the iterators and therefore we cannot use them for manipulating the data.

STL includes three container adapters: stack, queue and priority_queue, which can be summarized as follows:

i. **stack:** A stack is a data structure, which supports pushdown, pop-up behaviour. The element which is inserted (pushed) most recently is the only one that can be extracted. Extraction of the logically topmost element pops that element from the stack removes it so that the element inserted immediately before the popped element is now the next available element. The stack template class supports two functions: push() and pop() to insert and extract elements in the data structure.

ii. **queue:** A queue is a data structure wherein we can insert elements at the end and extract elements from the beginning. The queue template class supports two functions: push() and pop() to insert and extract elements in the data structure.

iii. **priority_queue:** A priority_queue is a data structure wherein we can insert elements at the end and extract element that has the highest priority. The priority_queue template class supports two functions: push() and pop() to insert and extract elements in the data structure.

3.1 Constructing a Sequence Container

There are multiple constructors provided by each container class to create container objects. The different constructors provide ways to specify the size of the container and initial values for its elements.

Here are some of the most common ways of construction of container, i.e., vector, list and deque.

```
i.   vector/deque/list<type> name
ii.  vector/deque/list<type> name(size)
iii. vector/deque/list<type> name(size, value)
iv.  vector/deque/list<type> name(myvector/mydeque/mylist)
v.   vector/deque/list<type> name(first, last)
```

Let's take a look at a simple example showing the various constructors.

```
#include<vector>        // Needed to use vectors
#include<deque>       // Needed to use deques
#include<list>          // Needed to use lists
#include<string>
```

```
using namespace std;
int main() {
// 1. Creating an empty container
vector<int> v1;     //Declare int vector v1 of zero size
list<string> l1;    //Declare a list of string type l1 of zero size
deque<float> d1;    //Declare dqueue of float type d1 of zero size
// 2. Creating a container of some size using the default value for
// built-in types or the default constructor for user-defined
// types such as classes
vector<int> v2(100);
//Creates a vector of 100 ints with initial values of 0
  deque<int> d2(5);
// Creates a deque of 5 ints with initial values of 0
// 3. Creates a vector of some size with specified initial values
vector<int> v3(100, -5);
// Creates a vector of 100 integers with initial values of -5.
   vector<string> v4(4, "dog");
// Creates a vector containing 4 strings initialized to "dog".
    list<string> l2(5, "cat")
//  Creates a list containing 5 strings initialized to "cat". This
//  allows for efficient deletion and insertion of cats in the
//  interior of the sequence.
// 4. Creates a vector by providing start and end iterators to another
//container.
string s1("Hello World");
vector<char> v5(s1.begin(), s1.end());
// Creates a vector containing "Hello World"
  list<int> l3(v3.begin(), v3.end());
// Creates a list containing the elements of v3
  deque<int> d3(v3.begin(), v3.end());
// Creates a deque containing the elements of v3
    int numbers[5] = {9, 3, 2, 5, 6};
    vector<int> v6(numbers, numbers + 3);
// Creates a vector containing the first three elements of the    array
numbers.
// 5. Creates a container initialized from another container.
    vector<int> v7(v6);    // creates a vector v7 from v6
    list<int> l4(l3);      // creates a list l4 from l3
    deque<int> d4(d2);        // creates a dqueue d4 from d2
    return 0;
 }
```

3.2 Application of Container

i. Vector Class Template

The vector is part of the C++ Standard Template Library (STL); a combination of general-purpose, templatized classes and functions that implement a wide range of common data structures

and algorithms. The vector is considered a container class. Like containers in real-life, these containers are objects that are designed to hold other objects. In short, a vector is a dynamic array designed to hold objects of any type, and capable of growing and shrinking as needed.

A vector is able to access elements at any position (i.e.,"random" access) with a constant time overhead,O(1). Insertion or deletion at the end of a vector is faster. As with the string, no bounds checking is performed when you use operator [].

Insertions and deletions anywhere other than at the end of the vector is not efficient (time overhead is O(N), N being the number of elements in the vector) because all the following entries have to be shuffled along to make room for the new entries, the storage being contiguous. Memory overhead of a vector is very low comparable to a normal array.

In order to use vector, you need to include the following header file:

```
#include<vector>
```

The vector is a part of the std namespace, so you need to qualify the name. This can be accomplished as shown below:

```
using std::vector;
vector<int> vInts;
```

or you can fully qualify the name like this:

```
std::vector<int> vInts;
```

or by declaring global namespaces such as:

```
using namespace std;
```

As for the interface to the vector container, the member functions and operators of vector are listed in the tables below.

Vector Member Functions

Following table shows some of the main vector functions. We can also use all the STL algorithms on a vector.

Function	Description
assign()	Erases a vector and copies the specified elements to the empty vector.
at()	Returns a reference to the element at a specified location in the vector.
back()	Returns a reference to the last element of the vector.
begin()	Returns a random-access iterator to the first element in the container.
capacity()	Returns the number of elements that the vector could contain without allocating more storage.
clear()	Erases the elements of the vector.
empty()	Returns True if the vector container is empty.
end()	Returns a random-access iterator that points just beyond the end of the vector.
erase()	Removes an element or a range of elements in a vector from specified positions.
front()	Returns a reference to the first element in a vector.
get_allocator()	Returns an object to the allocator class used by a vector.
insert()	Inserts an element or a number of elements into the vector at a specified position.
max_size()	Returns the maximum length of the vector.
pop_back()	Deletes the element at the end of the vector.
push_back()	Adds an element to the end of the vector.

rbegin()	Returns an iterator to the first element in a reversed vector.
rend()	Returns an iterator to the end of a reversed vector.
resize()	Specifies a new size for a vector.
reserve()	Reserves a contiguous block of memory for the vector
reverse()	Reverses the order of the elements in the vector.
size()	Returns the number of elements in the vector.
swap()	Exchanges the elements of two vectors.
vector()	Constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of some other vector.

Operators

Some of the operators defined for the vector container are:

Operator	Meaning

=	The assignment operator replaces the target vector's contents with that of the source vector:
	vector<int> a;
	vector<int> b;
	a.push_back(5);
	a.push_back(10);
	b.push_back(3);
	b = a; // The vector b now contains two elements: 5, 10

[ ]	The subscript operator returns a reference to an element at the specified position of the vector. A subscript value of zero returns a reference to the first element, and so on. The subscript must be between zero and size()-1. You can also use the subscript operator in a loop to access elements of a vector. The subscripted vector may appear on the left or right sides of an assignment (the returned reference is an lvalue):
	vector<double> vec;
	vec.push_back(1.2);
	vec.push_back(4.5);
	vec[1] = vec[0] + 5.0;
	vec[0] = 2.7; // Vector now has two elements: 2.7, 6.2

!=	Test whether two vectors are unequal
<	Test whether one vector is less than another
<=	Test whether one vector is less than or equal to another
==	Test whether two vectors are equal
>	Test whether one vector is greater than another
>=	Test whether one vector is greater than or equal to another

Following program illustrates the use of several functions of the vector class template:

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
  vector<int> v;      //creates an empty vector v of type int
```

```
cout<<"Original size of a vector is:"<<v.size()<<"\n";
// putting values into the vector
int a;
cout << "Enter three integer values:";
for(int i=0;i<3;i++)
{ cin >> a;
  v.push_back(a);
}
cout<<"Size after adding 3 values:";
cout<<v.size()<<"\n";
//Display the contents
cout<< "Contents of v:\n";
for(i=0;i<v.size();i++)
  cout<<v[i]<<" ";
  cout<<"\n";
// Insert one more value
v.push_back(40.6);

//Display the size and contents after adding value
cout<< "\n Size after inserting value:"<<v.size()<<"\n";
cout<< " Now Contents are:\n";
for(i=0;i<v.size();i++)
  cout<<v[i]<<" ";
  cout<<"\n";
//Inserting elements
vector<int>::iterator p=v.begin();      //iterator
p+=2;       // p points to 3rd element
v.insert(p,1,3);
//Display the size and contents after adding value
cout<< "\n Size after inserting value 3:"<<v.size()<<"\n";
cout<< "Contents after insert:\n";
for(i=0;i<v.size();i++)
  cout<<v[i]<<" ";
  cout<<"\n";
//removing 2nd and 3rd element
v.erase(v.begin()+1,v.begin()+2);
//Display the contents after deletion
cout<< "\n Size after erase:"<<v.size()<<"\n";
cout<< "Contents after erase:\n";
for(i=0;i<v.size();i++)
cout<<v[i]<<" ";
cout<<"\n";
return 0;
}
```

Output

Original size of a vector is: 0

Enter three integer values: 10 20 30

Size after adding 3 values: 3

Contents of v: 10 20 30

Size after inserting value: 4

Now Contents are: 10 20 30 40

Size after inserting value 3: 5

Contents after insert:10 20 3 30 40

Size after erase:3

Contents after erase:10 30 40

Explanation: In the above program, in main(), an integer vector called v is created with an initial capacity 0, i.e., initially v contains no elements. Next, the three elements are added at the end of the vector v using the push_back() function. The push_back() takes the value as an argument and add it to the end of the vector. You can also use insert() function to insert the value into the vector. It takes as arguments the position at which to start the insertion, the number of elements to insert and the value to insert. To insert a single element, simply supply an iterator for the position and the value to insert.

Then after inserting the values using push_back() to know the size of the vector size( ) function is used. So after inserting 3 values the content of the vector is 3. Since the type of the vector is int, so it can accept only integer values and therefore the statement.

v.push_back(40.6) truncates the values 40.6 to 40 and then puts it into the vector at its back end.

The program uses an iterator to access the vector elements, the statements

```
vector<int>::iterator p=v.begin();        //iterator
```

declares an iterator p and makes it point to the first position or start of the vector by using the begin() member function. This function returns an iterator to the start of the vector. The statement

```
p+=2;        // p points to 3rd element
v.insert(p,1,3);
```

inserts value 3 as the 3rd element. Similarly the statement

```
v.erase(v.begin()+1,v.begin()+2);
```

deletes 2nd and 3rd elements from the vector. To delete the elements from the vector we use the erase() or pop_back() function. The erase() function takes as arguments the position at which to start the deletion and the position of the element at which the deletion should stop(i.e., this last element is not deleted). The second argument is not required. To delete a single element, simply supply an iterator for the element as the erase() member function's single argument. If one wants to remove all the elements at once from the vector, the vector.clear() function can be used.

ii. List Class Template

The list container implements a classic list data structure; unlike a C++ array or an STL vector, the objects it contains cannot be accessed directly (i.e., by subscript). The list container is defined as a template class, meaning that it can be customized to hold objects of any type.

List is a Sequence that supports both forward and backward traversal, and provide a constant time insertion and removal of elements at the beginning or the end, or in the middle. Compared to vectors, they allow fast insertions and deletions.

Lists don't provide random access like an array or vector, They support bidirectional iterators.

To be able to use STL lists add this header file before you start using them in your source code:

