


# OBJECT-ORIENTED PROGRAMMING

Edited By
Dr. Mithilesh Kumar Dubey

ISBN: 978-93-87034-82-2

Printed by
**EXCEL BOOKS PRIVATE LIMITED**
Regd. Office: E-77, South Ext. Part-I, Delhi-110049
Corporate Office: 1E/14, Jhandewalan Extension, New Delhi-110055
+91-8800697053, +91-011-47520129
info@excelbooks.com/projects@excelbooks.com
internationalalliance@excelbooks.com
www.excelbooks.com

for
Lovely Professional University
Phagwara

# CONTENTS


| | | |
|---|---|---|
| **Unit 1:** | Review of Object-oriented Programming<br>*Balraj Kumar, Lovely Professional University* | 1 |
| **Unit 2:** | Beginning of OOP Language<br>*Manpreet Kaur, Lovely Professional University* | 18 |
| **Unit 3:** | Review of Functions<br>*Kumar Vishal, Lovely Professional University* | 56 |
| **Unit 4:** | Classes and Objects<br>*Manmohan Sharma, Lovely Professional University* | 68 |
| **Unit 5:** | Static Members<br>*Anil Sharma, Lovely Professional University* | 92 |
| **Unit 6:** | Constructors and Destructors<br>*Kumar Vishal, Lovely Professional University* | 115 |
| **Unit 7:** | Operator Overloading<br>*Balraj Kumar, Lovely Professional University* | 135 |
| **Unit 8:** | Type Conversion<br>*Mithilesh Kumar Dubey, Lovely Professional University* | 156 |
| **Unit 9:** | Inheritance<br>*Sahil Rampal, Lovely Professional University* | 168 |
| **Unit 10:** | Virtual Functions and Polymorphism<br>*Ajay Kumar Bansal, Lovely Professional University* | 210 |
| **Unit 11:** | Pointers and Dynamic Memory Management<br>*Mandeep Kaur, Lovely Professional University* | 231 |
| **Unit 12:** | Console I/O<br>*Ajay Kumar Bansal, Lovely Professional University* | 252 |
| **Unit 13:** | Working with Files<br>*Yadwinder Singh, Lovely Professional University* | 271 |
| **Unit 14:** | Advanced Concept in C++<br>*Pooja Gupta, Lovely Professional University* | 302 |

# SYLLABUS

## Object-oriented Programming

***Objectives:*** To impart the skills needed for Object Oriented programming and Console applications development. Student can map real world objects into programming objects. Implement the concept of reusability and data security. Learn user file management containing data.

| Sr. No. | Description |
|---|---|
| 1. | ***Review:*** Review of Basic Concepts of Object-oriented Programming & Introduction of OOP Languages, Comparison between Procedural Programming Paradigm and Object-oriented Programming Paradigm. |
| 2. | ***Beginning with OOP Language:*** Review of Tokens, Expressions, Operators & Control Structures.<br>Scope Resolution Operator, Member Dereferencing Operator, Reference Variables.<br>Review of Functions, Function Overloading, Inline Functions, Default Arguments. |
| 3. | ***Classes & Objects:*** Specifying a Class, Defining Member Functions, Creating Class Objects, Accessing Class Members. Access Specifies – Public, Private, and Protected Classes, Its Members, Objects and Memory Allocation |
| 4. | Static Members, the Const Keyword and Classes, the Static Objects. Friend Function & its Usage Empty Classes, Nested Classes, Local Classes. |
| 5. | ***Constructors & Destructors:*** Need for Constructors and Destructors, Copy Constructor, Dynamic Constructors, Destructors, Constructors and Destructors with Static Members. |
| 6. | ***Operator Overloading & Type Conversion:*** Defining Operator Overloading, Rules for Overloading Operators, Overloading of Unary Operators and various Binary Operators with Friend Functions and Member Functions.<br>Type Conversion – Basic Type to Class Type, Class Type to Basic Type, Class Type to another Class Type. |
| 7. | ***Inheritance:*** Introduction, Defining Derived Classes, Forms of Inheritance, Ambiguity in Multiple and Multipath Inheritance, Virtual Base Class, Overriding Member Functions, Order of Execution of Constructors and Destructors<br>***Virtual Functions & Polymorphism:*** Virtual Functions, Pure Virtual Functions, Abstract Classes, Introduction to Polymorphism |
| 8. | ***Pointers & Dynamic Memory Management:*** Understanding Pointers, Accessing Address of a Variable, Declaring & Initializing Pointers, Pointer to a Pointer, Pointer to a Function, Dynamic Memory Management – New and Delete Operators, this Pointer. |
| 9. | ***Console I/O:*** Concept of Streams, Hierarchy of Console Stream Classes, Unformatted I/O Operations, Managing Output with Manipulators. |
| 10. | ***Working with Files****:* Opening, Reading, Writing, Appending, Processing & Closing difference Type of Files, Command Line Arguments. |

# Unit 1: Review of Object-oriented Programming

Notes

CONTENTS

Objectives

Introduction

1.1 Basic Concept of Object-oriented Programming

1.2 Introduction to OOP Languages

1.3 Comparison between Procedural Programming and Object-oriented Programming Paradigm

1.3.1 Procedure-oriented Programming Paradigm

1.3.2 Object-oriented Programming Paradigm

1.4 Benefits of OOP

1.5 Applications of OOP

1.6 Software Crisis

1.7 Software Evolution

1.8 Summary

1.9 Keywords

1.10 Review Questions

1.11 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the basic concept of object-oriented programming
- Describe the OOP languages
- Compare the procedural Oriented and object-oriented programming

## Introduction

Programming practices have evolved considerably over the past few decades. As more and more programmers gained experience problems unknown hitherto, began to surface. The programming community became evermore concerned about the philosophy that they adopt in programming and approaches they practice in program development.

Factors like productivity, reliability, cost effectiveness, reusability etc. started to become major concern. A lot of conscious efforts were put to understand these problems and to seek possible solutions. This is precisely the reason why more and more programming languages have been developed and still continue to develop. In addition to this, approaches to program development have also been under intense research thereby evolving different frame works. One such, and probably the most popular one is object-oriented programming approach or simply OOP.

Notes

## 1.1 Basic Concept of Object-oriented Programming

Procedural programming deals with functional parts of the problem. Programmers identify what actions must be taken to solve the problem at hand. Let us now, try to understand what aspect of problems is dealt with in object-oriented approach. We shall be discussing some essential concepts that make a programming approach object-oriented.

In object-oriented parlance, a problem is viewed in terms of the following concepts:

1. Objects
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism
7. Dynamic binding
8. Message passing

Let us now study the entire concept in detail.

1. ***Objects:*** Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user-defined data such as vectors, time and lists.

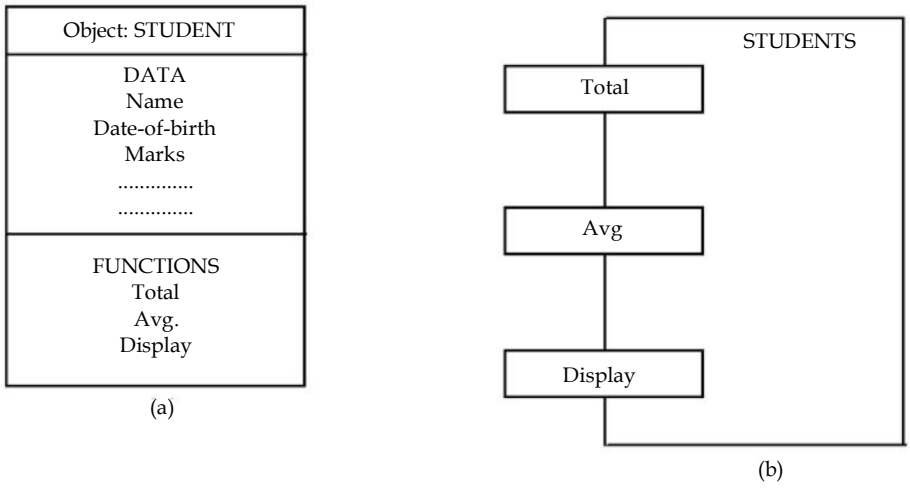
   They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data.



*Notes* Objects can interact without having to know details of each other data or code.

The figure 1.1 shows the two notations popularly used.

**Figure 1.1**



**Notes**

2. ***Classes:*** A class represents a set of related objects. The object has some attributes, whose value consist much of the state of an object. The class of an object defines what attributes an object has. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

   Classes are user defined data types that behave like the built-in types of a programming language. Classes have an interface that defines which part of an object of a class can be accessed from outside and how. A class body that implements the operations in the interface, and the instance variables that contain the state of an object of that class.

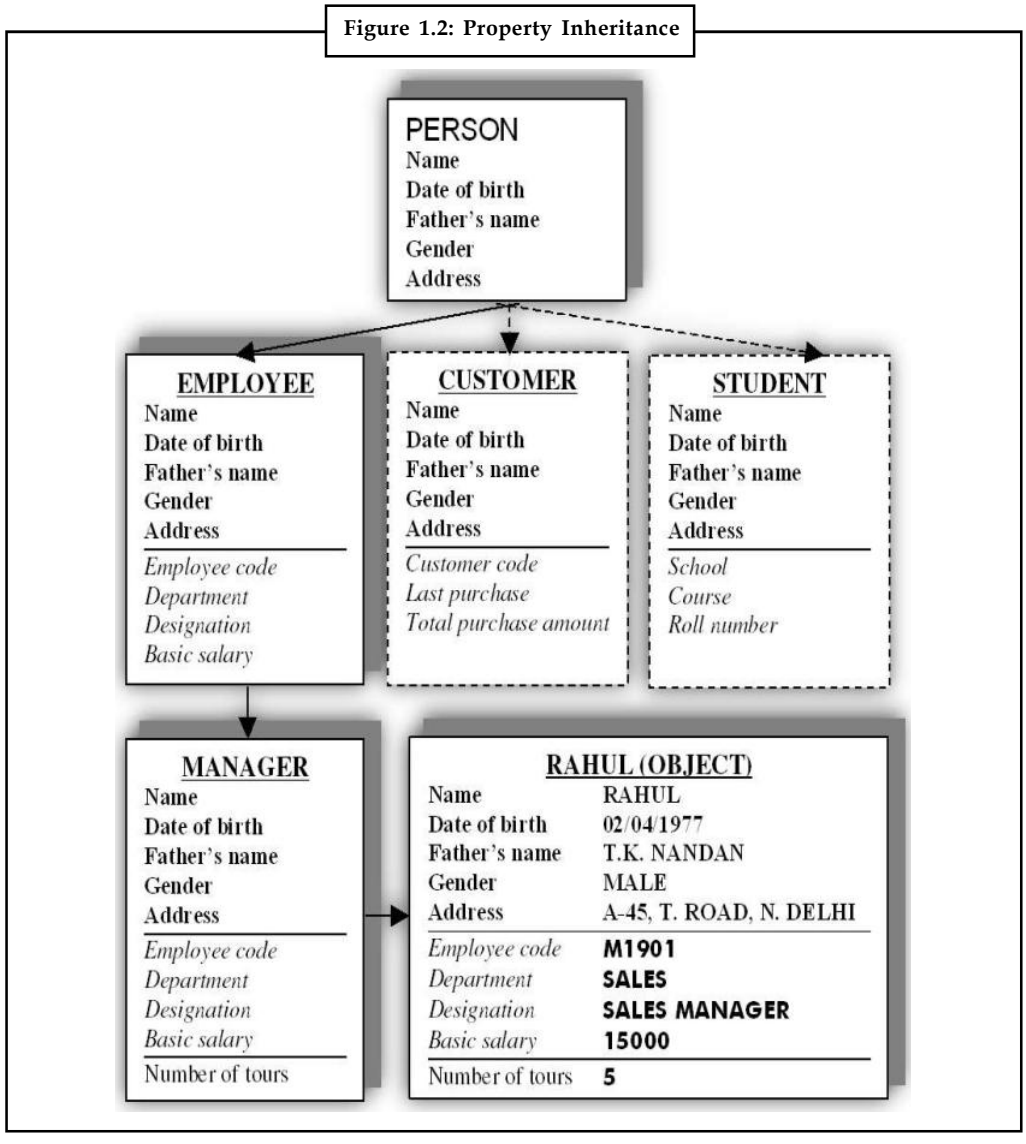
   The data and the operation of a class can be declared as one of the three types:

   (a) ***Public:*** These are declarations that are accessible from outside the class to anyone who can access an object of this class.

   (b) ***Protected:*** These are declarations that are accessible from outside the class to anyone who can access an object of this class.

   (c) ***Private:*** These are declarations that are accessible only from within the class itself.

**Figure 1.2: Property Inheritance**



**Notes**

3. ***Data Abstraction:*** Abstraction refers to the act of representing essential-features without including the background details or explanations. Abstraction is indispensable part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of the system. However, these components are not isolated from each other, they interact with each other. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).



*Example:* We can represents essential features without including background details and explanations.

```
index of text book.
 class School
    {
      void sixthc lass();
      void seventhclass();
       void tenthclass();
     }
```

4. ***Data Encapsulation:*** The wrapping up to data and functions into a single unit (class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions that are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.

5. ***Inheritance:*** Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, an manager class is a type of the class employee, which again is a type of the class person as illustrated below.

   The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived. The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the super class. In other words, a feature is placed in the higher level of abstraction. Once this is done, such features can be inherited from the parent class and used in the subclass directly. This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes.

   In OOP, 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 the combined features of both the classes. Each subclass defines only those features that are unique to it. In OOP, 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 the combined features of both the classes.



*Caution* Keep in mind that each subclass defines only those features that are unique to it.

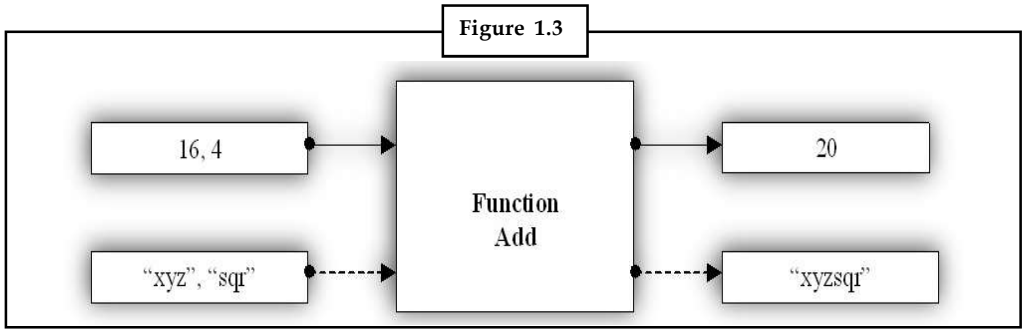
**Notes**

6. ***Polymorphism:*** Polymorphism means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of the data used in the operation. For example considering the operator plus (+).

   16 + 4 = 20

   "xyz" + "sqr" = "xyzsqr"

**Figure 1.3**




   The process of making an operator to exhibit different behavior at different instance is called operator overloading. Another example of polymorphisms is function overloading, where a single function can perform various different types of task.

7. ***Dynamic Binding:*** Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of call at run-time. This is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference. For example in the above figure, by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object so the procedure will be redefined in each class that defines the objects. At run-time, the code matching the object under reference will be called.