```
#include<list>
```

Member functions

Some commonly used member functions of the list class are:

Function	Description
back()	Returns a reference to the last element of the list.
begin()	Returns a random-access iterator to the first element in the list.
clear()	Erases the elements of the list.
empty()	Returns True if the list is empty.
end()	Returns a random-access iterator that points just beyond the end of the list.
erase()	Removes an element or a range of elements in a list from specified positions.
front()	Returns a reference to the first element in a list.
Insert()	Inserts an element or a number of elements into the list at a specified position.
max_size()	Returns the maximum length (the greatest number of elements that can fit) of the list.
merge()	Merge one list into other.
pop_back()	Deletes the last element of the list.
pop_front()	Deletes the first element of the list.
push_back()	Adds an element to the end of the list.
pop_front()	Adds an element at the start of the list.
rbegin()	Returns an iterator to the first element in a reversed list.
rend()	Returns an iterator to the end of a reversed list.
resize()	Specifies a new size for a list.
remove()	Removes specified elements.
remove_if	Removes elements conditionally.
reverse()	Reverses the order of the elements in the list.
size()	Returns the number of elements in the list.
swap()	Exchanges the elements of two lists.
sort()	Sorts the list elements in ascending order.
splice()	Merges two lists in constant time.
swap()	Swaps the contents of this list with another list.
unique()	Removes duplicate elements from the list.

In addition to these member functions, some STL algorithms (e.g., find) can be applied to the list container.

Note: The list template class is based on a doubly-linked list, and has all the functions of vector except capacity, reserve, at.

Operators

Some of the operators defined for the list container are:

Operator	Meaning

=	The assignment operator replaces the target list's contents with that of the source list:
	For example: list<int> a;
	list<int> b;
	a.push_back(5);
	a.push_back(10);
	b.push_back(3);
	b = a;
	// The list b now contains two elements: 5, 10

!=	Test whether two lists are unequal
<	Test whether one list is less than another
<=	Test whether one list is less than or equal to another
==	Test whether two lists are equal
>	Test whether one list is greater than another
>=	Test whether one list is greater than or equal to another

Note that there is no subscript operator for the list container, since random access to elements is not supported.

Following program illustrates the use of the above member functions:

```
#include<iostream>
#include<list>
#include<cstdlib>        // for using r and() function
using namespace std;
void display(list<int> &lst)
{
  list<int>::iterator iter;
  for(iter=lst.begin();iter!=lst.end();++iter)
  cout<<*iter;
    cout<<"\n\n";
}
int main()
{
  list<int>list1;        // Create an empty list
  list<int>list2(5);    // Create a list having 5 elements.
  int i;
  //Insert the element into the 1st list
  for(i=0;i<3;i++)
  list1.push_back(rand()/100);
  //Insert the element into the 2nd list
  list<int>::iterator iter;
  for(iter=list2.begin(); iter=list2.end();++iter)
    *iter=rand()/100;
  cout<<"The content of the list 1:";
```

```
  display(list1);
  cout<<"The content of the list 2:";
  display(list2);
  //Inserting elements at the end of the list2
  list2.push_front(10);
  list2.push_back(20);
  //Erasing elements at the front and back of the list1
  list1.pop_front();
  list1.pop_back();
  cout<<"Now the content of the list 1:";
  display(list1);
  cout<<"The content of the list 2:";
  display(list2);
  list<int> lst1,lst2;
  lst1=list1;      // Initialization of objects
  lst2=list2;
  //Merge the two unsorted lists
  list1.merge(list2);
  if(list2.empty())
    cout<<"List2 is now empty");
  cout<<"Content of list1 after merge:";
  display(list1);
  //Sort and Merge
  lst1.sort();
  lst2.sort();
  lst1.merge(lst2);
  cout<<"The merged sorted list is:";
  display(lst1);
  //Reverse a list
  lst1.reverse();
  cout<<"Reversed Merged List is:";
  display(lst1);
  return 0;
}
```

Output

The content of the list 1 : 1 2 3
The content of the list 2 : 30 40 50 60 70
Now the content of the list 1:2
The content of the list 2: 10 30 40 50 60 70 20
Content of list1 after merge: 2 10 30 40 50 60 70 20
List2 is now empty
The merged sorted list is: 2 10 20 30 40 50 60 70
Reversed Merged List is: 70 60 50 40 30 20 10 2

Explanation: The program uses two empty lists: list1 with zero length and list2 of size 5. In list1 we insert the value using push_back() and a math function rand() and in list2 we use a list type iterator iter and a for loop to insert values. The begin() function give the first position of the element

and end() function gives the position immediately after the last element respectively. We insert the element in the list2 at both ends using push_front() and push_back() functions. At the same time remove the first and last element from list1 using the pop_front() and pop_back() functions.

The merge() function merges list1 with list2 and the result is placed in the list1 hence the list2 becomes empty. Here we use the empty() function, which returns true if the invoking container is empty.

A list may be sorted in ascending order using a sort() member function and reverse() function is used to reverse the content of the list.

The display() function is used to display the contents of lists.

iii. Deque Class Template

The deque class is a sequence container that is part of the Standard Template Library, or STL. Deques, like vectors, provide random access iterators that allow constant time (fast) access to any of their elements and can be used with any of the generic algorithms. They also are optimized for insertions and deletions at their beginnings and ends. Due to internal structure of deques, they tend to be slightly less efficient than vectors. That is, manipulations using deques will operate slightly slower than the same operations on vectors. Deques are the container of choice if you will be doing insertions and deletions at both ends. As with vectors, if insertions and deletions will be done in the interior of the sequence, lists may be a better choice.

Member Functions

Following are the deque (pronounced deck) methods. Notice that they are the same as the vector member functions with two exceptions. First, two member functions have been added to insert and delete from the front: pop_front and push_front. Remember that deques are optimized to allow insertions and deletions at both their beginnings and ends. Second, the capacity and reserve methods, present in the vector class, are not part of deque. This is because the issues with the size of the allocated block of memory are pertinent only to vector. Vector requires a contiguous block of memory. Deque has a different internal structure. To be able to use STL deque add this header file before you start using them in your source code: `#include<deque>`

Function	Description
assign	Clears a deque and assigns the specified elements to it
at	Returns a reference to the element at the specified position
back	Returns a reference to the element at the end of the deque
begin	Returns an iterator to the start of the deque
clear	Removes all elements from the deque
empty	Tests whether the deque is empty
end	Returns an iterator to the end of the deque
erase	Removes element(s) at specified position(s)
front	Returns a reference to the first element in the deque
insert	Inserts element(s) at specified position(s)
max_size	Returns the maximum size of the deque

pop_back	Removes the element at the end of the deque
pop_front	Removes the element at the front of the deque
push_back	Adds an element to the end of the deque
push_front	Adds an element to the front of the deque
rbegin	Returns a reverse iterator to end of dqueue
rend	Returns a reverse iterator to beginning of dqueue
resize	Specifies a new size for a list
size	Returns the number of elements in the deque
swap	Exchanges the contents of two deques

Operator

Operator	Meaning
[]	Returns a reference to the element at the specified position
!=	Test whether two deques are unequal
<	Test whether one deque is less than another
<=	Test whether one deque is less than or equal to another
==	Test whether two deques are equal
>	Test whether one deque is greater than another
>=	Test whether one deque is greater than or equal to another

Most of the methods of deque are same as vector. The program given for vector will also work for deque. Following program shows how to use some of methods of the deque class that were not illustrated in the vector example. Remember that both vector and deque allow fast random access to their elements, with vector being slightly faster. Vectors are optimized for insertions and deletions at their ends. Deques are optimized for insertions and deletions at both ends.

```
#include<deque>     // Needed to use the deque class
#include<iostream>
using namespace std;
int main() {
// Create an empty deque
deque<float> deq1;
// Create an iterator variable
deque<float>::iterator iter = deq1.begin();

// Load from the back notice the ordering of the elements in the output
deq1.push_back(2.5);
deq1.push_back(3.5);
// iter was initialized above
for(; iter != deq1.end(); iter++)
{
  cout << *iter << " ";
}
```

```
cout << endl;
// Load from the front
// Notice the ordering of the elements in the output
deq1.push_front(67.8);
deq1.push_front(33.3);
for(iter = deq1.begin(); iter != deq1.end(); iter++)
{
  cout << *iter << " ";
}
cout << endl;
cout << "Deque contains " << deq1.size() << " elements" << endl;
// Clear the deck or deque
deq1.clear();

// Check if deque is emtpy
if(deq1.empty()) {
  cout << "Deque is empty" << endl;
}
else {
  cout << "Deque is not empty" << endl;
}
cout << "Deque contains " << deq1.size() << " elements" << endl;
return 0;
}
```

Output

2.5 3.5
33.3 67.8 2.5 3.5
Deque contains 4 elements
Deque is empty
Deque contains 0 elements

The following example shows how to combine the use of the generic algorithms with the deque container class. Notice how easy it is to search for a value and to merge two sequences. The combination of the container classes and the generic algorithms leads to simple programming solutions.

```
#include<deque>          // Needed to use the deque class
#include<algorithm>     // Needed for generic algorithms
#include<iostream>
using namespace std;
int main() {
// Create two deques containing number sequences

int arr1[5] = {4,8,6,7,5};
deque<int> deq1(arr1, arr1 + 5);
int arr2[5] = {4,10,8,6,12};
deque<int> deq2(arr2, arr2 + 5);
// Create an iterator variable
deque<int>::iterator iter;
// Search for a value within deq1
```

```
iter = find(deq1.begin(), deq1.end(), 4);
if(iter == deq1.end()) {
   cout << "deq1 does not contain 4" << endl;
}
else
{
  cout << "deq1 contains " << *iter << endl;
}
// Merge two sorted sequences into a single deque
// Both deq1 and deq2 must be sorted before using merge
// The mergedResults deque must be large enough to hold the results
deque<int> mergedResults(deq1.size() + deq2.size());
sort(deq1.begin(), deq1.end());

sort(deq2.begin(), deq2.end());

merge(deq1.begin(), deq1.end(), deq2.begin(), deq2.end(),

mergedResults.begin());
for(iter = deq1.begin(); iter != deq1.end(); iter++)
{
   cout << *iter << " ";
}
cout << endl;
for(iter = deq2.begin(); iter != deq2.end(); iter++)
{
    cout << *iter << " ";
}
cout << endl;
for(iter = mergedResults.begin(); iter != mergedResults.end(); iter++)
{
   cout << *iter << " ";
}
cout << endl;
return 0;
 }
```

Output

deq1 contains 4

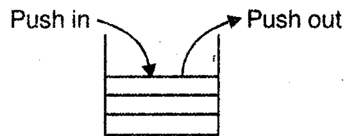
4 5 6 7 8

4 6 8 10 12

4 4 5 6 6 7 8 8 10 12

iv. A Stack Container Adapter

A stack is a data structure that supports LIFO, Last In First Out Processing. The last object pushed (placed) on the stack will be the first to be popped (removed). Figure shows the representation of stack.



The stack() constructor creates an empty stack. By default it uses the a deque as a container but a stack can only be accessed in last in first out manner. You may also use a vector or list as a container for a stack.

Member Function

The following members functions are defined for stack

Function	Description
empty	True if the stack has no elements
pop	Removes the top element of a stack
push	Adds an element to the top of the stack
size	Returns the number of items in the stack
top	Returns the top element of the stack

The comparison operators defined for stack are : ==, <,<=,!=,>,>=

Stacks are only accessed at their top. To be able to use *STL* stacks in a file of C++ source code or a header file add `#include<stack>` at the beginning of the file.

Following program illustrates how to create and manipulate stack:

```
#include<iostream>
#include<stack>
using namespace std;
int main()
{ stack<int, list<int>> st;
  cout<< "Add three elements into the stack:";
  for(int i=0;i<3;++i)
  {
    st.push(i);        // push three elements into the stack
    cout<<i<<"n";
  }
 // pop and print all elements from the stack
  cout<<"n The popped elements from the stack are:";
  int size=st.size();
  for(int i=0;i<size;++i)
  {
      cout << st.top() << ' ';
      st.pop();
  }
  return 0;
 }
```

Output

Add three elements into the stack:
1
2
3
The popped elements from the stack are:3 2 1

v. The Queue Container Adapter

A Queue is a data structure that supports FIFO, first in first out processing. Objects will be processed in the order they enter a queue, i.e., the elements are inserted at one end and taken out from the other end.

Figure 13.3 shows a representation of the way a queue normally processes data.

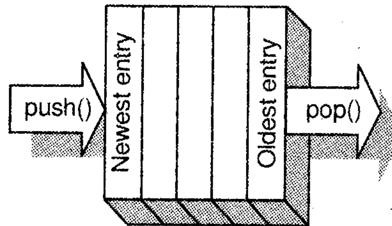


Figure 13.3: A FIFO queue

A queue constructor creates an empty queue. By default it uses a deque container but a queue can only be accessed in FIFO manner. You can also use list as a container for queue.

The comparison operators defined for queue are : ==,<,<=,!=,>,>=

To be able to use *STL* queues add header file(# include <queue>) before you start using them in your source code:

```
#include<queue>
```

Member Function

Function	Description
back	returns a reference to last element of a queue
empty	true if the queue has no elements
front	returns a reference to the first element of a queue
pop	removes the top element of a queue
push	adds an element to the end of the queue
size	returns the number of items in the queue

Following program illustrates how to create and manipulate queue.

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

```
{ queue<int, list<int>> qe;
  cout<< "Add three elements into the queue:";
  for(int i=0;i<3;++i)
  {
    qe.push(i);        // push three elements into the queue
    cout<<i<<"n"; }
 // pop and print all elements from the queue
  cout<<"\n The popped elements from the queue are:";
  int size=qe.size();
  for(int i=0;i<size;++i)
  {
    cout << qe.front() << " ";
   qe.pop(); }
  return 0;
 }
```

Output

Add three elements into the queue:

1

2

3

The popped elements from the queue are:1 2 3

vi. The Priority Queue Container Adapter

Priority Queues are like queues. The elements are popped from the sequence in order of priority which is based on the supplied comparison function (called a predicate). By default the predicate is less<>, i.e., while adding or removing a value from the priority queue the contents are arranged in descending order.

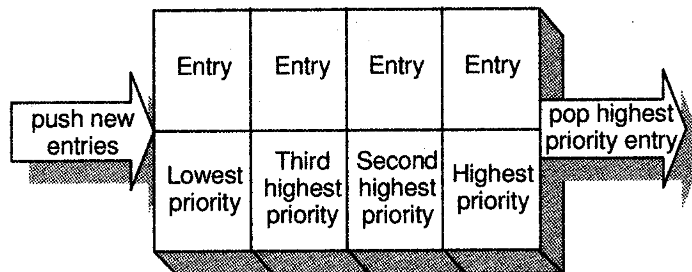


Figure 13.4: A priority queue

Figure 13.4 shows a representation of a priority queue. This type of queue assigns a priority to every element that it stores. New elements are added to the queue using the push() function, just as with a FIFO queue. This queue also has a pop() function, which differs from the FIFO pop() in one key area. When you call pop() for the priority queue, you don't get the *oldest* element in the queue. Instead, you get the element with the highest priority.

A priority queue constructor creates an empty priority queue. By default it uses vector as a container but you can also use a deque as a container for priority queue.

To be able to use STL priority queues add this before you start using them in your source code:
`#include<queue>`

Member Function

Function	Description
empty	true if the priority queue has no elements
pop	removes the top element of a priority queue
push	adds an element to the end of the priority queue
size	returns the number of items in the priority queue
top	returns the top element of the priority queue

Following program shows how to create priority queue and manipulate it with different functions:

```
#include<iostream>
#include<list>
#include<queue>
using namespace std;
int main()
{
    priority_queue<float, vector<float> > q;
    // insert six elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
   // pop and print the elements
   cout<< "Elements removed from the priority queue:";
   int size=q.size();
   for(int i=0; i<size;++i)
   {  cout << q.top() << ' ';
       q.pop();
   }
     return 0;
}
```

Output

Elements removed from the priority queue:
66.6 55.5 44.4 33.3 22.2 11.1

In the above program, we define the priority queue using the default less<> predicate, therefore the values removed from the sequence in order from largest to smallest. If we define the priority queue using the greater<> predicate then the highest priority value is the smallest because the values are arranged in the priority queue in ascending order. To use the greater<>predicate just replace the line

```
priority_queue<float, vector<float> > q;
```

with

```
priority_queue<float, vector<float>, greater<float>  > q;
```

and the output will be

Elements removed from the priority queue:

```
11.1 22.2 33.3 44.4 55.5 66.6
```

4. Algorithms

The STL algorithms are template C++ functions used to perform operations on containers. Although each container provides functions for its own basic operations, the standard algorithms provide more extended or complex actions. Also we can work with two different types of containers at the same time by using algorithms. By using `<algorithm>` header file in our program we can access the STL algorithms.

In order to be able to work with many different types of containers, the algorithms do not take containers as arguments. Instead, they take iterators that specify part or all of a container. In this way the algorithms can be used to work on entities that are not containers; *for example*: the function copy can be used to copy data from standard input into a vector.

The generic algorithms fall into following four categories:

Categories of Generic Algorithms

i. Non-modifying Sequence
ii. Mutating Sequence
iii. Sorting
iv. Numeric

4.1 Non-modifying Sequence Algorithms

On a container (sequence), you may need to perform different functions that don't need to modify the contents of the container on which they work. *For example:* If you want to search an element in a container then it does not require you to modify the contents of the container. STL defines the following non-modifying algorithms.

They all require the `#include<algorithm>` file.

Function	Description
adjacent_find()	Searches for adjacent matching elements within a sequence. It returns an iterator to the first match.
count()	Counts the number of elements in a sequence.
count_if()	Returns the number of elements in the sequence that matches a predicate.
equal()	Returns TRUE if two ranges are same.
equal_range()	Returns a range in which an element can be inserted into a sequence without changing the order of the sequence.
find()	Returns the first occurrences of a specified value in a sequence.
find_end()	Returns the last occurrences of a specified value in a sequence.
find_first_of	Returns the first element within a sequence that matches an element within a range.
find_if()	Finds the first match of a predicate in a sequence.
for_each()	Performs the operation for each elements in the range.

A priority queue constructor creates an empty priority queue. By default it uses vector as a container but you can also use a deque as a container for priority queue.

To be able to use STL priority queues add this before you start using them in your source code:
`#include<queue>`

Member Function

Function	Description
empty	true if the priority queue has no elements
pop	removes the top element of a priority queue
push	adds an element to the end of the priority queue
size	returns the number of items in the priority queue
top	returns the top element of the priority queue

Following program shows how to create priority queue and manipulate it with different functions:

```
#include<iostream>
#include<list>
#include<queue>
using namespace std;
int main()
{
    priority_queue<float, vector<float> > q;
    // insert six elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
   // pop and print the elements
   cout<< "Elements removed from the priority queue:";
   int size=q.size();
   for(int i=0; i<size;++i)
   {  cout << q.top() << ' ';
      q.pop();
   }
    return 0;
}
```

Output

Elements removed from the priority queue:

66.6 55.5 44.4 33.3 22.2 11.1

In the above program, we define the priority queue using the default less<> predicate, therefore the values removed from the sequence in order from largest to smallest. If we define the priority queue using the greater<> predicate then the highest priority value is the smallest because the values are arranged in the priority queue in ascending order. To use the greater<>predicate just replace the line

```
priority_queue<float, vector<float> > q;
```

with

```
priority_queue<float, vector<float>, greater<float>  > q;
```

and the output will be

Elements removed from the priority queue:

```
11.1 22.2 33.3 44.4 55.5 66.6
```

4. Algorithms

The STL algorithms are template C++ functions used to perform operations on containers. Although each container provides functions for its own basic operations, the standard algorithms provide more extended or complex actions. Also we can work with two different types of containers at the same time by using algorithms. By using `<algorithm>` header file in our program we can access the STL algorithms.

In order to be able to work with many different types of containers, the algorithms do not take containers as arguments. Instead, they take iterators that specify part or all of a container. In this way the algorithms can be used to work on entities that are not containers; *for example*: the function copy can be used to copy data from standard input into a vector.

The generic algorithms fall into following four categories:

Categories of Generic Algorithms

i. Non-modifying Sequence
ii. Mutating Sequence
iii. Sorting
iv. Numeric

4.1 Non-modifying Sequence Algorithms

On a container (sequence), you may need to perform different functions that don't need to modify the contents of the container on which they work. *For example:* If you want to search an element in a container then it does not require you to modify the contents of the container. STL defines the following non-modifying algorithms.

They all require the `#include<algorithm>` file.

Function	Description
adjacent_find()	Searches for adjacent matching elements within a sequence. It returns an iterator to the first match.
count()	Counts the number of elements in a sequence.
count_if()	Returns the number of elements in the sequence that matches a predicate.
equal()	Returns TRUE if two ranges are same.
equal_range()	Returns a range in which an element can be inserted into a sequence without changing the order of the sequence.
find()	Returns the first occurrences of a specified value in a sequence.
find_end()	Returns the last occurrences of a specified value in a sequence.
find_first_of	Returns the first element within a sequence that matches an element within a range.
find_if()	Finds the first match of a predicate in a sequence.
for_each()	Performs the operation for each elements in the range.

mismatch()	Finds first mismatch between the elements in two sequences. Iterators to two elements are returned.
search()	Searches for subsequence within a sequence.
search_n()	Searches for a sequence of a specified number of similar elements.

4.2 Mutating Sequence Algorithm

Some types of a sequence operations results in modifying the contents of a container on which they work. *For example*: If you want to copy one part of a sequence into another part of the same sequence so in such case the mutating sequence algorithm provide a copy function. STL defines the following mutating algorithm. They all require the `#include<algorithm>` file:

Function	Description
copy()	Copies a sequence.
copy_backward()	This functions copies the elements starting from the end of the sequence, i.e., it works backward to first.
fill()	Fills a sequence with a specified value.
fill_n()	Fills first n elements with a specified value.
generate()	Assigns the value returned by a generator function to all elements in a sequence.
generate_n()	Assigns the value returned by the generator function to first n element in a sequence.
iter_swap()	Exchanges the elements specified by the two iterators
random_shuffle()	Places elements in random order.
remove()	Removes elements of a specified value.
remove_copy()	After removing a specified value copies a sequence
remove_copy_if()	After removing elements that matches a predicate copies a sequence.
remove_if()	Removes the elements that match the predicate.
replace()	Replaces elements with a specified value.
replace_if()	Replaces elements matching a predicate.
replace_copy()	After replacing elements with a given value copies a sequence.
replace_copy_if()	After replacing elements matching a predicate copies a sequence.
reverse()	Reverses the order of elements
reverse_copy()	Sequence is copied in reverse order.
rotate()	Left-rotates the elements in a sequence
rotate_copy()	Copies a sequence after rotation.
swap()	Exchanges two elements
swap_ranges()	Exchanges elements in a sequence.
transform()	Applies the function to each element in a sequence and stores the result in a new sequence.
unique()	Removes duplicate elements form a sequence.
unique_copy()	After removing duplicate elements copies a sequence.

4.3 Sorting Algorithm

STL provides various functions for sorting the contents of a container (random-access container only). Following table lists the sorting and sorting related functions and their descriptions. Sorting algorithms require the #include<algorithm> file:

Function	Description
binary_search()	Performs a binary search on an ordered sequence.
equal_range()	Finds a subrange of elements with a given value.
includes()	Returns TRUE if one sequence includes all elements of another sequence.
inplace_merge()	Merges two sequences sorted in increasing order and the resulting sequence is also sorted.
lexicographical_compare()	Compares one sequence with another alphabetically.
lower_bound()	Finds the first position in a sequence that is not less than the specified value.
make_heap()	Creates a heap from a contents of a sequence.
max()	Returns the maximum of two values.
max_element()	Returns the maximum element within a sequence.
merge()	Merges two sorted sequences and place the result into a third sequence.
min()	Returns the minimum of two values.
min_element()	Returns the minimum elements within a sequence.
nth_element()	Arranges a sequence such that all elements to the left of the nth element are less than or equal to all elements to the right of the nth element.
next_permutation()	Constructs next permutation of a sequence.
prev_permutation()	Constructs a previous permutation of a sequence.
partial_sort()	Sorts a part of a sequence.
partial_sort_copy()	Sorts a part of a sequence and copies as many elements as will fit into a resulting sequence.
partition()	Arranges a sequence such that all elements that matches with a predicate are placed first and then all elements which are not matches with a predicate are placed.
pop_heap()	Removes the top element.
push_heap()	Adds an element on to the end of a heap.
sort()	Sorts a sequence.
sort_heap()	Sorts a heap within a specified range.
stable_partiotin()	Arranges a sequence such that all elements that matches with a predicate are placed first before the elements which are not matches with a predicate. The partitioning is stable that means the relative ordering of the sequence is preserved.
stable_sort()	Sorts a sequence. The sort is stable. This means that equal elements are not rearranged.
upper_bound()	Searches a sequence of sorted elements for the correct last position for the specified value and returns that position.
set_difference()	Creates a sequence that contains the difference between two ordered sets.

set_intersection()	Creates a sequence that contains the intersection of the two ordered sets.
set_symmetric_difference()	Creates a sequence that contains the symmetric difference between the two ordered sets.
set_union()	Creates a sequence that contains the union of the two ordered sets.

4.4 Numeric Algorithms

Numeric algorithms are used to perform four types of numeric calculations on the contents of a sequence. Following table describes these four functions along with their descriptions. They all require the `#include <numeric>` file:

Function	Description
accumulate()	Accumulates the results of operations on a sequence.
adjacent_difference()	Calculates the adjacent difference of the elements.
inner_product()	Accumulates the results of operation on a pair of sequences.
partial_sum()	Calculates the partial sum of the elements.

5. Iterators

Iterators are objects that acts like an pointer, i.e., they specify the location for containers or streams of data. *For example*: int* can be used as a location specifier for an array of integers, or an ifstream can be used as a location specifier for a file. STL provides a variety of iterators for its different collection types and for streams.

Iterators are used throughout the STL to access and list elements in a container. They are often used to traverse from one element to other, a process known as iterating through the container.

Iterator Categories

PU 1
Oct. 2011 – *5M*
★ What is the need for iterators? What is their role in STL?

There are five categories of iterators in STL and the Standard C++ Library which are as follows:

i. **Input Iterators:** Input Iterators are (along with Output Iterators) the least powerful of all iterator types; as such they are supported by all STL containers. A program can use an input iterator only to read the contents of a container. In order to traverse a sequence, an input iterator can be incremented (but not decremented) and compared for equality or inequality using the == and != operators.

ii. **Output Iterators:** A program can use an Output Iterators to write to the contents of a container. In order to traverse a sequence, an Output Iterator can be incremented (but not decremented).

iii. **Forward Iterators:** Forward Iterators combine Input Iterators and Output Iterators. They can be used to traverse containers in one direction (i.e., forward), for reading and/or writing. However, a program can save the value of a forward iterator in order to restart traversing a container from the iterator's original position.

iv. **Bidirectional Iterators:** Bidirectional Iterators can iterate in a forward or reverse direction over the contents of a container, both reading and writing as required by the program. In this way, a bi-directional iterator combines the capabilities of a forward iterator with the capability to traverse a sequence in reverse. Bidirectional supports all operations of forward iterators.

v. **Random-access Iterators:** The most powerful of the iterator types, random-access iterators have all the functionality of bi-directional iterators with the ability to use pointer arithmetic and all pointer comparisons.

Following *table 13.2* summarises the above iterators:

Table 13.2

Iterator	Description	Supported Operations
Input Iterator	This iterator is read-only. That is it cannot be assigned to.	=, ==, !=, *, ->, ++, No assignment of *i
Output Iterator	This iterator is write-only. It cannot be read.	* ,=, and ++
Forward Iterator	This iterator is Read/Write iterator.	=, ==, !=, *, ->, and ++
Bidirectional Iterator	This iterator is Read/Write iterator	=, ==, !=, *, ->, ++, and --
Random Access Iterator	This iterator is Read/Write iterator	=, ==, !=, +=, -=, *, ->, +, ++, -, -- , [n], <, <=, >, and >=

Each category adds new features to the previous one. The iterator categories obey the following order:

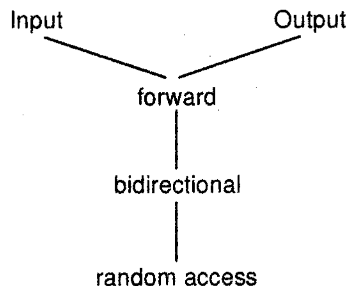


Figure 13.5

We can use the iterator having greater access capability in place of one that has lesser capability. *For example:* A bidirectional iterator can be used in place of forward iterator.

Iterators are declared using the iterator type defined by various containers. *Table 13.2* shows the type of iterator supported by each container.

In addition to the above type STL also support reverse iterators which are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

6. Function Objects

Function objects or function are classes that have the function operator, (), overloaded. Many of the algorithms have versions that take a function or function object as an parameter.

These function or function objects usually take one or two parameters and sometimes return a Boolean, i.e., true/false value or sometimes processes the objects that the algorithm finds and returns an object of the type in the container.

A function objects come in two varieties:

i. **Unary Function:** A function object that takes a single argument, i.e., one that is called as f(x).

ii. **Binary Function**: A function object that takes two arguments, i.e., one that is called as f(x, y)).

For using function object, we must include `<functional>` header.

Following table 13.3 summarizes the STL function objects:

Table 13.3

Function object	Description	Type
Arithmetic		
plus<T>	Takes two arguments of the same type and returns their sum, i.e., arg1+arg2	Binary
minus<T>	Returns the result of subtracting argument two from argument one, i.e., arg1–arg2	Binary
multiplies<T>	Takes two arguments of the same type and returns their product, i.e., arg1*arg2	Binary
divides<T>	Divides argument one by argument two, i.e., arg1/arg2	Binary
modulus<T>	Returns the modules of the two arguments, i.e., arg1%arg2.	Binary
negate<T>	Returns the negative of a single argument, i.e., –arg1.	Unary
Relational		
equal_to<T>	Takes two arguments and returns true if arg1 == arg2.	Binary
not_equal_to<T>	Takes two arguments and returns true if arg1 != arg2.	Binary
greater<T>	Takes two arguments and returns true if arg1 > arg2.	Binary
less<T>	Takes two arguments and returns true if arg1 < arg2.	Binary
greater_equal<T>	Takes two arguments and returns true if arg1 >= arg2.	Binary
less_equal<T>	Takes two arguments and returns true if arg1 <= arg2.	Binary
Logical		
logical_and<T>	Takes two arguments and returns true if arg1 && arg2.	Binary
logical_or<T>	Takes two arguments and returns true if arg1 || arg2.	Binary
logical_not<T>	Takes one argument and returns true if not, i.e., !arg1.	Unary

Note: Here the arg1 and arg2 are the objects of class T passed to the function object as arguments.

Predicate

Many algorithms and containers use a special kind of function called a predicate. A predicate is a function that returns a boolean value. There are two variations of predicates: Unary and Binary. A unary predicate takes only one argument whereas a binary predicate takes two arguments. These functions return true/false result. In case of a binary predicate the arguments are always in the order of first, second. For both unary and binary predicates the arguments will contain values of the type of objects being stored by the container.

Let's see how a predicate can be used to extend the functionality of the sort algorithm. We have already used less<> predicate while illustrating the example of priority queue. In the following

example let us see how the predicate greater<int> can be used to override the default ascending order. Likewise, the predicate less<int> restores the original ascending order:

```
#include<functional> //definitions of STL predicates
#include<algorithm>  //definition of sort
#include<vector>
#include<iostream>
using namespace std;
int main()
{
 vector<int> vi;
 vi.push_back(9);
 vi.push_back(5);
 vi.push_back(10);
 sort(vi.begin(), vi.end(), greater<int> () ); // descending order
 cout<< vi[0] << '\t' << vi[1] << '\t' << vi[2] <<endl; // output: 10 9 5
 sort(vi.begin(), vi.end(), less<int> () );  // now in ascending order
 cout<< vi[0] << '\t' << vi[1] << '\t' << vi[2] <<endl; // output: 5 9 10
 return 0;
 }
```

7. Allocators

Every STL container class defines an allocator class that manages the allocation of memory for the container or encapsulates the memory model that the program uses. Allocators hide the platform-dependent details such as the size of pointers, memory organization, reallocation model, and memory page size. Because a container can work with different allocator types, it can easily work in different environments simply by plugging a different allocator into it. An implementation provides a suitable allocator for every container. Normally, users should not override the default allocator.

The default allocator is an object for a class allocator and STL provides a default allocator object for each container, so you should not need to deal directly with the allocator class. You can also define your own allocator if needed by specialized applications.

For most uses, the default allocator is sufficient.

Note: Whenever a container inserts or removes an element, it uses its allocator to allocate and deallocate the memory for the object. The container does not know anything about the memory model of the machine, it relies on the allocator for all of its memory needs.

8. Adaptors

Sometimes you have a class that does the right thing, but has the wrong interface for your purposes. Adaptors are classes that sit between you and another class, and translate the messages you want to send into the messages the other class wants to receive. In short, an *adaptor* is a component that modifies the interface of another component.

For example: The copy function expects an input iterator to get its data from. The istream class has the right functionality: it acts as a source of data, but it has the wrong interface: it uses << etc.

STL uses several types of adaptors:

i. **Sequence Adaptor:** It is a container that's built on another container and that modifies its interface. *For example*: The container stack is usually implemented as a deque, whose non-stack operations are hidden. In addition, stack uses the operations back(), push_back(), and pop_back() of a deque to implement the operations top(), push(), and pop(), respectively.

For example