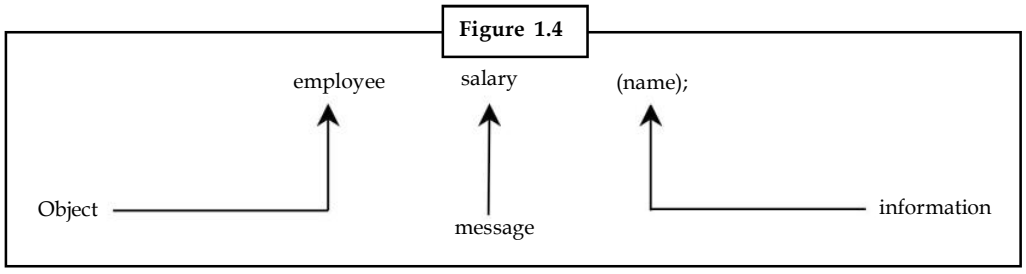
*Did u know?* **What is the difference between the static binding and dynamic binding?**

Static binding defines the properties of the variables at compile time. Therefore, they can't be changed. In dynamic binding the properties of the variables are determined at runtime.

8. ***Message Passing:*** Message passing is another feature of object-oriented programming. An object-oriented program consists of a set of objects that communicate with each other. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

   A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

**Figure 1.4**



**Notes**

As is evident from the ongoing discussion, object-oriented approach deals with objects and classes as against series of functions and as actions adopted by procedural approach.

### Self Assessment

Fill in the blanks:

1. Each object contains data and …… to manipulate the data.
2. …….. have an interface that defines which part of an object of a class can be accessed from outside and how.
3. The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the …… class.
4. At run-time, the code matching the …… under reference will be called.

## 1.2 Introduction to OOP Languages

Object oriented programming is not the right of any particular language. Although languages like C and Pascal can be used but programming becomes clumsy and may generate confusion when program grow in size. A language that is specially designed to support the OOP concepts makes it easier to implement them.

To claim that they are object-oriented they should support several concepts of OOP.

Depending upon the features they support, they are classified into the following categories:

1. Object-based programming languages.
2. Object-oriented programming languages.

**Table 1.1: Characteristics of Some OPP Languages**

| Characteristics | Sim ula | Small talk 80 | Objective C | C ++ | ADA | Object Pascal | Eittel |
|---|---|---|---|---|---|---|---|
| **Binding** (early or late) | Both | Late | Both | Both | Early | Late | Early |
| **polymorphism** (operator overloading) | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Data hiding** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Concurrency** | Yes | Poor | Poor | Poor | Difficult | No | Promised |
| **Inheritance** | Yes | Yes | Yes | Yes | No | Yes | Yes |
| **Multiple Inheritance** | No | Yes | Yes | Yes | No | Yes | Yes |
| **Garbage Collection** | Yes | Yes | Yes | Yes | No | Yes | Yes |
| **Persistence** | No | Pormised | No | No | Like 3GL | No | Some support |
| **Genericity** | No | No | No | No | Yes | No | Yes |
| **Object lib** | Yes | Yes | Yes | No | Not much | Yes | yes |

Notes

Major features required by object-based programming are:

1. Data encapsulation.
2. Data hiding and access mechanisms.
3. Automatic initialization and clear-up of objects.
4. Operator overloading.

Languages that support programming with objects are said to be object-based programming languages. These do not support inheritance and dynamic binding ADA is a typical example Object oriented programming incorporates all of objects-based programming features along with two additional feature, namely, inheritance and dynamic binding. Thus

Object oriented programming = Object-based features + inheritance + dynamic binding.

Languages that support these features include C++, Small talk, Java among others.



*Task* In a group of four explain how language that is specially designed to support the OOP concepts makes it easier to implement them.

## 1.3 Comparison between Procedural Programming and Object-oriented Programming Paradigm

Computer programming has been around for some decades now. The style and manner in which people have been developing programs itself has undergone a sea change. In the early days of computer development programming was looked upon as some kind of black magic – to be understood only by a few wizards mostly the people who had designed the computer. However, the entire scenario has changed now.

In the modern programming parlance, at least in most of the commercial and business applications areas, programming has been made independent of the target machine. This machine independent characteristic of programming has given rise to a number of different methodologies in which programs can now be developed. We will particularly concern ourselves with two broad programming approaches – or paradigm as they are called in the present context.

1. Procedure-oriented paradigm
2. Object-oriented paradigm

### 1.3.1 Procedure-oriented Programming Paradigm

Before you get into OOP, take a look at conventional procedure-oriented programming in a language such as C. Using the procedure-oriented approach; you view a problem as a sequence of things to do such as reading, calculating and printing. Conventional programming using high-level languages is commonly known as procedure-oriented programming.

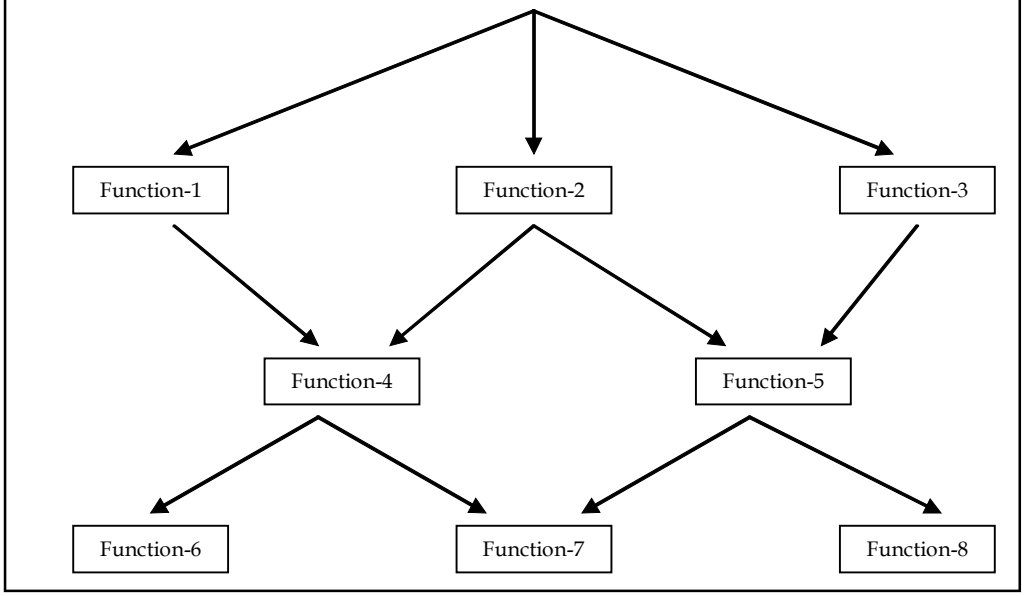


*Example:* COBOL, FORTRAN and C

You organize the related data items into C structures and write the necessary functions (procedures) to manipulate the data and, in the process, complete the sequence of tasks that solve your problem.

**Notes**

**Figure 1.5: Typical Structure of Procedure-oriented Programs**




Although data may be organized into structures, the primary focus is on functions. Each C function transforms data in some way.

*Example:* You may have a function that calculates the average value of a set of numbers, another that computes the square root, and one that prints a string.

You do not have to look far to find examples of this kind of programming – C function libraries are implemented this way. Each function in a library performs a well-defined operation on its input arguments and returns the transformed data as a return value. Arguments may be pointers to data that the function directly alters or the function may have the effect of displaying graphics on a video monitor.

A typical program structure for procedural programming is shown in Figure 1.5. The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem.

Procedure oriented programming basically consists of writing a list of instructions for the computer to following and organizing these instructions into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another.

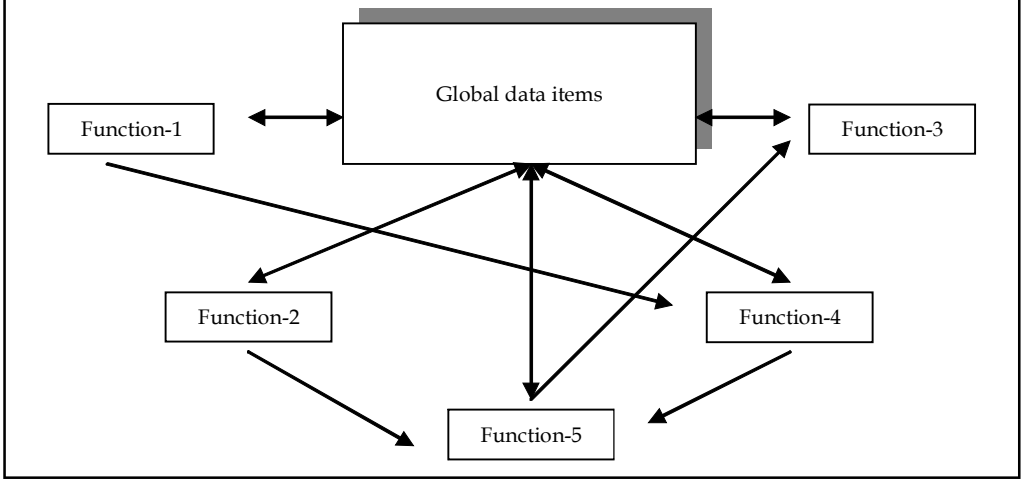
While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to the data? How are they affected by the functions that work on them?

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Figure 1.6 shows the relationship of data and functions in a procedure-oriented program.

Global data are more vulnerable 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 case we need to revise an external data structure, we should also revise all functions that access the data. This provides an opportunity for bugs to creep in.

Notes

**Figure 1.6: Relationship of Data and Functions in Procedural Programming**






*Did u know?* **What is the basic feature of Procedural-oriented Programming?**

The basic feature of Procedural-oriented Programming is to reuse the same code at different places in the program without coping it.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

1. Emphasis is on doing things (algorithms).
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data.
4. Data move openly around the system from function to function.

### 1.3.2 Object-oriented Programming Paradigm

The term Object-oriented Programming (OOP) is widely used, but experts do not seem to agree on its exact definition. However, most experts agree that OOP involves defining Abstract Data Types (ADT) representing complex real-world or abstract objects and organizing programs around the collection of ADTs with an eye toward exploiting their common features. The term data abstraction refers to the process of defining ADTs; inheritance and polymorphism refer to the mechanisms that enable you to take advantage of the common characteristics of the ADTs – the objects in OOP.

Before going any further into OOP, take note of two points. First, OOP is only a method of designing and implementing software. Use of object-oriented techniques does not impart anything to a finished software product that the user can see. However, as a programmer while implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. You can take advantage of the modularity of objects and implement the

Notes

program in relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.



*Task* An object is a software bundle of related state and behavior. Analyze.

Secondly, OOP has nothing to do with any programming language, although a programming language that supports OOP makes it easier to implement the object-oriented techniques. As you will see shortly, with some discipline, you can use objects even in C programs.

### Self Assessment

Fill in the blanks:

5. Languages that support programming with objects are said to be …….. programming languages.
6. The colossal efforts that went into developing this large population of programs would be rendered useless if these codes could not be……..
7. The technique of …….. decomposition has been used to specify the tasks to be completed in order to solve a problem.
8. ……. is only a method of designing and implementing software.

## 1.4 Benefits of OOP

Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The technology provides greater programmer productivity, better quality of software and lesser maintenance cost. The principle advantages are:

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
2. 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.
3. The principle of data hiding help the programmer to build secure programs that cannot be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist without any interference.
5. It is possible to map objects in the problem domain to those objects in the program.
6. It is easy partition the work in a project based on objects.
7. The data-centered design approach enables us to capture more details of a model in implementable form.
8. Object oriented systems can be easily upgraded from small to large systems.
9. Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
10. Software complexity can be easily managed.

Notes

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.



*Notes* Developing software that is easy to sue makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

### Self Assessment

Fill in the blanks:

9. OOP can eliminate …… code and extend the use of existing classes.
10. It is possible to have …… instances of an object to co-exist without any interference.

## 1.5 Applications of OOP

Object-oriented approach offers perhaps the most logical description of the real world. Therefore, it can be applied in almost all the situations of problem solving. Because of its effectiveness in problem solving and programming, OOP has been adopted all over the software industry. The existing systems are being migrated to OOP.

One important aspect of OOP that is particularly beneficial is the framework, which encompasses all the phases of a system development. Thus, OOA (Object Oriented Analysis), OOD (Object Oriented Design), OOT (Object Oriented Testing) etc. are far more suitable tools than their non-object-oriented counterparts.

Above all, the reusability feature of the Object Oriented approach has facilitated the rapid growth of software both in variety and scope.



*Did u know?* **What is Simula?**

Simula was the first object-oriented programming language.

OOP Provides Many Applications:

- ***Real time Systems:*** A real time system is a system that give output at given instant and its parameters changes at every time. A real time system is nothing but a dynamic system. Dynamic means the system that changes every moment based on input to the system. OOP approach is very useful for Real time system because code changing is very easy in OOP system and it leads toward dynamic behavior of OOP codes thus more suitable to real time system.
- ***Simulation and Modelling:*** System modelling is another area where criteria for OOP approach is countable. Representing a system is very easy in OOP approach because OOP codes are very easy to understand and thus is preferred to represent a system in simpler form.

Notes

- ***Hypertext and Hypermedia*:** Hypertext and hypermedia is another area where OOP approach is spreading its legs. Its ease of using OOP codes that makes it suitable for various media approaches.
- ***Decision support system*:** Decision support system is an example of Real time system that too very advance and complex system. More details is explained in real time system.
- ***CAM/CAE/CAD System*:** Computer has wide use of OOP approach. This is due to time saving in writing OOP codes and dynamic behavior of OOP codes.
- ***Office Automation System*:** Automation system is just a part or type of real time system. Embedded systems make it easy to use OOP for automated system.
- ***AI and expert system*:** It is mixed system having both hypermedia and real time system.

### Self Assessment

Fill in the blanks:

11. OOP approach is very useful for Real time system because code …….. is very easy in OOP system.
12. Automation system is just a part or type of …… system.

## 1.6 Software Crisis

Exactly what gave birth to object-oriented approach in programming couldn't be stated clearly. However, it is almost certain that by the end of last decade millions and millions lines of codes have been designed and implemented all over the world. The colossal efforts that went into developing this large population of programs would be rendered useless if these codes could not be reused.

Each time, writing a program used to be a new project in itself having nothing to do with the old existing codes. Programmers were, most of time, busy creating perhaps the same thing again and again; writing same or similar codes repeatedly in each project implementation.

Software industry faced a set of problems that are encountered in the development of computer software other than programs not functioning properly. Surveys conducted time to time revealed that more and more software development projects were failing to come to successful completion. Schedule and cost overrun were more frequent than seemed to be. This situation has been dubbed by various industry experts as "software crisis".

In the wake of this, questions like – how to develop software, how to support a growing volume of existing software, and how to keep pace with a rapidly growing demand for more error-free and efficient software – became important concerns in connection with software development activities.

Soon software crisis became a fact of life. Engineers and Scientists of the computing community were forced to look into the possible causes and to suggest alternative ways to avoid the adverse affects of this software crisis looming large at software industry all over the world.

It is this immediate crisis that necessitated the development of a new approach to program designing that would enhance the reusability of the existing codes at the least. This approach has been aptly termed as object-oriented approach.

Notes



*Task* Software industry faced a set of problems. What are the problems they are talking about?

## 1.7 Software Evolution

Unlike consumer products, software is not manufactured. In this sense, software is not a passive entity rather it behaves organically. It undergoes a series of evolutionary stages throughout its lifetime – starting from a problem terminating into a solution. That is why software is said to 'develop' or 'evolve' and not manufactured.

In other words, software is born, passes through various developmental phases, gets established, undergoes maintenance and finally grows old before being commissioned out of service.

Software engineers have developed a number of different 'life styles' through which software passes. In general these life styles are known as software development life cycle or SDLC in short.

Following are the developmental phases of software.

### *Study*

The genesis of any software begins with the study of the problems, which it intends to solve. Software cannot be envisaged unless there is a problem that it must solve. Therefore, studying the problem in depth, understanding the true nature of the problem and representing the problem in comprehensible manner is what necessitates inclusion of this phase.

### *Analysis*

It is a detailed study of the various operations performed by the proposed software. A key question that is considered in this phase of development is – What must be done to solve the problem? One aspect of analysis is defining the boundaries or interface of the software.

During analysis, data are collected in available files, decision points, and transactions handled by the present system. Bias in data collection and interpretation can be fatal to the developmental efforts.



*Notes* Training, experience and common sense are required for collection of the information needed to do the analysis.

Once analysis is completed the analyst has a firm understanding of what is to be done. The next step is to decide how the problem might be solved. Thus, in the software systems design, we move from the logical to the physical aspects of the life cycle.

### *Design*

The most creative and challenging phase of software life cycle is design. The term design describes both a final software system and a process by which it is developed. It refers to the technical specifications (analogous to the engineer's blueprints) that will be applied in implementing the

Notes

software system. It also includes testing the software. The key question around which this phase revolves is – How should the problem be solved?

The first step is to determine how the output is to be produced and in what format. Samples of the output (and input) are outlined.

Second, input data and master files (data base) have to be designed to generate the required output. The operational (processing) phase are handled through program construction and testing, including a list of the programs needed to meet the software objectives and complete documentation.

Finally, details related to justification of the system and an estimate of the impact of the software on the user are documented and evaluated before it is implemented.

### *Implementation*

This phase is primarily concerned with coding the software design into an appropriate programming language; testing the programs and installing the software. User training, site preparation, and data migration are other important issues in this phase.

### *Maintenance*

Change is inevitable. Software serving the users' needs in the past may become less useful or sometimes useless in the changed environment. Users' priorities, changes in organizational requirements, or environmental factors may call for software enhancements. A bank, for instance, may decided to increase its service charges for checking accounts from ₹ 3.00 to ₹ 450 for a minimum balance of ₹ 3,000. In this phase the software is continuously evaluated and modified to suit the changes as they occur.

## Self Assessment

Fill in the blanks:

13. During analysis, data are collected in available files, …… , and transactions handled by the present system.
14. User training, site preparation, and …… are other important issues in implementation phase.
15. The ……. phase is handled through program construction and testing.



*Caselet*

## Arithnet Technical Offers Model-based Testing Tools

Arithnet Technical Services, a subsidiary of Denmark-based ATS, providing Model-based Testing (MBT) solutions, is in the process of expanding its Indian operations. This will see a gradual ramp-up of its numbers to move to a support centre. The company plans to offer its testing solutions in an outsourced process model.

The leader of RedVest group of the Institute For Systems Programming of the Russian Academy of Sciences, Mr Alexander K. Petrenko, who is associated with product development in this space, told Business Line that the global market was gradually

*Contd...*

Notes

emerging towards a model-based testing approach. This would offer a whole range of applications for the technology products companies.

Mr Balaswamy, General Manager of Arithnet, said there had been significant growth in MBT in the last few years due to the popularity of object-oriented programming and models in software engineering. Explaining the advantages of MBT, Prof. Petrenko said in the traditional testing, a model of the application to be tested is implicit only in the testers mind. MBT takes the model out of the testers mind making it more useful for multiple testing tasks, shareable, reusable and describing more precisely the system to be tested.

These tool kits are integrated into popular integrated development environments such as Visual Studio from Microsoft or Forte from Sun Microsystems and support all phases of MBT model based testing— specifications, test design, test care generation from specifications, test execution and automatic analysis of outcome.

ATS has entered into a technical collaboration with the Institute For Systems Programming of the Russian Academy of Sciences, Moscow, for development of several software testing related products. While reaching out its MBT solutions, ATS provides services right from consultancy services, outsourced testing services as also training for corporations. "We are hosting interactive meetings with technology companies in Hyderabad to enable them to benefit from our solutions," Mr Immannuel Selvaraj, International Marketing Manager, Arithnet, said.

## 1.8 Summary

- Programming practices have evolved considerably over the past few decades.
- By the end of last decade, millions and millions lines of codes have been designed and implemented all over the word.
- The main objective is to reuse these lines of codes. More and more software development projects were software crisis.
- It is this immediate crisis that necessitated the development of object-oriented approach which supports reusability of the existing code.
- Software is not manufactured. It is evolved or developed after passing through various developmental phases including study, analysis, design, implementation, and maintenance.
- Conventional programming using high level languages such as COBOL, FORTRAN and C is commonly known as procedures-oriented programming.
- In order to Solve a problem, a hierarchical decomposition has been used to specify the tasks to be completed.
- OOP is a method of designing and implementing software.
- Since OOP enables you to remain close to the conceptual, higher-level model of the real world problem, your can manage the complexity better than with approaches that force you to map the problem to fit the features of the language.
- Some essential concepts that make a programming approach object-oriented are objects, classes, Data abstraction, Data encapsulation, Inheritance, Polymorphism, dynamic binding and message passing.
- The data and the operation of a class can be declosed public, protected or private. OOP provides greater programmer productivity, better quality of software and lesser maintenance cost.

Notes

- Depending upon the features they support, they are classified as object based programming languages and object-oriented programming languages.
- Object Oriented approach offers the most logical description of the real world. The framework of OOP encompasses all the phases of system development.
- The reusability features of OOP has facilitated the rapid growth of software.

## 1.9 Keywords

***Classes:*** A class represents a set of related objects.

***Data Abstraction:*** Abstraction refers to the act of representing essential-features without including the background details or explanations.

***Data Encapsulation:*** The wrapping up to data and functions into a single unit (class) is known as encapsulation.

***Design:*** The term design describes both a final software system and a process by which it is developed.

***Dynamic Binding:*** Binding refers to the linking of a procedure call to the code to be executed in response to the call.

***Inheritance:*** Inheritance is the process by which objects of one class acquire the properties of objects of another class.

***Message Passing:*** Message passing is another feature of object-oriented programming.

***Object-oriented Programming Paradigm:*** The term object-oriented programming (OOP) is widely used, but experts do not seem to agree on its exact definition.

***Objects:*** Objects are the basic run-time entities in an object-oriented system.

***Polymorphism:*** Polymorphism means the ability to take more than one form.

## 1.10 Review Questions

1. Outline the essential steps involved in carrying out a procedure-oriented programming study.
2. Inheritance is the process by which objects of one class acquire the properties of objects of another class. Analyze.
3. Examine what are the benefits of OOP?
4. How are classes different from objects? Illustrate with suitable examples.
5. What supports are necessary for a programming language to classify it as an object-oriented programming language?
6. "Software engineers have developed a number of different 'life styles' through which software passes". Do you agree with your statement? Why or why not?
7. The technology provides greater programmer productivity, better quality of software and lesser maintenance cost. Explain how?
8. Scrutinize the meaning of dynamic binding with an example.
9. Make distinctions between data abstraction and data encapsulation. Give examples

10. Compare what you look for in the problem description when applying object-oriented approach in contrast to the procedural approach. Illustrate with some practical examples.

**Notes**

### Answers: Self Assessment

| | | | |
|---|---|---|---|
| 1. | Code | 2. | Classes |
| 3. | Super | 4. | Object |
| 5. | Object-based | 6. | Reused |
| 7. | Hierarchical | 8. | OOP |
| 9. | Redundant | 10. | Multiple |
| 11. | Changing | 12. | Real time |
| 13. | Decision points | 14. | Data migration |
| 15. | Operational | | |

## 1.11 Further Readings



*Books*

E. Balagurusamy, *Object-oriented Programming through C++*, Tata McGraw Hill.

Herbert Schildt, *The Complete Reference – C++*, Tata Mc Graw Hill.

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.



*Online links*

http://en.wikipedia.org/wiki/Object-oriented_programming

http://www.aonaware.com/OOP1.htm

**Notes**

# Unit 2: Beginning of OOP Language

**CONTENTS**

Objectives

Introduction

2.1 Review of Tokens

2.1.1 Keywords

2.1.2 Identifiers

2.2 Assignment Expression

2.3 Operators

2.3.1 Arithmetic Operators

2.3.2 Assignment Operators

2.3.3 Unary Operators

2.3.4 Comparison Operators

2.3.5 Shift Operators

2.3.6 Bit-wise Operators

2.3.7 Logical Operators

2.3.8 Conditional Operators

2.4 Control Structures

2.4.1 The if-else Construct

2.4.2 Looping

2.5 Scope Resolution Operator

2.6 Member Dereferencing Operators

2.6.1 Memory Management Operators

2.6.2 Declaration of Variables

2.6.3 Dynamic Initialization of Variables

2.6.4 Reference Variables

2.7 Summary

2.8 Keywords

2.9 Review Questions

2.10 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the tokens
- Describe the expressions

Notes

- Explain the operators
- Discuss the control structures
- Explain the scope resolution operator
- Describe the member dereferencing operator
- Recognize the reference variable

## Introduction

C++ is a language in essence. It is made up of letters, words, sentences, and constructs just like English language. This unit discusses these elements of the C++ language along with the operators applicable over them.

## 2.1 Review of Tokens

As we know, the smallest individual units in a program are known as tokens. C++ has the following tokens:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators

A C++ program is 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.



*Did u know?* **How tokens are being separated?**

Tokens are usually separated by "white space." White space can be one or more:

(a) Blanks

(b) Horizontal or vertical tabs

(c) New lines

(d) Formfeeds

(e) Comments

### 2.1.1 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Table 2.1 gives the complete set of C++ keywords. The keywords not found in ANSI C are shown boldface. These keywords have been added to the ANSI C keywords in order to enhance its features making it an object-oriented language.

Notes

Table 2.1: C++ Keywords

| | | | |
|---|---|---|---|
| asm | double | new | switch |
| auto | else | operator | template |
| break | enum | private | this |
| case | extern | protected | throw |
| catch | float | public | try |
| char | for | register | typedet |
| class | friend | return | union |
| const | goto | short | unsigned |
| continue | if | signed | virtual |
| default | inline | sizeof | void |
| delete | int | static | volatile |
| do | long | struct | while |

### 2.1.2 Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

1. Only alphabetic characters, digits and underscores are permitted.
2. The name cannot start with a digit.
3. Uppercase and lowercase letters are distinct.
4. A declared keyword cannot be used as a variable name.

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, C++ places no limit on its length and, therefore, all the characters in a name are significant.

Care should be exercised while naming a variable that is being shared by more than one file containing C and C++ programs. Some operating systems impose a restriction on the length of such a variable name.



*Caution* The first character in an identifier must be a letter or the _ (underscore) character; however, beginning identifiers with an underscore is considered poor programming style.

## Self Assessment

Fill in the blanks:

1. The keywords not found in ANSI C are shown …….. .
2. A major difference between C and C++ is the limit on the length of a…….. .

## 2.2 Assignment Expression

Notes

The expression that makes uses of assignment operations is called as in assignment expression. Assignment expression assigns the value of an expression, a constant or a variable to an identifier. Assignment expressions are often referred to as assignment statements, since they are usually written as complete statement. The rules for writing assignment expressions are as follows:

1. Don't confuse the assignment (=) with the relational operator (==). The assignment operator is used to assign a value to an identifier, where as the equality operator is used to determine if two expressions have the same value. This,

```
        a = 1

and

        a == 1

are two different expressions.
```

2. The sign (=) equal to is an operator and not an equation maker, so it can appear anywhere in place of another operator. The following are legal assignment expressions.

```
a = b = c + 4;

a = 2 * (b = 10/c);
```

   C++ allows to use of multiple assignment operators in a series, for example,

```
a = b = c = 2;
```

   In such situation, the assignments are carried out from right to left. There are, however, restrictions to the level to which these assignments can be chained. For instance, Turbo C++ allows to chain maximum 70 assignment operators in a statement i.e.

```
v1 = v2 = v3 = ..... = v70 = 10;
```

   where v1, v2, v3 are variable of similar type and are assumed to be pre declared. When executed the value 10 will be assign to v70 and the value of v70 will be assign to v69 and so on. Finally the variable v1 will be assigned with the value 10 as the value assignment operation carried out from right to left.

3. With the compound assignment operators, compiler automatically supplies parentheses - explicit parenthesis around the expression which is entirely superfluous. For example, the following statement:

```
a + = b + 1;

is required to

a = a + (b+1);
```

4. If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the variable on the left.

5. Assignment operators have a lower precedence than any of the other operators. Therefore, various operations like unary operations, arithmetic operations, relational operations, equality operations and logical operations are all carried out before assignment operations.

Notes



*Example:*

```
#include <iostream>
using namespace std;
int main ()
{
  int a, b;          // a:?,  b:?
  a = 10;            // a:10, b:?
  b = 4;             // a:10, b:4
  a = b;             // a:4,  b:4
  b = 7;             // a:4,  b:7
   cout << "a:";
   cout << a;
   cout << " b:";
   cout << b;
   return 0;
}
```

### Self Assessment

Fill in the blanks:

3. Assignment expression assigns the value of an expression, a constant or a variable to an……..

## 2.3 Operators

An expression is a combination of variables, constants and operators written according to some rules. An expression evaluates to a value that can be assigned to variables and can also be used wherever that value can be used.

Individual constant, variables, array elements function references can be joined together by various operators to form expressions. C++ includes a large number of operators, which fall into several different categories. In this section we examine some of these categories in detail. Specifically, we will see how arithmetic operators; unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions.

The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands.

Operators are used to compute and compare values, and test multiple conditions. They can be classified as:

1. Arithmetic operators
2. Assignment operators

Notes

3. Unary operators
4. Comparison operators
5. Shift operators
6. Bit-wise operators
7. Logical operators
8. Conditional operators

### 2.3.1 Arithmetic Operators

There are five arithmetic operators in C++. They are

| Operator | Function |
|---|---|
| + | addition |
| – | subtraction |
| * | multiplication |
| / | division |
| % | remainder after integer division |

The % operator is sometimes referred to as the modulus operator.

There is no exponentiation operator in C++. However, there is a library function (pow) to carry out exponentiation. Alternatively you can write your own function to compute exponential value.

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set.). The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-pointing quotient.