```
#include<string>
#include<stack>
#include<iostream>
using namespace std;
int main()
{
  stack<string> strstack;
  strstack.push("Mrunal");
  strstack.push("Harischandre");
  string topmost = strstack.top();
  cout<< "topmost element is: "<< topmost << endl; // "Harischandre"
  strstack.pop();
  cout<< "topmost element is: "<< strstack.top() << endl; //"Mrunal"
  return 0;
}
```

Calling the member function pop() on an empty stack is an error. If you are not sure whether a stack contains any elements, you can use the member function empty() to check it first.

For example

```
stack<int> stk;
//...many lines of code
if(!stk.empty()) //test stack before popping it
{
    stk.pop();
}
```

ii. **Iterator Adaptors:** The interface of an iterator can be altered by an *iterator adaptor*. The member functions rend() and rbegin() return reverse iterators, which are iterators that have the meanings of operators ++ and -- exchanged. Using a reverse iterator is more convenient in some computations.

iii. **Function Adaptors:** It modifies the interface of a function or a function object. Earlier you saw the use of greater as a function adaptor for changing the computation of sort().

iv. **Negators:** Negators are used to reverse the result of certain Boolean operations.

v. **Binders:** Binders convert a binary function object into a unary function object by binding an argument to a specific value.

Solved Programs

1. Write a program using vector class.

Solution

```
// This example shows how to create a vector, initialize it with values,
//sort and print it
#include<vector>
#include<algorithm>
#include<iostream>
using namespace std;
int main()
{   // create a vector of 4 ints
  vector<int> v(4);
  // initialize the vector with values
  v[0]=2;
  v[1]=0;
  v[2]=3;
  v[3]=1;
  // sort the vector
  sort(v.begin(),v.end());
 // print the sorted vector
 for(vector<int>::iterator viter=v.begin();viter!=v.end();viter++)
 cout << *viter << " ";
 cout << endl;
  return 0;
}
```

2. Write a program using the list class.

Solution

```
// This example shows how to create a list, initialize it with values,
//find a particular value and print the element before and after that
//value
#include<list>
#include<iostream>
#include<string>
using namespace std;
int main()
{ // create a list
  list<string> l;
  // add some values to the list
  l.insert(l.end(),"Mrunal");
  l.insert(l.end(),"Ganesh");
  l.insert(l.end(),"Pooja");
  l.insert(l.end(),"Gaurav");
 // find the value "Mrunal"
 string value("Mrunal");
 list<string>::iterator liter=find(l.begin(),l.end(), value);
 // print out the value before and after "Stroustrup"
 if(liter!=l.end())
   cout << "Mrunal was found after " << *(--liter)
        << " and before " << *(++(++liter)) << endl;
 return 0;
}
```

3. Write a program using Stack.

Solution

```
#include<stack>
#include<vector>
#include<deque>
#include<string>
#include<iostream>
using namespace std;
int main(void)
{// Make a stack using a vector container
  stack<int,vector<int>, allocator> s;
 // Push a couple of values on the stack
  s.push(1);
  s.push(2);
  cout << s.top() << endl;
  // Now pop them off
  s.pop();
  cout << s.top() << endl;
  s.pop();
  // Make a stack of strings using a deque
  stack<string,deque<string>, allocator> ss;
  // Push a bunch of strings on then pop them off
  int i;
  for(i = 0; i < 10; i++)
  {  ss.push(string(i+1,'a'));
     cout << ss.top() << endl;
  }
  for(i = 0; i < 10; i++)
  { cout<< ss.top() << endl;
     ss.pop(); }
  return 0;
}
```

4. Write a program using Priority Queue.

Solution

```
#include<queue>
#include<deque>
#include<vector>
#include<string>
#include<iostream>
using namespace std;
int main(void)
{ // Make a priority queue of int using a vector container
  priority_queue<int, vector<int>, less<int>, allocator> pq;
  // Push a couple of values
  pq.push(1);
  pq.push(2);
  // Pop a couple of values and examine the ends
  cout << pq.top() << endl;
  pq.pop();
  cout << pq.top() << endl;
  pq.pop();
  // Make a priority queue of strings using a deque container
```

```
priority_queue<string, deque<string>, less<string>, allocator> pqs;
 // Push on a few strings then pop them back off
 int i;
 for(i = 0; i < 10; i++)
{ pqs.push(string(i+1,'a'));
  cout << pqs.top() << endl;  }
for(i = 0; i < 10; i++)
{cout << pqs.top() << endl;
 pqs.pop();   }
// Make a priority queue of strings using a deque container and greater
// as the compare operation
 priority_queue<string,deque<string>, greater<string>, allocator> pgqs;
// Push on a few strings then pop them back off
for(i = 0; i < 10; i++)
{pgqs.push(string(i+1,'a'));
 cout << pgqs.top() << endl;
}
for(i = 0; i < 10; i++)
{cout << pgqs.top() << endl;
 pgqs.pop(); }
return 0;
}
```

EXERCISES

A. Review Questions

1. What is STL?
2. What is Container? List the three types of containers.
3. What is the major difference between sequence container and associative container?
4. What is an algorithm?
5. How are STL algorithms implemented?
6. What are the different types of algorithms?
7. What is an iterator? What are its characteristics?

B. Programming Exercise

1. Write a code segment that does the following:
   i. Define a vector v1 with a maximum size of 10
   ii. Sets the first element of v1 to 1
   iii. Sets the last element of v1 to 10
   iv. Sets the other elements to 1
   v. Displays the contents of v1.
2. Write a program using count() algorithm to count the elements in a container having specified value.
3. Write a program using find() algorithm to locate the position of a specified value in a sequence container.

Collection of Questions asked in Previous Exams PU

1. Write short note on Standard Template Library. **[Oct. 2009, Apr. 2010, Oct. 2010 – *5M*]**
2. What is the need for iterators? What is their role in STL? **[Oct. 2011 – *5M*]**

14 Namespace

I. Introduction

Namespaces are a relatively new C++ feature just now starting to appear in C++ compilers. Namespaces allow us to group a set of global classes, objects and/or functions under a name. To say it another way, they serve to split the global scope in sub-scopes known as *namespaces*. The best example of namespace scope is the C++ Standard Library.

All classes, functions and templates are declared within the namespace named *std*. That is why we have been using the directive,

```
using namespace std;
```

in our program that uses the standard library. The using namespace statement specifies that the members defined in *std* namespace will be used frequently throughout the program.

What's a namespace, really?

It is a feature in C++ used to minimize name collisions (of variables, types, classes or functions) without some of the restrictions imposed by the use of classes, and without the inconvenience of handling nested classes in the global name space.

This *namespace* keyword assigns a distinct name to a library that allows other libraries to use the same identifier names without creating any name collisions (i.e., two things with the same name).

Furthermore, the compiler uses the namespace signature for differentiating the definitions. The larger the program the more useful this idea is, especially if you use libraries written by other people.

2. Defining a Namespace

Defining a namespace is similar to defining a class. *First* goes the namespace keyword, followed by the identifier (the namespace name), followed by member declarations enclosed in braces.

Syntax

```
namespace identifier
{ // declared or defined entities
  // (declarative region) }
```

where, the namespace keyword is used in order to uniquely identify a namespace, *identifier* is any valid identifier and within the *declarative region*, functions, variables, structs, classes and even (nested) namespaces can be defined or declared.

For example

```
namespace direct
{ class Arrow
  {   public:
     Arrow(int dir);
      void setDirection(int dir);
      private:
           int direction;  }
  // ...... other stuff    }
```

Namespaces cannot be defined within a block. So it is not possible to define a namespace within, *for example*: a function. A namespace, however, cannot have access specifiers, such as public: or private:. All members of a namespace are public. It cannot have a trailing semicolon, either. An important difference between classes and namespaces, is that class definitions are said to be closed, meaning that, once defined, new members cannot be added to it. A namespace definition is open, and can be split over several units. *For example*

```
// file SY.h
namespace SY { class Maker { ... };
               class SuperMaker : public Maker { ... }; }
// file data.h
namespace SY {  class Binder { ... };
                class DataBinder : public Binder { ... };}
```

In this example, there are two files (SY.h and data.h), both defining namespace SY. The definition of SY on data.h does not conflict with the one in SY.h, but actually *extends* it. If you look closely at the Standard Library, you'll notice that no single header file declares all members of namespace std. Each file only declares some members, adding them to the global std namespace.

2.1 Referring to Members of a Namespace

Given a namespace and members that are defined or declared in it, the scope resolution operator can be used to refer to the members that are defined in that namespace.

Syntax

```
namespace_name::member
```

For example:

```
namespace SY
{ void j() };
SY::j();
```

In this example, the scope resolution operator provides access to the function `j` held within namespace SY. The scope resolution operator `::` is used to access identifiers in both global and local namespaces. Any identifier in an application can be accessed with sufficient qualification. This is a rather cumbersome way to refer to the `j()` function in the SY namespace, especially so if the function is frequently used.

However in such cases, we can use a using statement to simplify their access. The using statement has two forms: using directive and using declaration.

i. Using Directive: A using directive provides access to all namespace qualifiers and the scope operator. This is accomplished by applying the **using** keyword to a namespace identifier.
Syntax

```
using namespace name;
```

The *name* specifies the name of the namespace you want to access. All of the 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. *For example*

```
include<iostream>
namespace first { int var = 5; }
namespace second { double var = 3.1416; }
int main ()
{ using namespace second;//all members are visible of second namespace
  cout << var << endl;
  cout << (var*2) << endl;
  return 0;
 }
```

Output

3.1416
6.2832

In this case we have been able to use var without having to precede with any scope operator.

ii. Using Declaration: A *using declaration* provides access to a specific namespace member. This is accomplished by applying the using keyword to a namespace name with its corresponding namespace member.

Syntax

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

For the declaration to work, the member must be declared inside the given namespace. *For example*

```
namespace A {int i;
  int k;
  void f;
  void g;  }
 using A::k   // only a specific member (i.e., K) is visible
```

In this example, the using declaration is followed by A, it is the name of namespace, which is then followed by the scope operator (: :), and k. This format allows k to be accessed outside of namespace A through a using declaration. After issuing a using declaration, any extension made to that specific namespace will not be known at the point at which the using declaration occurs.

2.2 Difference between using declaration and using directive

[Oct. 2011 - *5M*]

A namespace is a scope in which declarations and definitions are grouped together. In order to refer to any of these from another scope, a full qualified name is required. However, repeating the full qualified name over and over again is tedious, error prone and less readable. Instead, a *using declaration* or a *using directive* can be used.

	Declaration	Directive
i.	A using declaration is a sequence consisting of the keyword using followed by a namespace:: member.	A using directive is a sequence consisting of the keyword using.
ii.	It instructs the compiler to locate every occurrence of a certain declaration (type, operator, function, etc.) in the specified namespace, as if the full qualified name were supplied.	A using directive instructs the compiler to recognize all members of a namespace and not just one.
iii.	General **Syntax**: using `namespacename::memberofnamespace;`	General **Syntax**: using `namespacename;`

iv.	`#include <vector> //STL vector; belongs to namespace std`
	`void main()`
	`{`
	`using std::vector; //using declaration; every occurrence of vector is looked up in std`
	`vector <int> vi; //without a using declaration, a full qualified name: std::vector<int> would be required`
	`//...`
	`}//end of main; the above using declaration goes out of scope here`
		`#include <vector> //STL vector; belong to namespace std`
		`#include <iostream> //iostream classes and operators are also in namespace std`
		`void main()`
		`{`
		`using namespace std;`
		`//directive; all <iostream> and <vector> declarations now accessible`
		`vector <int> vi;`
		`vi.push_back(10);`
		`cout<<vi.begin();`
		`}`

3. The standard namespace

Many entities of the runtime available software (*For example*: cout, cin, cerr, string and templates defined in Standard Template Library) are now defined in std namespace. There are various ways to access things inside std namespace which are:

i. Explicit access: Explicitly mention the namespace each time you use a facility.