Suppose that a and b are integer variables whose values are 8 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

| Expression | Value |
|---|---|
| a + b | 12 |
| a – b | 4 |
| a * b | 32 |
| a / b | 2 |
| a % b | 0 |

**Notes**



*Notes* Note that the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that a1 and a2 are floating-point variables whose values are 14.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

| **Expression** | **Value** |
|---|---|
| a1 + a2 | 16.5 |
| a1 – a2 | 12.5 |
| a1 * a2 | 29.0 |
| a1 / a2 | 7.25 |

Finally, suppose that x1 and x2 are character-type variables that represent the character M and U, respectively. Some arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

x1+x2 = 162

x1+x2+'5'=215

Note that M is encoded as (decimal) 77, U is encoded as 85, and 5 is encoded as 53 in the ASCII character set.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient.

The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C++ assign the sign of the first operand to the remainder. Thus, the condition

a = ((a/b) * b) + a % b)

will always be satisfied, regardless of the signs of the values represented by a and b.

Suppose that x and y are integer variables whose values are 12 and –2, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

| **Expression** | **Value** |
|---|---|
| x+y | 10 |
| x–y | 12 |
| x*y | –24 |
| x/y | –6 |
| x%y | 0 |

**Notes**

If x had been assigned a value of –12 and y had been assigned 2, then the value of x/y would still be –6 but the value of a x%y would be 0. Similarly, if x and y had both been assigned negative values (–12 and –2, respectively), then the value of x/y would be 3 and the value of x%y would be –2.

x = ((x/y)*y) + (x%y)

will be satisfied in each of the above cases. Most versions of C++ will determine the sign of the remainder in this manner, though this feature is unspecified in the formal definition of the language.

Here is an illustration of the results that are obtained with floating-point operands having different signs. Let y1 and y2 be floating-point variables whose assigned values are 0.70 and 3.50. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

| **Expression** | **Value** |
|---|---|
| y1+y2 | 2.72 |
| y1–y2 | –4.28 |
| y/y2 | –0.2728 |

Operands that differ in type may undergo type conversion before the expression takes on its final value. In general, the final result will be expressed in the highest precision possible, consistent with the data type of the operands. The following rules apply when neither operand is unsigned.

1. If both operands are floating-point types whose precisions differ (e.g., a float and a double), the lower-precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. Thus, an operation between a float and double will result in a double; a float and a long double will result in a long double; and a double and a long double will result in a long double. (Note: In some versions of C++, all operands of type float are automatically converted to double.)
2. If one operand is a floating-point type (e.g., float, double or long double) and the other is a char or an int (including short int or long int), the char or int will be converted to the floating-point type and the result will be expressed as such. Hence, an operation between an int and a double will result in a double.
3. If neither operand is a floating-point type but one is long int, the other will be converted to long int and the result will be long int. Thus, an operation between a long int and an int will result in a long int.
4. If neither operand is a floating-point type or a long int, then both operands will be converted to int (if necessary) and the result will be int. Thus, an operation between a short into and an int will result in an int.



*Task* What does the following expression evaluate to?

6 + 5 * 4 % 3

Notes

### 2.3.2 Assignment Operators

| Operator | Description | Example | Explanation |
|---|---|---|---|
| = | Assign the value of the right operand to the left | a = b | Assigns the value of b to a |
| += | Adds the operands and assigns the result to the | a +=b | Adds the of b to a<br>The expression could also be left operand written as a = a+b |
| –= | Subtracts the right operand from the left operand and stores the result in the left operand | a -=b | Subtracts b from a<br>Equivalent to a = a-b |
| *= | Multiplies the left operand stores the result in the left operand | a*=b | Multiplies the values a and b by the right operand and<br>and stores the result in a<br>Equivalent to a = a*b |
| /= | Divides the left operand stores the result in the left operand | a/=b | Divides a by b and stores the by the right operand and result in a Equivalent to a = a/b |
| *= | Divides the left operand stores the remainder in the left operand | a%=b | Divides a by b and stores the by the right operand and remainder in a<br>Equivalent to a = x%y |

Any of the operators used as shown below:

A <operator>=y

can also be represented as

a=a <operator> b

that is, b is evaluated before the operation takes place.

You can also assign values to more than one variable at the same time. The assignment will take place from the right to the left. For example,

a = b = 0;

In the example given above, first b will be initialized and then a will be initialized.



*Did u know?* **What is self assignment?**

Self assignment is when someone assigns an object to itself. For example,

```
#include "Fred.h"   // Defines class Fred

void userCode(Fred& x)

{

  x = x;             // Self-assignment

}
```

Notes

### 2.3.3 Unary Operators

| Operator | Description | Example | Explanation |
|---|---|---|---|
| ++ | Increases the value of the operand by one | a++ | Equivalent a = a+1 |
| – | Decreases the value of the operand by one | a- | Equivalent to a = a-1 |

The increment operator, ++, can be used in two ways - as a prefix, in which the operator precedes the variable.

```
++var;
```

In this form the value of the variable is first incremented and then used in the expression as illustrated below:

```
var1=20;
        var2 = ++var1;
```

This code is equivalent to the following set of codes:

```
        var1=20;
        var1 = var1+1;
        var2 = var1;
```

In the end, both variables var1 and var2 store value 21.

The increment operator ++ can also be used as a postfix operator, in which the operator follows the variable.

```
var++;
```

In this case the value of the variable is used in the expression and then incremented as illustrated below:

```
var1 = 20;
var2 = var1++;
```

The equivalent of this code is:

```
var1 = 20;
var2=var1;
var1 = var1 + 1;     // Could also have been written as var1 += 1;
```

In this case, variable var1 has the value 21 while var2 remains set to 20.

In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms.

If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the C++ statement, var2 = var1++; the original of var1 is assigned to var2. In the statement, var2 = ++var1++; the original value of var1 is assigned to var2. In the statement, var2 = ++var1; the incremented value of var1 is assigned to var2.

Notes



*Notes* The operators, '++' and '—', are best used in simple expressions like the ones shown above.

### 2.3.4 Comparison Operators

Comparison operators evaluate to true or false.

| Operator | Description | Example | Explanation |
|---|---|---|---|
| == | Evaluates whether the operands are equal. | A==b | Returns true if the values are equal and false otherwise |
| != | Evaluates whether the operands are not equal | a!=y | Returns true if the values are not equal and false otherwise |
| > | Evaluates whether the left operand is greater than the right operand | a>b | Returns true if a is greater than b and false |
| < | Evaluates whether the left operand is less than the right operand | a<b | Returns true if a is greater than or equal to b and false otherwise |
| >= | Evaluates whether the left operand is greater than or equal to the right operand | a>=b | Returns true if a is greater than or equal to b and false otherwise |
| <= | Evaluates whether the left operand is less than or equal to the right Operand | A<=b | Returns true if a is less than or equal to b and false otherwise. |



*Example:*

```
#include <iostream>
using namespace std;
int main ()
{
   int a, b=3;
   a = b;
   a+=2;               // equivalent to a=a+2
   cout << a;
   return 0;
}
```

Notes

### 2.3.5 Shift Operators

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. The following displays the binary representation of digits 0 to 8.

| Decimal | Binary Equivalent |
|---|---|
| 0 | 00000000 |
| 1 | 00000001 |
| 2 | 00000010 |
| 3 | 00000011 |
| 4 | 00000100 |
| 5 | 00000101 |
| 6 | 00000110 |
| 7 | 00000111 |
| 8 | 00001000 |

Shift operators work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, bool, float, or double data types.

| Operator | Description | Example | Explanation |
|---|---|---|---|
| >> | Shifts bits to the right, filling sign bit at the left | a=10 >> 3 | The result of this is 10 divided by 23. An explanation follows. |
| << | Shifts bits to the left, filling zeros at the right | a=10 << 3 | The result of this is 10 multiplied by 23. An explanation follows. |



*Example:*

```
#include <iostream>
using namespace std;
int main() {
    cout    << "5 times 2 is " << (5 << 1) << endl
            << "20 divided by 4 is " << (20 >> 2) << endl;
}
```

#### *Shifting Positive Numbers*

If the int data type occupies four bytes in the memory, the rightmost eight bits of the number 10 are represented in binary as

0 0 0 0 1 0 1 0

When you do a right shift by 3(10 >> 3), the result is

0 0 0 0 0 0 0 1

Notes

10/23, which is equivalent to 1.

When you do a left shift by 3 (10 << 3), the result is

0 1 0 1 0 0 0 0

10*23, which is equivalent to 80

*Shifting Negative Numbers*

For negative numbers, the unused bits are initialized to 1. Therefore, –10 is represented as:

1 1 1 1 0 1 1 0

### 2.3.6 Bit-wise Operators

| Operator | Description | Example | Explanation |
|---|---|---|---|
| & (AND) | Evaluates to a binary value after a bit-wise AND on the operands | a & b | AND results in a 1 if both the bits are 1, any other combination results in a 0 |
| ! (OR) | Evaluates to binary value after a two operands | a ! b | OR results in a 0 when both the bit-wise OR on the bits are 0, any other combination results in a 1. |
| ^ (XOR) | Evaluates to a binary value after a bit-wise XOR on the two operands | a ^ b | XOR results in a 0 if both the bits are of the same value and 1 if the bits have different values. |
| ~ | Converts all 1 bits to 0s and (inversion) | | Example given below. all 0 bits to 1s |

In the example shown in the table, a and b are integers and can be replaced with expressions that give a true or false (boo1) result.



*Example:* When both the expressions evaluate to true, the result of using the & operator is true. Otherwise, the result is false.

*The ~ Operator*

If you use the ~ operator, all the 1a in the bite are converted to 0s and vice versa. For example, 10011001 would become 01100110.

### 2.3.7 Logical Operators

Use logical operators to combine the results of Boolean expressions.

| Operator | Description | Example | Explanation |
|---|---|---|---|
| && | Evaluates to true, if both the conditions evaluate to true, false otherwise | a>6&&y<20 | The result is true if condition 1 (a>6) and condition 2 (y<20) are both true. If one of them is false, the result is false. |

Notes

| | | | |
|---|---|---|---|
| \|\| | Evaluate to true, if at least one of the conditions evaluates to true and false if none of the conditions evaluate to true. | a>6 \|\| y < 20 | The result is true if either condition1 (a>6) and condition2 (y<20) or both evaluate to true. If both the conditions are false, the result is false. |

### *Short Circuit Logical Operators*

These operators (&&, ||) appear to be similar to the bit-wise & and | operators, except that they are limited to Boolean expressions only. However, the difference lies in the way these operators work. In the bit-wise operators, both the expressions are evaluated. This is not always necessary since:

false & a would always result in false

true | a would always result in true

Short circuit operators do not evaluate the second expression if the result can be obtained by evaluating the first expression alone.

For example

a < 6 && y > 20

The second condition (b>20) is skipped if the first condition is false, since the entire expression will anyway, be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result, will anyway, be true. These operators, && and ||, are therefore, called short circuit operators.

*Notes* If you want both the conditions to be evaluated irrespective of the result of the first condition, then you need to use bit-wise operators.

## 2.3.8 Conditional Operators

| Operator | Description | Example | Explanation |
|---|---|---|---|
| (condition) va11, va12 | Evaluates to va11 if the condition returns true and va12 if the condition returns false | a = (b>c) ? b:c | A is assigned the value in b, if b is greater than c, else a is assigned the value of c. |

This example finds the maximum of two given numbers.

```
If (num1 > num2)
{
imax = num1;
}
else
{
imax = num2;
}
```

Notes

In the above program code, we determining whether num1 is greater than num2. The variable, imax is assigned the value, num1, if the expression, (num1 >num2), evaluates to true, and the value, num2, if the expression evaluates to false. The above program code can be modified using the conditional operator as:

Imax = (num1 > num2) ? num1 : num2;

The ?: Operator is called the ternary operator since it has three operands.

This example calculates the grade of a Ram based on his marks.

Int marks = 0;

Cout << "Please enter marks of the Ram;

Cin >> marks;

Char grade = (marks < 80), the variable, grade is assigned the value, 'A'. If Ram's score is less than or equal to 80 grade is assigned the value 'B'.

The following code will display either "PASSED" or "FAILED", depending on the value in the variable, score.

Cout << {score > 50? "PASSED" or FAILED, depending on the value in the variable, score.

Cout << (score > 50? "PASSED" : "FAILED"} << end1;

### Self Assessment

Fill in the blanks:

4. The data items that operators act upon are called……... .
5. A declared keyword cannot be used as a …….. name.
6. The operands acted upon by arithmetic operators must represent …… values.
7. An operation between a float and double will result in a…….. .
8. A …….. can have a value of one or zero.

## 2.4 Control Structures

C++ program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as branching. There is also a special kind of branching, called selection, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as looping. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C++. Several logical expressions are given below.

count < = 100

sqrt( x + y + z) > 0.00

answer = = 0

Notes

balance >= cutoff

ch1 < 'A'

letter1 = 'a'

The first four expressions involve numerical operands. Their meaning should be readily apparent. In the fifth expression, ch1 is assumed to be a char type variable. This expression will be true if the character represented by ch1 comes before T in the character set, i.e., if the numerical value used to encode the character is less than the numerical value used to encode the letter T.

The last expression makes use of the char-type variable letter. This expression will be true if the character represented by letter is something other than x.

In addition to the relational and equality operators, C++ contains two logical connectives (also called logical operators), && (AND) || (OR), and the unary negative operator !. The logical connectives are used to combine logical expressions, thus forming more complex expressions. The negation operator is used to reverse the meaning of a logical expression (e.g., from true to false).



*Task* Explain how control structures are a very useful programming idea.

### 2.4.1 The if-else Construct

The if conditional construct is followed by a logical expression in which data is compared and a decision is made based on the result of the comparison. The syntax is:

```
if (boolean_expr)
{
        statements;
}
else
{
        statements;
}
```

The if-else statement is used to carry out a logical test and then take on of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true of false).

The else portion of the if-else statement is optional. Thus, in its simplest general form, the statement can be written as:

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

The expression must be placed in parentheses, as shown. In this form, the statement will be executed only if the expression has a nonzero value (i.e., if expression is true).



*Notes* If the expression has a value of zero (i.e., if expression is false), then the statement will be ignored.

**Notes**

The statement can be either simple or compound. In practice, it is often a compound statement, which may include other control statements. For example, consider the following program:

```
#include <iostream.h>
int main()
{
char chr;
cout << "Please enter a character:" ;
cin >> chr;
if (chr == 'X')
cout << end1 << "The character is X";
else
cout << end1 << "The character is not X";
return 0;
}
```

Note that the operator, ==, used for comparing two data items, is different from the assignment operator, =,

In an if-else block of statements, the condition is evaluated first. If the condition is true (value is non-zero), the statements in the immediate block are executed, if the condition is false (value is zero) the statements in the else block are executed.

In the above example, if the input character is 'X', the message displayed is, 'the character is X' otherwise, the message, 'The character is not X' is displayed. Also, note that the condition has to be specified within parenthesis.

If, however, the operator, =, is used instead of the operator, ==, the statement is executed as an assignment. For example, if the statement, if (chr == 'X'), was written as, if (chr = 'X'), then, chr would have been assigned the value, 'X' and the condition would be evaluated as true. Thus, the message, the character is 'X' would have been displayed, regardless of the input.

This program could also have been written as given below:

```
#include <iostream.h>
int main()
{
char chr;
cout << "Please enter a character: ";
cin >> chr;
if (chr != 'X')
cout << end1 << "The character is no X";
else
cout << end1 << "The character is X";
return 0;
}
```

**Notes**

The operator, !=, is an equivalent of not equal to, and is written without any space between ! and =.