```
// Option 1
#include<iostream>
int main() {  std::cout << "Hello, world!" << std::endl;
  }
```

Here, cout and the manipulator endl are explicitly qualified by their namespace. That is to write to standard output, you must specify std::cout and the endl manipulator must be referred to as std::endl. But if your program uses hundreds of references to library names, then qualifying each name individually is not possible.

ii. **Using statement:** If you are using only a few names from the standard library, it may make more sense to specify a using statement for each individually. The advantage of this approach is that you can still use those names without std:: qualification, but you will not be bringing the entire standard library into the global namespace.

For example: Consider the following program:

```
// Option 2
#include<iostream>
using std::cout;
using std::endl;
int main()
{  cout << "Hello, world!" << endl;
  }
```

Here, cout and endl may be used directly, but the rest of the std namespace has not been brought into view.

iii. **The using namespace statement:** You can use "using namespace std" statement in the source code file that references the namespace and this will make the standard library facilities available throughout the program. *For example*

```
// Option 3
#include<iostream>
using namespace std;
int main()
{    cout << "Hello, world!" << endl;
  }
```

In the above program, we use the "using namespace std" statement in the source code because of that the std namespace is brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with std::. But we can use this approach only for smaller teaching programs. However, it does make globally visible a lot of function and variable names that you're

probably not going to use. *For example*: We use a graphics include file that defines a value RED. Unfortunately RED is also defined in the std namespace, thus causing a potential name-clash when a global "using namespace std" is used. For big programs it's probably better not to use this option.

Note that the original C++ library was defined in the global namespace. If you will be converting older C++ programs, then you will need to either include a using namespace std statement or qualify each reference to a library member with std::. This is especially important if you are replacing old .h header files with the new-style headers. Remember, the old .h headers put their contents into the std namespace.

4. Nested Namespace

PU Oct.09, Oct.10 – 5M
★ What is Namespace? Explain the nested namespaces with an example.

We can define one namespace into another. This is called nesting of namespace. Consider the following example:

```
namespace first
    {   namespace Second
        {  void *pointer;        }      }
```

Now the variable *pointer* is defined in the *Second* namespace, nested under the *First* namespace. In order to refer to this variable (i.e., *pointer*), the following options are available:

i. The fully qualified name can be used. A fully qualified name of an entity is a list of all the namespaces that are visited until the definition of the entity is reached, glued together by the scope resolution operator:

```
int main()
{  First::Second::pointer = 0;   }
```

ii. A *using* declaration for *First::Second* can be used. Now *Second* can be used without any prefix, but *pointer* must be used with the *Second*:: prefix:

```
...
using First::Second;
int main()
{   Second::pointer = 0;   }
```

iii. A *using* declaration for *First::Second::pointer* can be used. Now *pointer* can be used without any prefix:

```
...
using First::Second::pointer;
int main()
{  pointer = 0;   }
```

iv. A using directive can be used:

```
...
using namespace First::Second;
int main()
{  pointer = 0;   }
```

v. Alternatively, two separate using directives could have been used:

```
...
using namespace First;
using namespace Second;
int main()
{  pointer = 0;   }
```

vi. A combination of *using* declarations and *using* directives can be used. *For example:* A *using* directive can be used for the *First* namespace, and a `using` declaration can be used for the *Second::pointer* variable:

```
...
using namespace First;
using Second::pointer;
int main()
{  pointer = 0;  }
```

At every *using* directive all entities of that namespace can be used without any further prefix. If a namespace is nested, then that namespace can also be used without any further prefix. However, the entities defined in the nested namespace still need the nested namespace's name. Only by using a using declaration or directive the qualified name of the nested namespace can be omitted. Consider the following program illustrating the working of nested namespace:

```
#include<iostream>
using namespace std;
namespace Name1
{ double i;
  namespace Name2
  {      //Nesting namespace
    double j; } }
int main()
{  Name1::i=20.5
  Name2::j=30.5    // gives error, Name2 is not in view
  Name1::Name2::j=30.5;  // Using fully qualified name
  cout << "i= " << Name1::i<< "\n "
  cout << "j= " << Name1::Name2::j<< "\n ";
  using namespace Name1;
  //Bring Name1 in view, so Name2 can be used to refer to j
  cout << "i= " << i << "\n;
  cout << "j= " << Name2::j <<  "\n ";
  return 0;
 }
```

Output

i=20.5

j=30.5

i=20.5

j=30.5

5. Unnamed Namespace

A namespace with no identifier before an opening brace produces an *unnamed namespace* or *anonymous namespace*. Unnamed namespace allows you to create unique identifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace may be used directly, without qualification. But outside the file, the identifiers are unknown. Here's the format.

> **1** PU
> **Apr. 2010 – 5M**
> * What is Namespace? Explain unnamed namespace using a suitable example.

```
namespace
{  namespace_body;}
```

A left brace immediately follows the keyword *namespace* without an intervening name qualifier. All members defined in *namespace_body* are in an unnamed namespace that is guaranteed to be unique for each file. At link time, member names in one unnamed namespace do not conflict with equivalent member names from other unnamed namespaces in separate file. Following example demonstrates how unnamed namespaces are useful:

```
#include<iostream>
using namespace std;
namespace
{ //unnamed namespace
  const int i = 4;
  int variable; }
int main()
{ cout << i << endl;        // i in unnamed namespace
  variable = 100;           // variable in unnamed namespace
  return 0;
 }
```

In the previous example, the unnamed namespace permits access to *i* and *variable* without using a scope resolution operator. The following example illustrates an improper use of unnamed namespaces:

```
#include<iostream>
using namespace std;
namespace
{ const int i = 4; }
int i = 2;
int main()
{    cout << i << endl; // error
     return 0;
 }
```

Inside *main, i* causes an error because the compiler cannot distinguish between the global name and the unnamed namespace member with the same name. In order for the previous example to work, the namespace must be uniquely identified with an identifier and *i* must specify the namespace it is using. You can extend an unnamed namespace within the same file. *For example*

```
#include<iostream>
using namespace std;
namespace {      //unnamed namespace
  int variable;
  void funct(int);    //prototype }
namespace {    // same unnamed namespace
  void funct(int i) { cout << i << endl; } }
int main()
{    funct(variable);
   return 0;
}
```

Both the prototype and definition for *funct* are members of the same unnamed namespace. *Note:* Items defined in an unnamed namespace have internal linkage. Rather than using the keyword *static* to define items with internal linkage, define them in an unnamed namespace instead.

6. Namespace Alias

We can declare alternate names for existing namespaces according to the following format:

Syntax

```
namespace alias_name = namespace_name;
```

Follow the keyword *namespace* with your alias, i.e., (alias_name). *namespace_name* must be a name of the previously defined namespace. You can use more than one alias for the same namespace qualifier, but you can't alias an existing alias. Aliases apply only to the translation unit or source code file where they appear; the linker sees the original name. Namespace aliases are convenient shorthand names and used when a namespace has a long name. *For example*

```
namespace INTERNATIONAL_BUSINESS_MACHINES
{  void f(); }
 namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

In this example, the *IBM* identifier is an alias for *INTERNATIONAL_BUSINESS_MACHINES*. Namespace aliases may appear in using directives, using declarations, and qualified namespace members.

Creating an Alias for a Nested Namespace

An alias can also be applied to a nested namespace. *For example*

```
namespace X {  namespace Y  {  class Z { ... };  }  }
```

The full qualifier for Z is X::Y::Z, but we can declare an alias using

```
namespace w = X::Y;
```

This declares w as an alias for namespace X::Y, thus we can access Z using `w::z`.

EXERCISES

A. Review Questions

1. How do we access the variables declared in a named namespace?
2. What is a namespace conflict? How is it handled in C++?
3. What is meant by unnamed namespace? Give the example illustrating the use of unnamed namespace.
4. How can you create alias for nested namespace?

B. Programming Exercises

1. Define a namespace named Constants that contains declarations of some constants. Write a program that uses the constants defined in the namespace Constants.
2. Identify the error in the following program.

```
#include<iostream.h>
namespace A  {  int i;
   void display()
   { cout << i;}}
void main()
{     namespace Inside { int insideI;
      void dispInsideI() {
   cout << insideI;  }  }
   A::i=10;
   cout << A::I;
   A::dispI();
   Inside::insideI=20;
   cout<<Inside::insideI;
   Inside::dispInsideI();  }
```

Collection of Questions asked in Previous Exams PU

1. What is Namespace? Explain the nested namespaces with an example. **[Oct. 2009 – 5M]**
2. What is Namespace? Explain unnamed namespace using a suitable example. **[Apr. 2010 – 5M]**
3. What is Namespace? Explain nested namespace using a suitable example. **[Oct. 2010 – 5M]**
4. What is the difference between using declaration and using directive? **[Oct. 2011 – 5M]**

15 New Style Casts And RTTI

I. Introduction

To support modern, object-oriented programming Standard C++ contains two features: Run-Time Type Identification (RTTI) and a New-Style Casts (a set of four additional casting operators: dynamic_cast, const_cast, static_cast and reinterpret_cast). These two features were not the part of the original specification for C++ but both were added later to provide enhanced support for run-time polymorphism. RTTI allows us to identify the type of an object during the execution of the program and the casting operators give us a safer and more controlled ways to cast.

Why Cast?

Casts are used to convert the type of an object, expression, function argument, or return value to that of another type. Some conversions are performed automatically by the compiler without intervention by the programmer. These conversions are called *implicit conversions*. The standard C++ conversions and user-defined conversions are performed implicitly by the compiler where needed. Other conversions which must be explicitly specified by the programmer and are appropriately called *explicit conversions*. When a type is needed for an expression that cannot be obtained through an implicit conversion or when more than one standard conversion creates an ambiguous situation, the programmer must explicitly specify the target type of the conversion.

In C, an expression, expr, of type S can be cast to another type T in one of the following ways. By using an explicit cast:

```
(T)expr
```

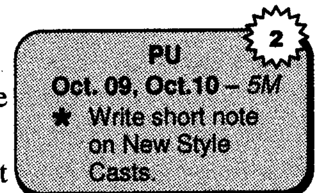
or by using a functional form:

```
T(expr)
```

We will refer to either of these constructs as the *old C-style casts*.

The old C-style casts have several shortcomings. First, the syntax is the same for every casting operation. This means it is impossible for the compiler (or users) to tell the intended purpose of the cast. Is it a cast from a base class pointer to a derived class pointer? Does the cast remove the "const-ness" of the object? Or, is it a conversion of one type to a completely unrelated type? The truth is, it is impossible to tell from the syntax. As a result, this makes the cast harder to comprehend, not only by humans, but also by compilers which are unable to detect improper casts. Another problem is that the C-style casts are hard to find. Parenthesis with an identifier between them are used all over C++ programs. There is no easy way to "grep" a source file and get a list of all the casts being performed. Perhaps the most serious problem with the old C-style cast is that it allows you to cast practically any type to any other type. Improper use of casts can lead to disastrous results. The old C-style casts have created a few holes in the C type system and have also been a source of confusion for both programmers and compilers. Even in C++, the old C-style casts are retained for backwards compatibility. However, using the new C++ style casting operators will make your programs more readable, less error-prone and type-safe, and easier to maintain.

2. New-Style Casts



The new C++ casting operators are intended to provide a solution to the shortcomings of the old C-style casts by providing:

i. **Improved syntax:** Casts have a clear, concise, although somewhat cumbersome syntax. This makes casts easier to understand, find, and maintain.

ii. **Improved semantics:** The intended meaning of a cast is no longer ambiguous. Knowing what the programmer intended the cast to do makes it possible for compilers to detect improper casting operations.

iii. **Type-safe conversions** allow some casts to be performed safely at run-time. This will enable programmers to check whether a particular cast is successful or not.

C++ introduces four new casting operators:

a. static_cast, to convert one type to another type;

b. dynamic_cast, for *safe* navigation of an inheritance hierarchy; and

c. const_cast, to cast away the "const-ness" or "volatile-ness" of a type;

d. reinterpret_cast, to perform type conversions on un-related types.

Note: Use **const_cast** and **reinterpret_cast** as a last resort, since these operators present the same dangers as old style casts. However, they are still necessary in order to completely replace old style casts.

Let us see these cast operators one by one.

3. Static_cast

A **static_cast** is used for all conversions that are well-defined. These include "safe" conversions that the compiler would allow you to do without a cast and less-safe conversions that are however well-defined. The types of conversions covered by **static_cast** include typical castless conversions, forcing a conversion from a **void*** and implicit type conversions, converting between related types, such as numeric types, etc.

The static_cast operator has the following **syntax**:

```
static_cast<Type>(expression)
```

The static_cast operator converts a given expression to a specified type.

For example: The expression *static_cast<T>(v)* converts the value of the expression *v* to that of type *T*. It can be used for any cast that is performed implicitly on assignment. In addition, any value may be cast to void, and any implicit cast can be reversed if that cast would be legal as an old-style cast.

Program demonstrating the use of static_cast operator

```
#include<iostream>
using namespace std;
int main() {
  int a = 53;
  int b = 2;
  float d = a/b;
  float m = static_cast<float>(a)/b;
  cout << "d = " << d << endl;
  cout << "m = " << m << endl;
}
```

Output

d = 26

d = 26.5

In this example, d = *a/b;* produces an answer of type int because both *a* and *b* are integers. Conversely, *m* = *static_cast<float>(a)/b;* produces an answer of type float. The *static_cast* operator converts variable *a* to a type *float*. This allows the compiler to generate a division with an answer of type *float*. All static_cast operators resolve at compile time and do not remove any const or volatile modifiers.

Applying the static_cast operator to a null pointer will convert it to a null pointer value of the target type. You can explicitly convert a pointer of a type `A` to a pointer of a type `B` if `A` is a base class of `B`. If `A` is not a base class of `B`, a compiler error will result. You may cast an lvalue of a type `A` to a type `B&` if the following are true:

i. A is a base class of B.

ii. You are able to convert a pointer of type A to a pointer of type B.

iii. The type B has the same or greater const or volatile qualifiers than type A.

iv. A is not a virtual base class of B.

v. The result is an lvalue of type B.

A pointer to member type can be explicitly converted into a different pointer to member type if both types are pointers to members of the same class. This form of explicit conversion may also take place if the pointer to member types is from separate classes, however one of the class types must be derived from the other.

4. Dynamic_cast

The dynamic_cast operator performs type conversions at run time. The dynamic_cast operator converts the base class pointer to a derived class pointer. You can use dynamic_cast only when the base class has at least one virtual pointer, i.e., it performs casts on polymorphic objects.

The **dynamic_cast** operator:

i. Makes downcasting much safer than conventional static casting.

ii. Obtains a pointer to an object of a derived class that is given a pointer to a base class of that object.

iii. Returns the pointer only if the specific derived class actually exists.

Note: If the specified derived class does not exist zero is returned.

Dynamic casts have the following **Syntax**:

```
dynamic_cast<type_name>(expression)
```

The operator converts the expression to the desired type *type_name*. The type_name can be a pointer or a reference to a class type. If the cast to type_name fails, the value of the expression is zero. *For example*: The expression *dynamic_cast<T>(v)* converts the expression *v* to type *T*.

Type *T* must be a pointer or reference to a complete class type or a pointer to void. If *T* is a pointer and the *dynamic_cast* operator fails, the operator returns a null pointer of type *T*. If *T* is a reference and the *dynamic_cast* operator fails, the operator throws the exception *std::bad_cast*. You can find this class in the standard library header *<typeinfo>*.

The primary purpose for the *dynamic_cast* operator is to perform type-safe downcasts. A downcast is the conversion of a pointer or reference to a class *A* (base class) to pointer or reference to a class *B* (derived class), where class *A* is a base class of *B*. The problem with downcasts is that a pointer of type A* can and must point to any object of a class that has been derived from *A*.

The *dynamic_cast* operator ensures that if you convert a pointer of class *A* to a pointer of a class *B*, the object that *A* points to belongs to class *B* or a class derived from *B*.

Program demonstrating the use of dynamic_cast operator

```
#include<iostream>
using namespace std;
class B
{ public:
  virtual void func()
   { cout << "Inside class Base \n"; }
```

```
};
class D: public B
{ public:
   void func()
   { cout << "Inside derived class \n"; }
};
int main()
{ B *bp, b_obj;
   D *dp, d_obj;
   dp = dynamic_cast<D*> (&d_obj);
   if(dp)
   { cout<<"Cast from derived class* to derived class*\n";
     dp → func();}
   else
   { cout<<"Error \n";}
     bp = dynamic _cast<B*>(&d_obj);
     if(bp)
     { cout<<"Cast from derived* to base* \n";
       bp→func(); }
   else
     cout<<"Error \n";
     bp = dynamic_cast<B*> (&b_obj);
     if(bp)
     { cout<<"Cast from base* to base*\n";
       bp→func(); }
   else
     cout<<"Error \n";
     dp = dynamic_cast(D*) (&b_obj);
     if(dp)
     { cout<<"Error \n";    }
   else
     { cout<<"Cast from base* to derived* not possible. \n";    }
       bp = &d_obj; //bp points to derived obj.
       dp = dynamic_cast<D*> (bp);
   if(dp)
     {     cout<<"Casting bp to a derived possible as"
           <<"bp is pointing to a derived object. \n";
     dp→func()}
  else
     cout<<"Error \n";
     bp = &b_obj;
     dp = dynamic_cast<D*> (bp);
     if(dp)
     cout<<"Error";
   else
     cout<<"Casting bp to a derived not possible as"
           <<"bp is pointing to a Base Object. \n";
     dp = &d_obj;
     bp = dynamic_cast<B*> (dp);
   if(bp)
   { cout<<"Casting dp to a base* \n";
     bp→func()   }
```

```
    else
    cout<<"Error \n";
    return 0;
 }
```

Output

Cast from derived class *to derived class*

Inside derived class

Cast from Derived *to base*

Inside derived class

Cast from base* to base*

Inside class Base

Cast from base* to derived* not possible.

Casting bp to a derived possible as bp is pointing to a derived object

Inside derived class

Casting bp to a derived not possible as bp is pointing to a Base object.

Casting dp to a base*

Inside derived class.

5. Const_Cast

A *const_cast operator* is used to add or remove a **const** or **volatile** modifier to or from a type.

This is the *only* conversion allowed with **const_cast**; if any other conversion is involved it must be done separately or you'll get a compile-time error. Remember also that while const_cast may remove the const qualifier of an object, this doesn't mean that you're allowed to modify it. In fact, trying to modify a const object causes undefined behavior. Therefore, use const_cast cautiously when it is used for the removal of const or volatile.

Syntax

```
const_cast<Type>(expression)
```

Type and the type of expression may only differ with respect to their **const** and **volatile** qualifiers. Their cast is resolved at compile time. A single **const_cast** expression may add or remove any number of **const** or **volatile** modifiers. The result of a **const_cast** expression is an rvalue unless Type is a reference type. In this case, the result is an lvalue. Types cannot be defined within **const_cast**.

Program demonstrating the use of const_cast operator

```
#include<iostream>
using namespace std;
void f(int* p) {
  cout << *p << endl; }
int main(void) {
```

```
  const int a = 10;
  const int* b = &a;
   // Function f() expects int*, not const int*
  //   f(b);
  int* c = const_cast<int>(b);
  f(c);
   // Lvalue is const
  //   *b = 20;
   // Undefined behavior
  //   *c = 30;
   int a1 = 40;
  const int* b1 = &a1;
  int* c1 = const_cast<int>(b1);
   //Integer a1, the object referred to by c1, has not been declared
//const
  *c1 = 50;
   return 0;
 }
```

The compiler will not allow the function call `f(b)`. Function `f()` expects a pointer to an **int**, not a **const int**. The statement `int* c = const_cast<int>(b)` returns a pointer c that refers to `a` without the **const** qualification of `a`. This process of using **const_cast** to remove the **const** qualification of an object is called *casting away constness*. Consequently the compiler will allow the function call `f(c)`.

The compiler would not allow the assignment `*b = 20` because `b` points to an object of type **const int**. The compiler will allow the `*c = 30`, but the behavior of this statement is undefined. If you cast away the constness of an object that has been explicitly declared as **const**, and attempt to modify it, the results are undefined.

However, if you cast away the constness of an object that has not been explicitly declared as **const**, you can modify it safely. In the above example, the object referred to by `b1` has not been declared **const**, but you cannot modify this object through `b1`.

You may cast away the constness of `b1` and modify the value to which it refers.

6. Reinterpret_cast

A reinterpret_cast operator handles conversions between unrelated types. This is the least safe of the casting mechanism and the one most likely to point to bugs. At the very least, your compiler should contain switches to allow you to force the use of **const_cast** and **reinterpret_cast**, which will locate the most unsafe of the casts.

Syntax

```
reinterpret_cast<Type>(expression)
```

The reinterpret_cast operator produces a value of a new type that has the same bit pattern as its argument. reinterpret_cast cannot be used to convert between pointers to two different classes that are related by inheritance (use static_cast or dynamic_cast), nor can it be used to cast away const or volatile qualification (use const_cast).