The general form of an if statement which includes the else clause is:

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

If the expression has a nonzero value (i.e., if expression is true), then statement1 will be executed. Otherwise (i.e., if expression is false), statement2 will be executed.



*Example:*

```
#include <stdio.h>
int main() {
    int x = 0;
    if ( 1 )    // if statement #1
    // {
        if ( !x )    // if statement #2
            printf_s("!x\n");
        else    // paired with if statement #2
            printf_s("x\n");
    // }
}
```

The following, for example, program uses a nested if...else construct to check if the input character is uppercase or lowercase.

```
#include <iostream.h>
int main()
{
        char inp;
        cout << "Please enter a character: ";
        cin >> inp;
        if (inp >= 'A')
                if (inp <= 'Z')
                        cout << end1 << "Uppercase";
                else if (inp >= 'a')
                {
                        if (inp <= 'z')
                                cout << end1 << "Lowercase";
                        else
                                cout << end1 << "Input character > z";
                }
```

**Notes**

```
                else
                {
                        cout << end1 << "input character > z but less than a";
                }
else
{
                cout << end1 << "Input character less than A";
}
return 0;
}
```

The following program shows how the clarity of steps can be increased by using braces ({and}) for enclosing parts of the code that follow an if or an else statement.

```
#include <inostream.h>
int main()
{
        char inp;
        cout << "Enter a character: ";
        cin >> inp;
        if (inp >= 'A')
        {
                if (inp <= 'z')
                {
                        cout << end1 << "Uppercase";
                }
                else if (inp >= 'a')
                {
                        if (inp <= 'z')
                        {
                                cout << end1 << "Lowercase";
                        }
                }
        }
        return 0;
}
if (circle) {
                scanf ( ' &f' ,   & radius);
                area = 3. 14159 * radius   * radius;
```

**Notes**

```
                printf ( * Area of circle = & f* , area);
}
else
{
                scanf ( '&f      & f", & length , & width) ;
                area = length * width;
                printf   ( * Area of rectangle = &f ', area);
}
```

The above example shows how an area can be calculated for either of two different geometric figures. If circle is assigned a nonzero value, however, then the length and width of a rectangle are read into the computer, the area is calculated and then displayed. In each case, the type of geometric figure is included in the label that accompanies the value of the area. Here is an example of three nested if-else statements.

```
If (   ( time > = 0.) && ( time < 12.) ) printf ( * Good Morning*) ;
else if ( ( time > = 12.) && ( time < 18.) ) printf ( * Good Afternoon *);
else if ( ( time  > = 18.) && ( time < 24.)) printf (   * Good Evening *) ;
else printf   ( * Time is out of range *);
```

This example causes a different message to be displayed at various times of the day. Specifically, the message Good Morning will be displayed if time has a value between 0 and 12; Good Afternoon will be displayed if time has a value between 12 and 18; and Good Evening will be displayed if time has a value between 18 and 24. An error message (Time is out of range) will be displayed if the value of time is less than zero, or greater than or equal to 24.

The following program uses a cascading if-else construct to determine if the input character is a vowel, else it prints an appropriate message.

```
# include < iostream.h>
int main ( )
{
        char in_chr;
        cout << " Enter a character in lowercase " < , endl;
        cin >> in_chr;
        if ( in_chr == 'e')
                cout << endl << " Vowel    a" << endl;
        else if ( in_chr == 'e')
                cout   <, endl   << "Vowel       e" <, endl;
        else if ( in_chr == ' i )
                cout   << endl << " Vowel    i " << endl);
        else if ( in_chr == ' o   ' )
                cout <, endl << " Vowel   o " << endl;
        else if ( in_chr == 'u')
```

Notes

```
                cout <, endl <<   " Vowel u" << endl;
        else
                cout << endl << " The character is not a Vowel" << endl;
        return o;
}
```

## 2.4.2 Looping

The while statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.

The general form of the while statement is:

```
while (expression) statement;
```

The statement will be executed repeatedly, as long as the expression is true (i.e., as long expression has a nonzero value). This statement can be simple or compound, through it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition for the loop.



*Did u know?* **What is the common error while using the while loop?**

A common error when using the while loop is to forget to do anything that might change the value being tested on while statement from within the loop. If the variable being tested is not changed within the loop then the condition will either never be true and the loop will therefore never get executed or it will always be true and will execute forever.

### *Consecutive Integer Quantities*

In this example, we try to display the consecutive digits 0,1,2, . . . . 9.

```
# include < iostream.h >
main ( )              / * display the integers   0 through 9 * /
{
        int digit =0;
        while ( digit < = 9)
{
                cout<<digit<<endl;
                ++ digit;
}
}
```

Initially, a digit is assigned a value of 0. The while loop then displays the current value digit, increase its value by 1 and then repeats the cycle, until the value of digit exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning and ending with 9. Thus, when the program is executed, the following output will be generated.

0 **Notes**

1

2

3

4

5

6

7

8

9

This program can be written more conspicuously as:

```
#   include  <  iostream.h>
main  ( )                    / * display the integers 0 through 9 */
{
        int digit = 0;
        while (digit < =   9) {
                cout<<digit<<endl;
                digit + = 5;
}
```

when executed, this program will generate the same output as the first program.

The following code generates the Fibonacci series between 1 and 100. In this series, each number is the sum of its two preceding numbers. The series starts with 1.

```
# include < iostream .h>
int main ( )
{
        int num1 = 1,   num2 = 1;
        cout << num1 << endl;
        while ( num2 < 100)
        {
                cout << num2 << endl;
                num2 += num1;
                num1 = num2 - num1;
        }
        return 0;
}
```

**Notes**

Output

1

1

2

3 As you can see, each number in the series is the

4 sum of the two preceding numbers.

5

8

13

21

34

55

89

### *The Break Statement*

The break statement causes the program flow to exit the body of the while loop. The following program code illustrates the use of the break statement.

```
#include <iostream.h>
int main()
{
        int num1 = 1, num2 = 1;
        cout << num1 << end1;
        while (num2 < 150)
        {
                cout << num2 << end1;
                num2 += num1;
                num1 = num2 - num1;
                if (num2 == 89)
                break
        }
        return 0;
}
```

The output of the previous program is:

1

1

2

Notes

3

5

8

13

21

34

55

The control exits the loop when the condition, num2 == 8 9, becomes true.



*Task* Break statement is used with the conditional switch statement and with the do, for, and while loop statements. Explain how with an example.

### *The do-while Statement*

Sometimes, however, it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the do-while statement.

The general form of the do-while statement is:

```
do
statement
while (expression);
```

The statement will be executed repeatedly, as long as the value of expression is true (i.e., is nonzero). Notice that the statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statement can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of expression so the looping action can terminate.

Here is another example to do the same thing, using the to display the consecutive digits (0, 1, 1, 2, 9) using the do-while loop.

```
#include <studio.h>
main()              /* display the integers 0 through 9 */
{
        int digit = 0;
        do {
        printf ("%d\n", digit++);
        }while (digit <= 9);
}
```

This program below accepts characters from keyboard until the character, 'I', is entered and displays whether the total number of consonant characters entered is more than, less than, or equal to the total number of vowels entered.

**Notes**

```
# include <iostream.h>
int main ( )
{
        int constant, vowel ;
        char inp ;
        consonant = vowel 0 ;
        do
        {
                inp = ' ;
                cout << endl. << " Enter a character ( ! to quit ) ";
                cin >> inp;
                switch ( inp)
                {
                        case 'A'      :
                        case   'a'    :
                        case 'E'      :
                        case 'e'      :
                        case 'I'            :
                        case 'i'            :
                        case 'O'      :
                        case 'o'      :
                        case 'U'      :
                        case 'u'      :     vowel   = vowel +   1;
                                            break ;
                        case ' !'     :     break;
                        default       :     consonant = consonant + 1;
                }
        } while ( inp ! = '!')
        if ( constant > vowel )
        cout << endl << " Consonant count greater than vowel count" ;
        else if ( consonant < vowel )
                cout << endl < " Vowel count greater than consonant count";
        else
                 cout << endl << " Vowel ! and consonant counts are equal ";
        return 0;
}
```

Notes

The do-while loop displays the current value of digit, increases its value by 1, and then tests to see if the current value of digit exceeds 9. If so, the loop terminates, otherwise, the loop continues, using the new value of digit. Note that the test is carried out at the end of each pass through the loop. The net effect is that the loop will be repeated 10 times, resulting in 10 successive lines of output.

For example, let us see the calculation for the average of numbers:

```
/ * calculate the average of n number */
# include < stdio. h>
main ( )
{
        int n,count   = 1;
        float x, average, sum = 0;
        /* initialize and  read in a value for n */
        printf ( "How many numbers ?");
        scanf ( "%d" , %n);
        /* read  in the numbers 8/
        do {
                printf ( 8 x = 8) ;
                scanf ( "% f , %x);
                sum += x;
                ++ count ;
        } while ( count < = n);
        /* calculate the average and display the answer*/
        average = sum /n
        printf (" \ n The average is &f \n ",average);
}
```



*Notes* One interesting thing about the while loop is that if the loop condition is false, the while loop may not execute at all.

### *The for Statement*

The for statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of the for statement is

for (expression-1; expression-2; expression-3) statement;

Notes

where expression-1 is used to initialize some parameter (called an index) that controls the looping action, expression-2 represents a condition that must be true for the loop to continue execution, and expression-3 is used to alter the value of the parameter initially assigned by expression-1. Typically, expression-1 is an assignment expression, expression-2 is a logical expression and expression-3 is a unary expression or an assignment expression.

When the for statement is executed, expression-2 is evaluated and tested at the beginning of each pass through the loop, and expression-3 is evaluated at the end of each pass. Thus, the for statement is equivalent to

Expression-1;

```
while  (expression-2)
{
        statement
        expression-3;
}
```

The looping action will continue as long as the value of expression-2 is not zero, that is as long as the logical condition represented by expression-2 is true.

The program will display the consecutive duals using for loop.

```
# include < stdio. h>
main ( )                    /*   display the numbers 0 through 9*/
{
        int  digit
        for  ( digit = 0 , digit < = 9; + + digit)
                printf  ( " % d\n" , digit);
}
```

The first line of the for statement contains three expression, enclosed in parentheses. The first expression assigns an initial value 0 to the integer variable digit; the second expression continues the looping action as long as the current value of digit does not exceed 9 at the beginning of each pass, and the third expression increases the value of digit by 1 at the end of each pass through the loop.



*Task* The for statement is ideal when we know exactly how many times we need to iterate, because it lets us easily declare, initialize, and change the value of loop variables after each iteration. Analyze and explain with an example.

## Self Assessment

Fill in the blanks:

9. The …….. is used to carry out a logical test and then take on of two possible actions, depending on the outcome of the test.
10. The …… statement is used to carry out operations, in which a group of statements is executed repeatedly, till a condition is satisfied.

**Notes**

11. The …… statement causes the program flow to exit the body of the while loop.

12. The do-while loop displays the …….. value of digit, increases its value by 1.

## 2.5 Scope Resolution Operator

C++ is also a block-structured language. We know that the same variable name can be used to have different meanings in different blocks. The scope of a variable extends from the point of its declaration till the end of the block, containing the declaration. A variable declared inside a block is said to be local to that block. Blocks in C++ are often nested. For example the following style is common:

```
. . . . .
. . . . .
{
        int  x=10;
        . . . . .
        . . . . .
{
            int x=1;
            . . . . .
            . . . . .
}
. . .
. .
}
```

The declaration in an inner block hides a declaration of the same variable in the outer block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following format:

```
::variable_name
```

This operator allows access to the global version of a variable.

Program illustrating the use of :: is given below:

```
# include <iostream.n>
int  m=10;            //globalm
main()
{
int  m=20             //m is redeclared; local to main
        {
   int n=m;
   int m=30;   //m is declared again
   cout << "we are in innerblock \n";
```

Notes

```
    cout << "n=" <<n<< "\n";
    cout << "m=" <<m<< "\n";
    cout << "::m=" <<::m<< "\n";
   }
    cout << "\n we are in outer block \n";
    cout << "m=" <<m<< "\n";
    cout << "::m=" <<::m<< "\n";
}
```

The output of the program:

We are in inner block.

n = 20

m = 30

::m = 10

We are in outer block

m = 20

::m = 10



*Did u know?* **What is the use of scope resolution operator?**

The :: (scope resolution) operator is used to qualify hidden names so that you can still use them.

## 2.6 Member Dereferencing Operators

Once a class (or a structure) is defined, its members (both data and methods/function) can be accessed (or referenced) using two operators - (.) dot operator and (->) arrow operator. While (.) operator takes class or struct type variable as operand, (->) takes a pointer or reference variable as its operand.

With class and structure type variables, (.) dot operator is used to reference the members, as shown in the following program code:

```
struct  abc
{       int a, b, c;  };
class  pqr
{       int p, q, r;
        int  doit();   };

abc  structone;
pqr  classone;
structone.a  =  20;
structone.b  =  120;
```

Notes

```
structone.c = 30;
classone.p = 56;
classone.q = 16;
classone.r = 51;
classone.doit();
```

Evidently, each member can be referenced by using (.) dot operator. However, (.) operator does not work on pointer or reference type variables, as is shown in the following code snippet.

```
abc *ptrAbc;
pqr *ptrPqr;
abc abcone;
pqr pqrone;
ptrAbc = abcone;     //pointer variable ptrAbc points to object abcone
ptrAbc->a;
ptrPqr->doit();
```



*Notes* It is possible to access the value of variables pointed by the pointer variables using pointer. This is performed by using the Dereference operator in C++ which has the notation *.

### 2.6.1 Memory Management Operators

Along with malloc(), calloc() and frec() functions, C++ also defines two unary operators new and delete that perform the task of allocating and freeing the memory. Since these operators manipulate memory on the free store, they are also known as free store operators. A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete.

The general form of the new operator

```
pointer_variable = new data_type;
```

The pointer_variable holds the address of the memory space allocated.

```
p = new int;
q = new float;
```

We can also initialize the memory using the new operator. This is done as follows:

```
pointer_variable= new data_type(value);
Value specifies the initial value as shown below:
int*p = new int(25);
```

When an object is no longer needed, it is destroyed to release the memory space for reuse. The general form is:

```
delete pointer_variable;
```

Notes

For example,

```
delete p;
delete q;
```

If we want to free a dynamically allocated array we must use the following form of delete.

```
delete [size]pointer_variable;
```

For example, statement

```
delete [ ]p;
```

will delete entire array pointed by p.

The new operator has several advantages over the function malloc().

1. It automatically computes the size of the data objects.
2. It automatically returns the correct pointer type so there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. They both it new and delete can be overloaded.



*Did u know?* **What is the need for Memory Management operators?**

The concept of arrays has a block of memory reserved. The disadvantage with the concept of arrays is that the programmer must know, while programming, the size of memory to be allocated in addition to the array size remaining constant.

In programming there may be scenarios where programmers may not know the memory needed until run time. In this case, the programmer can opt to reserve as much memory as possible, assigning the maximum memory space needed to tackle this situation. This would result in wastage of unused memory spaces. Memory management operators are used to handle this situation in C++ programming language.

## 2.6.2 Declaration of Variables

If you are familiar with C, you would know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program.

C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually come across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope, sometimes far away from the place of declaration.

Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type it is. C++ allows the declaration of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes programs much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use. The following example illustrates this point.

```
main ( )
{
        float x;      // declaration
```

Notes

```
        float sum = 0;
        for (int i = 0; i<5; i++)   //declaration
        {
                cin >> x;
                sum = sum+x;
        }
        float average;              //declaration
        average = sum / i;
        cout << average;
}
```



*Did u know?* **What is the disadvantage of this style of declaration?**

The only disadvantage of this style of declaration is that we cannot see at a glance all the variables used in a scope.

### 2.6.3 Dynamic Initialization of Variables

One additional feature of C++ is that it permits initialization of the variables at run time. This is referred to as dynamic initialization. Remember that, in C, a variable must be initialized using a constant expression and the C compiler would fix the initialization code at the time of compilation. However, in C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

…..

…..

int n = strlen(string);

…..

float area = 3.14159 *rad *rad;

This means that both the declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The two statements in the following example of the previous section

```
float average;              // declare where it is necessary
average = sum / i;
```

can be combined into a single statement:

```
float average = sum /i;     // initialize dynamically
// at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed using information that is known only at the run time.

Notes

### 2.6.4 Reference Variables

C++ introduces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data_type & reference_name = variable_name
```



*Example:*

```
float total = 100;

float &sum = total;
```

total is a float type variable that has already been declared, sum is the alternative name declared to represent the variable total. Both the variables refer to the same data object in the memory. Now, the statements

```
cout << total;
```

and

```
cout << sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object that it names. Note that the initialization of a reference variable is completely different from assignment.

Note that C++ assigns additional meaning to the symbol &. Here, & is not an address operator.

The notation float & means reference to float. Some more examples are presented below to illustrate this point:

int n[10];

int &x = n[10]; //x is alias for n[10]

char &a = '\n'; // initialize reference to a literal

The variable x is an alternative to the array element n[10]. The variable a is initialized to the new line constant. This creates a reference to the otherwise unknown location where the new line constant \n is stored.

The following references are also allowed:

1. int x;

   int *p=&x;

   int &m = *p;

2. int &n = 50;

Notes

The first set of declarations causes m to refer to x which is pointed to by the pointer p and the statement in (2) creates an int object with value 50 and name n.

A major application of reference variables is when passing arguments to functions. Consider the following code snippet:

```
void f(int &x)       // uses reference
{
       x = x+10;     // x is incremented; so also m
}
main ( )
 {
       int m = 10;
       f(m);  // function call
       ….
       …
}
```

When the function call f(m) is executed, the following initialization occurs:

```
Int &x = m;
```

Thus x becomes an alias of m after executing the statement f(m);.

Since the variable x and m are aliases, when the function increments x, m is also incremented. The value of m becomes 20 after the function is executed. In traditional C, we accomplish this operation using pointers and dereferencing techniques.

The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference and eliminates the copying of object parameters back and forth.



*Notes* Note that the references can be created not only for built-in data types but also for user-defined data types such as structures and classes. References work wonderfully well with these user-defined data types.

## Self Assessment

Fill in the blanks:

13. The scope of a variable extends from the point of its declaration till the end of the……..., containing the declaration.
14. (.) operator does not work on …….. or reference type variables.
15. A reference variable provides an alias (alternative name) for a previously defined……... .

Notes



Caselet

## Getting IT Right

**VRML**

Virtual Reality Modeling Language or Virtual Reality Markup Language is used in creating 3-dimensional objects for a virtual reality environment. For example, it could be used to create a prototype of moving objects such as the planets in space on a screen. In response to the mouse click, the VRML objects are triggered to move across the screen and create a flow of movement.

VRML objects can generate 2D and 3D animation, sound effects and other visual effects that can be inserted as objects in other interfaces such as HTML and Java programs.

The file extension used for VRML is ".wrl". VRML is designed to be used on the Internet, intranets and other local client systems. VRML can be used for engineering and scientific visualisation, multimedia presentations as well as entertainment and educational titles. For example, VRML can be used to create 3D designs for auto manufacturers.

It can also combine other commonly used 3D objects, and translate them to VRML file format. VRML objects can be reused without having to be recoded.

**OOP**

Object Oriented Programming is a technique that considers methods and data as objects. Other techniques such as process-oriented models deal with methods and data as independent entities, whereas OOP emphasises the packing of methods and data together. T he method is a set of instructions that is are executed on the given data to achieve a result. For example, in a program which adds two numbers and prints the result, the method is addition and data is the input of 2 numbers.

In OOP the objects can be reused. In the OOP technique the methods are wrapped together in a container called class and used whereever the program requires it. A class contains the definition of how the objects should behave. OOP manages complexity by way of inheritance, encapsulation and polymorphism.

Encapsulation is the wrapping of both methods and data together as an object. The object is protected to prevent manipulation by other parts of the program.

When an object acquires the properties of another object, i.e., an object obtains the properties of all the methods and interfaces for execution that are created in another object, it is called inheritance.

Polymorphism implies that the same interface can be used for different classification of uses. An example of polymorphism is when a set of information has to be modified in three different files viz, inserting into file number one, changing the contents in file 2 and deleting the information in file 3.

The OOP technique works best when encapsulation, inheritance and polymorphism are combined and work together. Some of the popular versions that are available in market are C++, Java, etc.

**Accelerator Board**

The accelerator board is an additional fit-in for the motherboard to boost the speed of the computer. The motherboard will have a socket to insert any CPU or FPU (floating

Contd...

Notes

point unit). The floating point unit is a specially designed chip which can calculate the floating point values (number with decimal values) much faster than usual. A FPU can handle all graphical applications much faster than any other chip.

The latest computers provide scope for further and higher upgradation. If a motherboard has a provision for ZIF socket, the upgradation is much easier. Zero Insertion Force (ZIF) is a ready-made socket that allows one to insert and remove any chip set without additional tools to support.

The co-processor is a supporting chip fitted in with the CPU to assist in specific types of operations. This will reduce the burden on the CPU when very complex mathematical and scientific calculations have to be done. For instance, the Math Co-processor (MC) will assist the CPU in faster processing of applications that have floating point values. The computer should have a co-processor supporting program that calls the MC to make C recognise floating point calculations. Otherwise the co-processor will not recognise the instructions.

To boost the speed of the system, all the co-processors can be fitted to the motherboard.

## 2.7 Summary

- C++ language is made up of letters, words, sentences and constructs just like English language.
- A collection of characters, much like a word in English language are called tokens.
- A token can be a keyword, or identifier, constant, string or an operator.
- Keywords are the reserve words that cannot be used as names of variable or other user-defined program elements.
- Identifiers refer to the names of variables, functions, arrays, classes etc. created by the programmer, Basic types.
- An array represents named list of finite number of similar data elements. A function is a named part of a program that can be invoked from the other parts of the program.
- A pointer is a variable that holds a memory address of another variable. A reference is an alternative name for an object. A constant is a data item whose data value can never change during the program run.
- Operators are the symbols that represent specific operations. Arithmetic operators are unary +, Unary–, +, –, *, / and % . '%' uses pure integer division thereby requires integer operands.
- Comparison operators, also called relational operators, compare the relationships among values. The order of evaluation among logical operators is NOT, AND, OR, i.e.!, &&, 11.
- Along with mallo(), calloc() and frec() functions, C++ also defines two unary operators, new and delete, that perform the task of allocating and freeing the memory.

## 2.8 Keywords

***Character Constant:*** One or more characters enclosed in single quotes.

***Expression:*** A combination of variables, constants and operators written according to some rules.

***Identifies:*** The names of variables, functions, arrays, classes, etc. created by the programmer.

Notes

***Keywords:*** Explicitly reserved identifiers that cannot be used as names for the program variables or other user defined program elements.

***Operands:*** The data items that operators acted upon all called operands.

***Scope Resolution Operator:*** This operator enables a program to access a global variable when a local variable of the same name is in scope.

***String Constant:*** Sequence of characters enclosed in double quotes.

***Tokens:*** A collections of characters, much live a word in English language. The smallest individual unit in a program.

## 2.9 Review Questions

1. Are 'break' and 'continue' bad programming practices? Analyze.
2. Does the last element in a loop deserve a separate treatment? Why or why not?
3. Which useful alternative control structures do you know? Explain with an example.
4. Does C++ provide any trinary operator? If yes, explain its working.
5. Convert the following while loop into for loop.

```
int i = 10;
while(i < 20)
{
     cout << i;
     (i++)++;
}
```

6. In control structure switch-case what is the purpose of default in C++?
7. Write a C++ program to display and add five numbers.
8. What is the difference between dynamic and reference declaration of variables?
9. What are operators? What are their functions? Give examples of some unary and binary operators.
10. What are keywords? Can keywords be used as identifiers?

### Answers: Self Assessment

1. Boldface
2. Name
3. Identifier
4. Operands
5. Variable
6. Numeric
7. Double
8. Bit
9. If-else statement
10. While
11. Break
12. Current
13. Block
14. Pointer
15. Variable

## 2.10 Further Readings

Notes



*Books*

E. Balagurusamy, *Object-oriented Programming through C++*, Tata McGraw Hill.

Herbert Schildt, *The Complete Reference C++*, Tata Mc Graw Hill.

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.



*Online links*

http://www.cplusplus.com/doc/tutorial/operators/

http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

**Notes**

# Unit 3: Review of Functions

**CONTENTS**

Objectives

Introduction

3.1 The MAIN Function

3.2 Function Overloading

3.3 Inline Functions

3.4 Default Arguments

3.5 Function Prototyping

3.6 Summary

3.7 Keywords

3.8 Review Questions

3.9 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the functions
- Describe the function overloading
- Explain the inline functions
- Discuss the default arguments
- Describe the function prototyping

## Introduction

Those who are familiar with language C would agree that writing a C program is nothing more than writing C functions (including the main function). C++ on the other hand is all about writing codes for defining and manipulating classes and objects.

Conceptually an object may have only data members specifying its attributes. However, such an object would serve no useful purpose. For the purpose of establishing communication with the object it is necessary that the object provide methods, which are C like functions. Though C++ functions are very similar to C functions, yet they differ significantly as you will discover in this unit.

## 3.1 The MAIN Function

An application written in C++ may have a number of classes. One of these classes must contain one (and only one) method called main method. Although a private main method is permissible in C++ it is seldom used. For all practical purposes the main method should be declared as public method.

Notes

The main method can take various forms as listed below:

```
main()

main(void)

void main()

void main(void)

int main()

int main(void)

int main(int argc, char *argv[])

main(int argc, char *argv[])

void main(int argc, char *argv[])
```

As is evident from the above forms, return type of the main method specifies the value returned to the operating system once the program finishes its execution. If your main method does not return any value to the operating system (caller of this method), then return type should be specified as void.



*Caution* In case you design your main method in such a way that it must return a value to the caller, then it must return an integer type value and therefore you must specify return type to be int.

### Self Assessment

Fill in the blanks:

1. A function is a group of ……. that is executed when it is called from some point of the program.
2. The execution of the program starts from the function …….. .
3. The …… specifies the type of the data the function returns.
4. The …… list could be empty which means the function do not contain any parameters.
5. A function declaration is same as the declaration of the……... .

## 3.2 Function Overloading

A function may take zero or more arguments when called. The number and type of arguments that a function may take is defined in the function itself. If a function call fails to comply by the number and type of argument(s), the compiler reports the same as error.

Suppose we write a function named sum to add two numerical values given as arguments. One can write the function as:

```
int sum(int a, int b)

{

        return (a + b);

}
```

**Notes**

Now suppose we want the function to take float type argument then the function definition must be changed as:

```
float  sumfloat(float  a,  float  b)
{
        return  (a  +  b);
}
```

As a matter of fact the function sum may take so many names as shown below.

```
int  sumint(int  a,  int  b)
{
        return  (a  +  b);
}
short  sumshort(short  a,  short  b)
{
        return  (a  +  b);
}
long  sumlong(long  a,  long  b)
{
        return  (a  +  b);
}
float  sumdouble(double  a,  double  b)
{
        return  (a  +  b);
}
```

This can be very tiring and extremely difficult to remember all the names. Function overloading is a mechanism that allows a single function name to be used for different functions. The compiler does the rest of the job. It matches the argument numbers and types to determine which functions is being called. Thus we may rewrite the above listed functions using function overloading as:

```
int  sum(int  a,  int  b)
{
        return  (a  +  b);
}
float  sum(float  a,  float  b)
{
        return  (a  +  b);
}
short  sum(short  a,  short  b)
{
        return  (a  +  b);
```

Notes

```
}
long sum(long a, long b)
{
        return (a + b);
}
float sum(double a, double b)
{
        return (a + b);
}
```

Overloaded functions have the same name but different number and type of arguments. They can differ either by number of arguments or type of arguments or both. However, two overloaded function cannot differ only by the return type.



*Task* In overloaded functions, the function call determines which function definition will be executed. Explain with an example.

### Self Assessment

Fill in the blanks:

6. Function overloading is a feature of C++ that allows us to create ……. functions with the same name, so long as they have different parameters.
7. Overloaded functions have the same name but different number and…… .
8. If a function call fails to compile by the number and type of argument(s), the compiler reports the same as…….. .
9. Function overloading can lower a programs …….. significantly while introducing very little additional risk.
10. Member function declarations with the same name and the name parameter-type-list cannot be …….. if any of them is a static member function declaration.

## 3.3 Inline Functions

C++ provides facility of inline function, which is designed to speed up the program execution. This is done by the C++ compiler, which incorporates it into a program.

When any program is compiled the output of compilation is a set of machine language instructions, which is in executable program. When a program is run, this complied copy of program is put into memory. Each instruction gets a memory address, which is used to refer that instruction. These instructions are executed as per the logic in the program. Instructions are executed step by step unless there are any loop or branching instructions. When there is branch or jump instruction, some instructions are skipped and forward or backward jump is made.

In normal function call, the control of execution is transferred to the location where the function is loaded in memory that is the starting executable instruction of the function. When all the instructions of the function are executed the control returns back to main program from where left.

**Notes**

Now the next instruction in the main program will get executed. Jumping back and forth keeping track of where to jump means that there is an overhead in elapsed time to functions. When the function is declared inline the compiler replaces the function call with the corresponding function code. As a result the program does not have to spend time to jump to another location to execute the function and then come back. But there is a memory overhead. If the function is called n times in the program then it is copied n times making the program code or in particular executable code large. If the value of n is large, there is a lot of memory overhead.



*Task* How do you tell the compiler to make a non-member function inline?

The following program demonstrates the use of inline function to calculate square of an input number.

```
#include <iostream.h>
#include <conio.h>
inline float square(float a)
{
retum (a * a);
}
void main()
{
float b;
cout <<" Enter a number";
cin >> b;
a = square(b); .
              cout << "\nThe square of " << b << "is " << a;
}
```

You should see the output as shown below.

Enter a number 1.2

The square of 1.2 is 1.44

Actually, compiler compiles the code by inserting the function definition at the place where it is called as shown below.