You can explicitly perform the following conversions:

i. A pointer to any integral type large enough to hold it

ii. A value of integral or enumeration type to a pointer

iii. A pointer to a function to a pointer to a function of a different type

iv. A pointer to an object to a pointer to an object of a different type

v. A pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

vi. A null pointer value is converted to the null pointer value of the destination type. In the following example, reinterpret_cast is used to "cheat" the compiler, enabling the programmer to examine the individual bytes of a float variable:

```
float f=10;
unsigned char *p = reinterpret_cast<unsigned char*> (&f);
for(int j=0; j<4; ++j)
 cout<<p[j]<<endl;
```

The use of reinterpret_cast explicitly warns the reader that an unsafe (and probably a nonportable) conversion is taking place. When using reinterpret_cast, the programmer rather than the compiler is responsible for the results.

Program demonstrating the use of reinterpret_cast operator

```
#include<iostream>
using namespace std;
int main()
{ int i;
  char *p = "This is a string";
  i = reinterpret_cast<int> (p); // cast pointer to integer
  cout << i;
  return 0;
 }
```

Output

7648

7. Run-Time Type Information (RTTI)

Run-Time Type Information (RTTI) is a major extension to the C++ language made by the ISO standard committee. Run-Time Type Information (RTTI) is a mechanism that allows the type of an object to be determined during program execution. RTTI was added to the C++ language because many vendors of class libraries were implementing this functionality themselves. This caused incompatibilities between libraries. Thus, it became obvious that support for run-time type information was needed at the language level. *There are three main C++ language elements to run-time type information:*

> **1** PU
> **Oct. 2009 – *5M***
> ★ What is RTTI? Explain with suitable example.

> **1** PU
> **Apr. 2010 – *5M***
> ★ Write short note on RTTI.

i. **The dynamic_cast operator** : We have already seen that the dynamic cast operator used for conversion of polymorphic types. This operator combines type-checking and casting in one operation. It checks whether the requested cast is valid, and performs the cast only if it is valid.

ii. **The typeid operator:** This operator returns the run-time type of an object, i.e., it is used for identifying the exact type of an object. If the operand provided to the typeid operator is the name of a type, the operator returns a type_info object that identifies it. If the operand provided is an expression, typeid returns the type of the object that the expression denotes.

iii. **The type_info class** : This class describes the RTTI available, and is used to define the type information returned by the typeid operator. This class provides to users the possibility of shaping and extending RTTI to suit their own needs. This ability is of most interest to implementers of object I/O systems such as debuggers or database systems.

7.1 The typeid Operator

The `typeid` operator provides a program with the ability to retrieve the actual derived type of the object referred to by a pointer or a reference. This operator, along with the dynamic_cast operator, are provided for RTTI support in C++.

Syntax

```
typeid(type-id)
typeid(expr)
```

The typeid operator requires RTTI to be generated, which must be explicitly specified at compile time through a compiler option.

The result of typeid operator is **const std::type_info** or **const type_info &** that represents the type of expression *expr*. You must include the standard template library header <*typeinfo*> to use the typeid operator.

If expr is a reference or a dereferenced pointer to a polymorphic class, *typeid* will return a *type_info* object that represents the object that the reference or pointer denotes at run time. If it is not a polymorphic class, *typeid* will return a *type_info* object that represents the type of the reference or dereferenced pointer. The following example demonstrates this:

```
#include<iostream>
#include<typeinfo>
using namespace std;
struct A { virtual ~A() { } };
struct B : A { };
struct C { };
struct D : C { };
int main() {
  B bobject;
  A* ap = &bobject;
  A& ar = bobject;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;
  D dobject;
  C* cp = &dobject;
  C& cr = dobject;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}
```

Output

ap: B

ar: B

cp: C

cr: C

Classes *A* and *B* are polymorphic; classes *C* and *D* are not. Although *cp* and *cr* refer to an object of type *D, typeid(*cp)* and *typeid(cr)* return objects that represent class *C*. lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions will not be applied to expr. *For example*: The output of the following example will be `int [10]`, not `int *`:

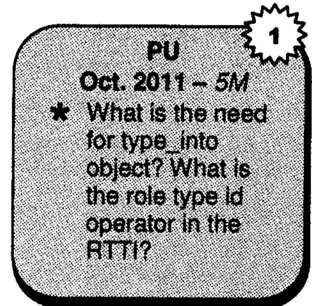
```
#include<iostream>
#include<typeinfo>
using namespace std;
int main() {
  int myArray[10];
  cout << typeid(myArray).name() << endl;
}
```

If *expr* is a class type, that class must be completely defined.

The *typeid* operator ignores top-level *const* or *volatile* qualifiers.

7.2 The type_info Class

The class type_info describes type information generated by the typeid operator. The primary functions provided by type_info are equality, inequality, before and name. From <typeinfo.h>, the definition is:




```
class type_info
{    public:
    virtual ~type_info();
    bool operator==(const type_info &rhs) const;
    bool operator!=(const type_info &rhs) const;
    bool before(const type_info &rhs) const;
    const char *name() const;
    private:
    type_info(const type_info &rhs);
    type_info &operator=(const type_info &rhs); };
```

The overloaded == and != provide for the comparison of types. The **before()** function return true if the invoking object used as parameter in collation order. This function is mostly for internal use only. Its return value has nothing to do with inheritance or class hierarchies. The **name()** function returns a pointer to the name of the type. The constructor is a private member function, so there is no way for a programmer to create a variable of type "type_info". The only source of "type_info" objects is in the "typeid" operator.

8. A Simple Application of Run-Time Type ID

The following program hints at the power of RTTI. In the following program, the function **factory()** creates instances of various types of objects derived from the class Mammal. (A function which produces objects is sometimes called an object factory.). The specific type of object created is determined by the outcome of a call to rand(), C++'s random number generator. Thus, there is no way to know in advance what type of object will be generated. The program creates 10 objects and counts the number of each type of mammal. Since any type of mammal may be generated by a call to factory(), the program relies upon typeid to determine which type of object has actually been made.

Program demonstrating the run-time type id

```
#include<iostream>
using namespace std;
class Mammal{
public:
  virtual bool lays_eggs()
  { return false;}   //Mammal is polymorphic
    // . . .
  };
  class Rat: public Mammal{
  public:
    //.  . . };
  class Platypus: public Mammal{
  public:
    bool lays_eggs() {return true;}
```

```
    //. . . };
  class Cat:public Mammal{
  public:
    //. . . };
  // A factory for objects derived from Mammal.
  Mammal *factory()
  {    switch (rand()%3){
       case 0: return new Cat;
       case 1: return new Rat;
       case 2: return new Platypus; }
  return 0; }
int main()
{ Mammal *ptr;     // pointer to base class
  int i;
  int c=0, r=0, p=0;
  //generate and count objects
  for(i=0;i<10;i++){
  ptr=factory();   //generate an object
  cout<<"object is"<<typeid(*ptr).name();
  cout<<endl;
  //count it
  if(typeid(*ptr)==typeid(Cat))
  c++;
  if(typeid(*ptr)==typeid(Rat))
  r++;
  if(typeid(*ptr)==typeid(Platypus))
  p++; }
cout<<endl;
cout<< "Animals generated:\n";
cout<< "Cats:" << c << endl;
cout<< "Rats:" << r << endl;
cout<< "Platypus:" << p << endl;
return 0;
 }
```

Output

Object is class Platypus
Object is class Platypus
Object is class Rat
Object is class Rat
Object is class Platypus
Object is class Rat
Object is class Cat
Object is class Cat
Object is class Rat
Object is class Platypus
Animals generated:
Cats:2
Rats:4
Platypus:4

9. TYPEID can be applied to Template Classes

We can apply the typeid operator to template classes. The type of an object that is an instance of a template class is in part determined by what data is used for its generic data when the object is instantiated. Two instances of the same template class that are created using different data are therefore different types. Here is a example:

Program: Using typeid with templates

```
#include<iostream>
using namespace std;
template<class T> class myclass{
T a;
public:
  Myclass(T i) {a=i;}
  //. . . };
int main()
{ myclass<int> obj1(10), obj2(8);
  myclass<double> obj3(8.3);
  cout << "Type of obj1 is";
  cout << typeid(obj1).name() << endl;
  cout << "Type of obj2 is";
  cout << typeid(obj2).name() << endl;
  cout << "Type of obj3 is";
  cout << typeid(obj3).name() << endl;
  cout << endl;
  if(typeid(obj1)==typeid(obj2))
    cout<< "obj1 and obj2 are of the same type\n";
  if(typeid(obj1)==typeid(obj3))
    cout << "Error\n";
  else
    cout << "obj1 and obj3 are of different types\n";
  return 0;
 }
```

Output

Type of obj1 is class myclass<int>
Type of obj2 is class myclass<int>
Type of obj3 is class myclass<double>
obj1 and obj2 are of the same type
obj1 and obj3 are different types

Note that even though two objects are of the same template class type, if their parameterized data does not match, they are not equivalent types. In the above program, obj1 is of type myclass<int> and obj3 is of type myclass<double>. Thus, they are of different types.

EXERCISES

A. Review Questions

1. List the new operators added by the ANSI C++ standard committee.
2. What is the application of dynamic_cast operator?
3. How does the reinterpret_cast differ from the static_cast?
4. What is the dynamic casting? How is it achieved in C++?
5. What is typeid operator? When is it used?

B. Programming Exercises

1. Write a program to demonstrate the use of dynamic_cast operator.
2. Write a program to demonstrate the use of typeid operator.
3. Use RTTI to assist in program debugging by printing out the exact name of a template using **typeid( )**. Instantiate the template for various types and see what the results are.
4. What is the problem with the following statements?

```
const int m=100;
double *ptr= const_cast<double*>(&m);
```

5. What will be the output of the following program?

```
#include<iostream.h>
class person{ //. . . }
int main()
{ person abc;
  cout<<"abc is a";
  cout<<typeid(abc).name()<<"\n";   }
```

Collection of Questions asked in Previous Exams PU

1. What is RTTI? Explain with suitable example. [Oct. 2009 – 5M]
2. Write short note on New Style Casts. [Oct. 2009, Oct. 2010 – 5M]
3. Write short note on RTTI. [Apr. 2010 – 5M]
4. What is the need for type_into object? What is the role type id operator in the RTTI? [Oct. 2011 – 5M]

Suggestive Readings:

1. Krishna Mohan &MeeraBanerji. Developing Communication Skills; Macmillan Publishers Ltd
2. Dr.K.Alex. Soft Skills; S.Chand Publishing
3. Swets, Paul. W. 1983. The Art of Talking So That People Will Listen: Getting Through to Family, Friends and Business Associates. Prentice Hall Press. New York
4. Lewis, Norman. 1991. Word Power Made Easy. Pocket Books
5. Herbert Schildts : C++ - The Complete Reference, Tata McGraw Hill Publications.
6. Balaguru Swamy : C++, Tata McGraw Hill Publications.
7. Balaguruswamy : Object Oriented Programming and C++, TMH.
8. Shah & Thakker : Programming in C++, ISTE/EXCEL.
9. Johnston : C++ Programming Today, PHI.
10. Object Oriented Programming and C++, Rajaram, New Age International.
11. Samanta : Object Oriented Programming with C++ & JAVA, PHI.