```
#include <iostream.h>
#include <conio.h>
inline float square(float a)
{
retum (a * a);
}
void main()
{
```

Notes

```
float b;
cout <<" Enter a number";
cin >> b;
a = b * b;          //instead of square(b); .
              cout << "\nThe square of " << b << "is " << a;
}
```

Since, the code will be coded wherever it is called, the program size increases and hence requires more memory space.



*Notes* However, because no function calling overhead is involved, execution becomes fast.

### Self Assessment

Fill in the blanks:

11. Inline functions are functions where the ……. is made to inline functions.
12. The inline function takes the format as a …….. but when it is compiled it is compiled as inline code.

## 3.4 Default Arguments

C++ functions can have arguments having default values. The default values are given in the function prototype declaration. Whenever a call is made to a function without specifying an argument, the program automatically assigns values to the parameters as specified in the default function prototype declaration.

Default arguments facility allows easy development and maintenance of programs .To establish a default value you have to use the function prototype. Because the compiler looks at the prototype to see how many arguments a function uses, the function prototype also has to alert the program to the possibility of default arguments. The method is to assign a value to the argument in the prototype. The following program explains the use of default arguments:

```
#include <iostream.h>
void rchar (char ch = '*', int num = 10);
//default arguments with values '*' and 10
void main (void)
{
rchar ( );
              rchar ('=');
rchar ('+', 30);
}
       void rchar (char ch, int num)
{
```

**Notes**

```
for (int j = 0; j < num; j ++)
                cout << ch;
                cout << endl;
}
```

The function rchar() is called 10 times in the main( ) function with different number of arguments passed to it. This is allowed in C++.

The first function call to rchar( ) takes no arguments in the main( ) function. The function declaration provides default values for the two arguments required by rchar(). When values are not provided, the compiler supplies the defaults '*' and 10.

In the second call, one argument is missing which is assumed to be the last argument. rchar() assigns the single argument '=' to the ch parameter and uses the default value 10 for num.

In the third case, both arguments are present and no defaults are considered. A few points worth remembering here.

The missing or default arguments must be the trailing arguments, those at the end of the argument list. The compiler will flag an error when you leave something out. Default arguments may be useful when an argument has the same value.

```
       void  abc (int a, float b, int c = 10, char d = 'P'); //correct
defaulting!
```

The above declaration is correct because no un-defaulted arguments appear after defaulted arguments. For this reason the following declaration is incorrect.

```
       void  abc (int a, float b = 2.5, int c, char d = 'P'); //incorrect
defaulting!
```



*Did u know?* **How to use two default parameter values?**

```
#include <iostream>
 int f(int length, int width = 25, int height = 1);
 int main()
 {
      int length = 100;
      int width = 50;
      int height = 2;
      int area;

       area = f(length, width, height);
       std::cout << "First time area equals " << area << "\n";
       area = f(length, width);
       std::cout << "Second time area equals " << area << "\n";
       area = f(length);
       std::cout << "Third time area equals " << area << "\n";
```

Notes

```
    return 0;
}
int f(int length, int width, int height)
{
    return (length * width * height);
}
```

## 3.5 Function Prototyping

Prototype of a function is the function without its body. The C++ compiler needs to about a function before you call it, and you can let the compiler know about a function is two ways – by defining it before you call it or by specifying the function prototypes before you call it. Many programmers prefer a 'Top-Down' approach in which main() appears ahead the user-defined function definition. In such approach, function access will precede function definition. To overcome this, we use the function prototypes or we declare the function. Function-prototype are usually written at the beginning of a program, ahead of any programmer-defined functions including main(). In general, function prototype is written as:

```
Return-type name (type 1 arg. 1, type 2 arg. 2, ….. type n arg. n);
```

Where return-type represents the data type of the value returned by the function, name represents the function name, type 1, type 2 …. type n represents the data-type of arguments art. 1, arg. 2 … arg n. The function prototypes resemble first line to function definition, though it ends with the semicolon. Within the function declaration, the names of arguments are optional but data-types are necessary for eq. Int count (int);

The code snippet given below will cause compilation error because the main function does not know about the called function one().

```
void main()
{
    …
    one();        //incorrect!! No function prototype available for one()
    …
}
void one()
{
    //function definition
}
```

To resolve this problem you should define the function before it is called as shown below:

```
void one()
{
    //function definition
}
void main()
```

Notes

```
{
        …
        one();       //Correct!! Function one is defined before calling
        …
}
```

Another elegant alternative to this problem is to include a function prototype before it called no matter where the function has been actually defined as shown below:

```
void one(void);           //prototype of function one()
void main()
{
        …
        one();       //Correct!! Function one is defined before calling
        …
}
void one()
{
              //function definition
}
```



*Did u know?* **What are function signatures?**

The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature.

## Self Assessment

Fill in the blanks:

13. A default parameter is a function parameter that has a …….. value provided to it.
14. Default arguments facility allows easy development and …….. of programs.
15. Function-prototype are usually written at the …… of a program.



*Caselet*

### IEEE Tutorial on Advanced Computing

The IEEE Computer Society and IEEE Kerala Section have announced a one-day tutorial in the series, "Frontiers in computing practice: AOP and JML - Novel programming paradigms for the Java programmer."

*Contd...*

Notes

The tutorial will be held from 9:30 a.m. on Saturday, July 29, at the Amphitheater of the Centre for Development of Advanced Computing (C-DAC) here. Dr Venkatesh Choppella and Mr Satish Babu will anchor the session, an IEEE spokesman said.

The last decade has seen Java transition into a mature, powerful, well-supported language capable of building scalable applications for the enterprise. Much intellectual effort has gone into exploring extensions to the core Java language.

Two of these extensions are gradually gaining ground: Aspect/J, the Java extensions for Aspect-Oriented Programming (AOP), and Java Modeling Language (JML). Although completely different in nature, both these extensions demonstrate the power of the core Java language. Also, both are completely compatible with traditional Java.

**Relevance of AOP**

Both procedural programming and object-oriented programming focuses on separation and encapsulation of concerns. Some software concerns, however, defy easy encapsulation. These are called 'cross-cutting' concerns because they exist in many parts of the program.

For example, a logging strategy affects every single part of the system. Logging thereby crosscuts all classes, methods, and procedures. Other crosscutting concerns include security and debugging.

AOP allows "weaving in" of code to implement crosscutting concerns across existing classes and methods through a construct called an "aspect." Aspects permit associating an "advice" (code snippets) to "join points" (weave-in points in existing methods where the advice should be invoked).

For example, for an application written without security in mind, a security advice can retrofit user authentication just before every method call in a class. The aspect is completely separate, and the original source file is unmodified.

**Versatile JML**

Design-by-contract (DBC) provides a formal mechanism to ensure the correct working of a software procedure or method. At the minimum, DBC requires that preconditions, post-conditions, invariants and side effects can be tested (through mechanisms such as assertions).

JML provides a way to describe contracts in a Java-compatible manner, by embedding JML statements as special comments within Java source files. This way, JML is totally non-intrusive, and at the same time, provides much more rigour to the module by enforcing DBC.

## 3.6 Summary

An application written in C++ may have a number of classes. One of these classes must contain one (and only one) method called main method. Although a private main method is permissible in C++ it is seldom used. For all practical purposes the main method should be declared as public method. A function may take zero or more arguments when called. The number and type of arguments that a function may take is defined in the function itself. If a function call fails to comply by the number and type of argument(s), the compiler reports the same as error. When any program is compiled the output of compilation is a set of machine language instructions, which is in executable program. When a program is run, this complied copy of program is put into memory. C++ functions can have arguments having default values. The default values are given in the function prototype declaration. Prototype of a function is the function without its body. The C++ compiler needs to about a function before you call it, and you can let the compiler

Notes

know about a function is two ways – by defining it before you call it or by specifying the function prototypes before you call it.

## 3.7 Keywords

***Function:*** The best way to develop and maintain a large program is to divide it into several smaller program modules of which are more manageable than the original program. Modules are written in C++ as classes and functions. A function is invoked by a function call. The function call mentions the function by name and provides information that the called function needs to perform its task.

***Function Declaration:*** Statement that declares a function's name, return type, number and type of its arguments.

***Function Overloading:*** In C++, it is possible to define several function with the same name, performing different actions. The functions must only differ in their argument lists. Otherwise, function overloading is the process of using the same name for two or more functions.

***Function Prototype:*** A function prototype declares the return-type of the function that declares the number, the types and order of the parameters, the function expects to receive. The function prototype enables the compiler to verify that functions are called correctly.

***Inline Function:*** A function definition such that each call to the function is, in effect, replaced by the statements that define the function.

***Reference:*** Alias name of a variable.

***Return:*** The C++ keyword return is one of several means we use to exit a function. When the return 0 statement is used at the end of main function, the value 0 indicates that the program has terminated successfully or returns NULL value to the operating system.

***Scope:*** The program part(s) in which a function or a variable is accessible.

## 3.8 Review Questions

1. What is function prototyping? Why is it necessary? When is it not necessary?
2. What is purpose of inline function?
3. Differentiate between the following:
   (a) void main()
   (b) int main()
   (c) int main(int argn, char argv[])
4. In which cases will declare a member function to be friend?
5. Write a program that uses overloaded member functions for converting temperature from Celsius to Kelvin scale.
6. To calculate the area of circle, rectangle and triangle using function overloading.
7. Do inline functions improve performance?
8. How can inline functions help with the tradeoff of safety vs. speed?

Notes

9. Prototype of a function is the function without its body. Analyze and explain with an example.
10. How can you create a C++ function f(int,char,float) that is callable by my C code?

### Answers: Self Assessment

| | | | |
|---|---|---|---|
| 1. | Statements | 2. | Main( ) |
| 3. | Return_type | 4. | Parameter |
| 5. | Variable | 6. | Multiple |
| 7. | Type of arguments | 8. | Error |
| 9. | Complexity | 10. | Overloaded |
| 11. | Call | 12. | Normal function |
| 13. | Default | 14. | Maintenance |
| 15. | Beginning | | |

## 3.9 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://www.cplusplus.com/doc/tutorial/functions/

http://cpp-tutorial.cpp4u.com/structures_functions.html

**Notes**

# Unit 4: Classes and Objects

**CONTENTS**

Objectives

Introduction

4.1 Specifying a Class

4.2 Defining Member Functions

4.3 Creating Class Objects

4.3.1 Objects as Function Arguments

4.3.2 Returning Objects

4.4 Accessing a Member of Class

4.5 Access Specifiers

4.5.1 The Public Access Specifier

4.5.2 The Private Access Specifier

4.5.3 The Protected Access Specifier

4.6 Summary

4.7 Keywords

4.8 Review Questions

4.9 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize how to specify a class
- Define the member functions
- Explain the creation of class objects
- Access the class members
- Discuss the access specifiers

## Introduction

Classes and objects are at the core of object-oriented programming. Writing programs in C++ essentially means writing classes and creating objects from them. In this unit you will learn to work with the same.

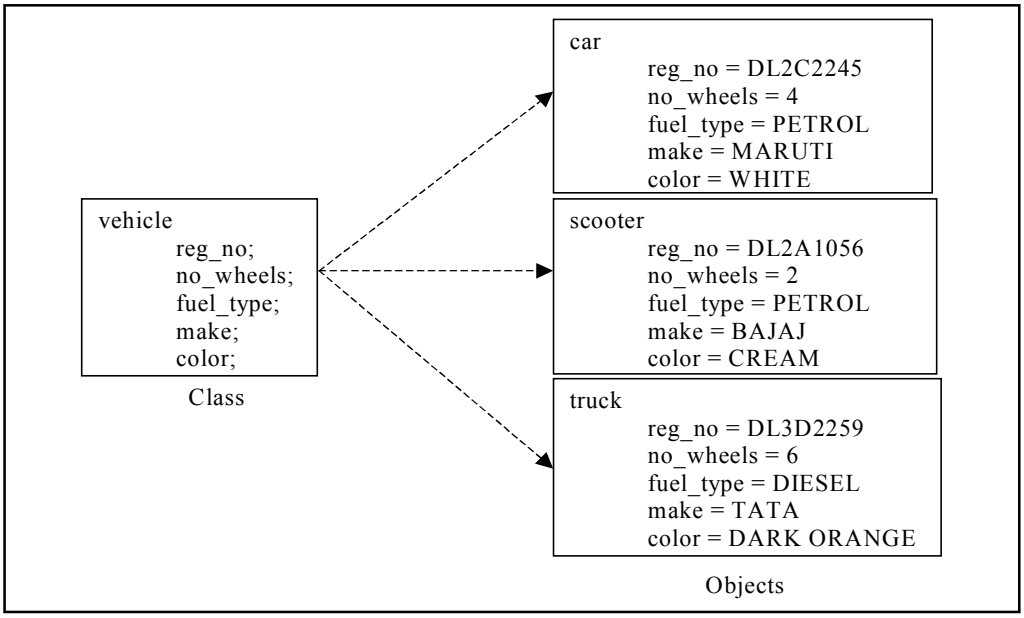
It is important to note the subtle differences between a class and an object, here. A class is a template that specifies different aspects of the object it models. It has no physical existence. It occupies no memory. It only defines various members (data and/or methods) that constitute the class.

An object, on the other hand, is an instance of a class. It has physical existence and hence occupies memory. You can create as many objects from a class once you have defined a class.

Notes

You can think of a class as a data type; and it behaves like one. Just as a data type like int, for example, does not have a physical existence and does not occupy any memory until a variable of that type is declared or created; a class also does not exist physically and occupies no memory until an object of that class is created.

*Example:* To understand the difference clearly, consider a class of vehicle and a few objects of this type as depicted below:




In this example vehicle is a class while car, scooter and truck are instances of the class vehicle and hence are objects of vehicle class. Each instance of the class vehicle – car, scooter and truck – are allocated individual memory spaces for the variables – reg_no, no_wheels, fuel_type, make and, color – so that they all have their own copies of these variables.

## 4.1 Specifying a Class

Like structures a class is just another derived data-type. While structure is a group of elements of different data-type, class is a group of elements of different data-types and functions that operate on them. C++ structure can also have functions defined in it.

There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data-type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions.

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data-type that can be treated like any other built-in data-type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

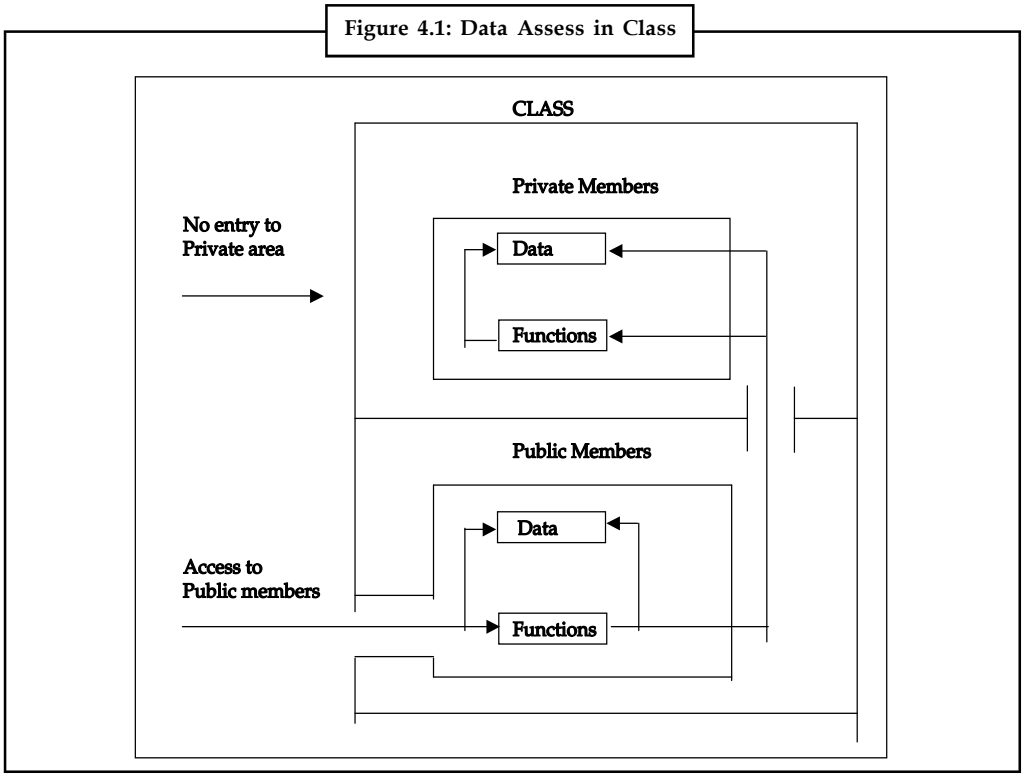
**Notes**

```
class <class_name>
{
private:
        variables declaration;
        function declarations;
public:
        variable declaration;
        function declarations;
};
```

The class declaration is similar to a struct declaration. The keyword class specifies that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called members. They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords private and public are known as visibility labels.

The members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are private. If both the labels are missing, then, by default, all the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

**Figure 4.1: Data Assess in Class**




The variables declared inside a class are known as data members and the functions are known as member functions. Only the member functions can have access to the private data members and

Notes

private functions. However, the public members (both functions and data) can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as encapsulation. The access to private and public members of a class is well explained diagrammatically in the Figure 4.1.

Let us consider the following declaration of a class for student:

```
class  student
{
private:
        int  rollno;
        char  name  [20];
public:
        void  getdata(void);
        void  disp(void);
};
```

The name of the class is student. With the help of this new type identifier, we can declare instances of class student. The data members of this class are int rollno and char name [20]. The two function members are getdata() and disp(). Only these functions can access the data members. Therefore, they provide the only access to the data members from outside the class. The data members are usually declared as private and member functions as public. The member functions are only declared in the class. They shall be defined later.



*Task* The binding of data and functions together into a single class-type variable is referred to as encapsulation. Do you agree with this statement? Why or why not?

A class declaration for a machine may be as follows:

```
class  machine
        {
                int  totparts,  partno;
                char  partname  [20];
        public:
                void  getparts  (void);
                void  disp(part_no);
        };
```

Having defined the class, we need to create object of this class. In general, a class is a user defined data type, while an object is an instance of a class. A class provides a template, which defines the member functions and variable that are required for objects of a class type. A class must be defined prior to the class declaration.

The general syntax for defining the object of a class is:

```
class  <class_name>
{
```

**Notes**

```
        private:
                //data
                //  functions
        public:
                //functions
};
```

<class_name> object1, object2,… objectN;

where object1, object2 and objectN are the instances of the class <class_name>.

A class definition is very similar to a structure definition except the class definition defines the variables and functions.

Consider the following program segment which declares and creates a class of objects.

```
class  student
{
private:
        int  rollno;
        int  age;
        float  height;
        float  weight;
public:
        void  getinfo( );
        void  disinfo( );
        void  process( );
        void  personal( );
};
```

student std; //std is the object of the class student

In another example the employee details such as name. code, designation, address, salary age can be grouped as follows.

```
class  employee
{
private:
        char  name[20];
        int  code;
        char  designation[20];
        char  address[30];
        float  salary;
        int  age;
public:
```

Notes

```
        void salary( );
        void get_info( );
        void display_info( );
};
```

employee x, y; //creates x and y - two objects of the class employee

By now it should be clear to you that you can use a class just as you use a data type. In fact you can also create an array of objects of a particular class.



*Did u know?* **What is the difference between defining a class member function completely within its class or to include only the prototype and later its definition?**

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

### Self Assessment

Fill in the blanks:

1. A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and…….. .
2. By default, all members of a class declared with the …….. keyword have private access for all its members.
3. All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called ……... .
4. A class provides a template, which defines the member functions and …… that are required for objects of a class type.
5. A class must be defined prior to the class ……... .

## 4.2 Defining Member Functions

We have learnt to declare member functions. Let us see how member functions of a class can be defined within a class or outside a class.

A member function is defined outside the class using the :: (double colon symbol) scope resolution operator. The general syntax of the member function of a class outside its scope is:

```
<return_type> < class_name>:: <member_function>(arg1, arg2....argN)
```

The type of member function arguments must exactly match with the types declared in the class definition of the <class_name>. The Scope resolution operator (::) is used along with the class name in the header of the function definition. It identifies the function as a member of a particular class. Without this scope operator the function definition would create an ordinary function, subject to the usual function rules of access and scope.

**Notes**

The following program segment shows how a member function is declared outside the class declaration.

```
class  sample
{
private:
        int  x;
        int  y;
public:
        int  sum  ();                  // member  function  declaration
};
int  sample:: sum  ( ) //member  function  definition
{
        return  (x+y);
}
```



*Notes* Please note the use of the scope operator double colon (::) is important for defining the member functions outside the class declaration.

Let us consider the following program snippet:

```
class  first
{
private:
        int  x;
        inty;
public  :
        int  sum();
};
class  second
{
private:
        intx;
        int  y;
public:
        int  sum();
};
first  one;
```

Notes

```
second two;
int sum()
//error, scope of the member function is not defined
{
        return (x+y);
}
```

Both classes in the above case are defined with the same member function names. While accessing these member function, it gives an error. The scope of the member function sum( ) is not defined. When accessing the member function sum( ), control will be transferred to both classes one and two. So the scope resolution operator (::) is absolutely necessary for defining the member functions outside the class declaration.

```
int one:: sum( )     // correct
{
        return (x+y);
}
```

### Self Assessment

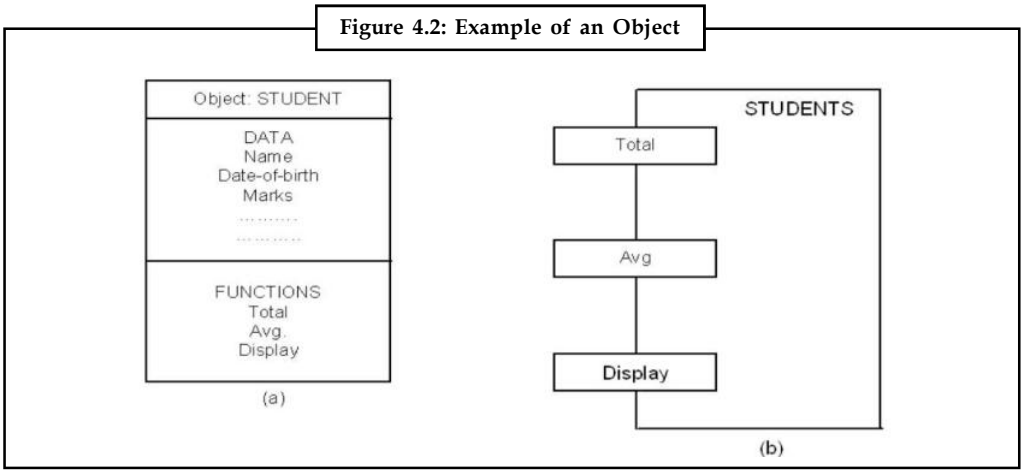
Fill in the blanks:

6. Member functions is only declared inside the class but …… outside the class.
7. The type of member function arguments must exactly match with the types declared in the…….. .
8. Only …….. identifies the function as a member of a particular class.
9. Without this scope operator, the function definition would create an ordinary function, subject to the usual function rules to access an…….. .

## 4.3 Creating Class Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data; they may also represent user-defined data such as vectors, time and lists.

**Figure 4.2: Example of an Object**



Notes

They occupy space in memory that keeps its state and is operated on by the defined operations on the object. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other data or code.

### 4.3.1 Objects as Function Arguments

Like any other data type argument, objects can also be passed to a function. As you know arguments are passed to a function in two ways:

1. by value
2. by reference

Objects can also be passed as arguments to the function in these two ways. In the first method, a copy of the object is passed to the function. Any modification made to the object in the function does not affect the object used to call the function. The following Program illustrates the calling of functions by value. The program declares a class integer representing a integer variable x. Only the values of the objects are passed to the function written to swap them.

```
#include<iostream.h>
#include<conio.h>
class  integer
{
       int  x;
public:
       void  getdata()
{
              cout  <<  "Enter  a  value  for  x";
              cin  >>  x;
}
       void  disp()
{
              cout  <<  x;
}
       void  swap(integer  a1  ,  integer  a2)
{
              int  temp;
              temp  =  a2.x;
              a2.x  =  al.x;
              a1.x  =  temp;
}
};
main()
{
```

Notes

```
        integer int1, int2;
        int1.getdata();
        int2.getdata();
        cout <<"\n the value of x belonging to object int1 is ";
        int1.disp();
        cout <<"\n the value of x belonging to object int2 is ";
        int2.disp();
        integer int3; .
int3.swap(int1, int2); //int3 is just used to invoke the function
        cout <<" \nafter swapping ";
        cout <<"\n the value of x belonging to object intI is ";
        int1.disp();
        cout <<"\n the value of x belonging to object int2 is ";
        int2.disp() ;
        cout<< "\n";
        getche();
}
```

You should get the following output.

Enter a value for x 15

Enter a value for x 50

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

after swapping

the value of x belonging to object int is 15

the value of x belonging to object int2 is 50

In the second method, the address of the object is passed to the function instead of a copy of the object. The called function directly makes changes on the actual object used in the call. As against the first method any manipulations made on the object inside the function will occur in the actual object.



*Caution* The second method is more efficient as only the address of the object is passed and not the entire object.

The following Program illustrates calling of a function by reference. The program declares a class integer to represent the variable x and defines functions to input and display the value. The function written to swap the integer values of the object takes the addresses of the objects.

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

**Notes**

```
class integer
{
        int x;
public:
        void getdata()
        {
                cout << "Enter a value for x";
                cin >> x;
}
void disp()
{
                cout << x;
}
        void swap(integer *a1)
{
                int temp;
                temp = x;
                x = a1->x;
                a1->x = temp;
}
};
main()
{
        integer int1, int2;
        int1.getdata();
        int2.getdata();
        cout <<"\n the value of x belonging to object int1 is ";
        int1.disp();
        cout <<"\n the value of x belonging to object int2 is ";
        int2.disp();
        int1.swap(&int2);
        cout << "\n after swapping ";
        cout <<"\n the value of x belonging to object int1 is ";
        int1.disp();
        cout <<"\n the value of x belonging to object int2 is ";
        int2.disp();
        cout << "\n";
        getche();
}
```

Notes

You should see the following output.

Enter a value for x 15

Enter a value for x 50

the value of x belonging to object int1 is 15

the value of x belonging to object int2 is 50

after swapping

the value of x belonging to object int1 is 50

the value of x belonging to object int2 is 15

### 4.3.2 Returning Objects

Just as a function takes an object as its argument, it can also return an object. The following program illustrates how objects are returned. The program declares a class integer representing an integer variable x and defines a function to calculate the sum of two integer values. This function finally returns an object which stores the sum in its data member x.

```
#include<iostream.h>
#include<conio.h>
class  integer
{
int  x;
public:
void  getdata(int  x1)
{
       x  =  x1;
}
       void  disp()
       {
              cout  <<  x;
}
integer  sum(integer  int2)
{
              integer  int3;
              int3.x  =  x  +  int2.x;
              return(int3);
}
};
main()
{
```

Notes

```
        integer int1, int2, int3;
        int1.getdata(15);
        int2.getdata(25);
        cout<<"\n the value of x for object int1 ";
        int1.disp();
        cout<<"\n the value of x for object int2 ";
        int2.disp(); .
cout<<"\n the sum of private data values of x belonging to objects int1 and
int2 is ";
        int3 = int1.sum(int2);
        int3.disp();
        getche();
}
```

You should see the following output from the program.

the value of x for object int1 15

the value of x for object int2 25

the sum of private data values of x belonging to objects int1 and int2 is 40



*Did u know?* **What is Returning a Reference to a const object?**

Though the main reason for using const reference is efficiency, there are restrictions on when this choice could be used. If a function returns an object that is passed to it, either by object invocation or as a method argument, we can increase the efficiency of the method by having it pass a reference.

### Self Assessment

Fill in the blanks:

10. You can use a class type to create instances or …….. of that class type.
11. In C++, unlike C, you do not need to precede declarations of class objects with the keywords union, struct, and class unless the …….. of the class is hidden.

## 4.4 Accessing a Member of Class

Member data items of a class can be static. Static data members are data objects that are common to all objects of a class. They exist only once in all objects of this class. The static members are used when the information is to be shared. They can be public or private data. The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory.

When a static member is declared private, the non-member functions cannot access these members. But a public static member can be accessed by any member of the class. The static data member should be created and initialized before the main function control block begins.

Notes



*Example:* Consider the class account as follows:

```
class account
{
private:
        int acc _no;
        static int balance;                //static data declaration
public:
        void disp(int acc _no);
        void getinfo( );
};
```

The static variable balance is initialized outside main() as follows:

int account::balance = 0; //static data definition

Consider the following Program which demonstrates the use of static data member count. The variable count is declared static in the class but initialized to 0 outside the class.

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

class counter
{
private:
static int count;
public:
void disp();
};
int counter::count = 0;
void counter::disp()
{
        count++
        cout << "The present value of count is " << count << "\n";
}
main()
{
        counter cntl ;
        for(int i=0; i<5; i++)
                cnt1.disp();
        getche();
```

Notes

```
}
```

You should get the following output from this program.

The present value of count is 1

The present value of count is 2

The present value of count is 3

The present value of count is 4

The present value of count is 5



*Task* Analyze how the static data member should be created and initialized before the main function control block begins.

## 4.5 Access Specifiers

Members of a class can access other members (properties and methods) of the same class. Functions, operators and other classes (corresponding objects) outside the class description of a particular class can also access members of that class. An access specifier decides whether or not a function or operator or class, outside the class description can access the members it controls inside its class description. The members an access specifier controls are the members typed under it in the class description (until the next specifier).

We will use functions and classes in the illustrations of accesses to class members. We will not use operators for the illustrations.

We will be using the phrase, external function. This refers to a function or class method that is not a member of the class description in question. When we say an external function can access a class member, we mean the external function can use the name (identifier of property or name of method) of the member as its argument or as an identifier inside its definition.



*Notes* Note that there is no specific order required for member access, as shown in the preceding example. The allocation of storage for objects of class types is implementation dependent, but members are guaranteed to be assigned successively higher memory addresses between access specifiers.

### 4.5.1 The Public Access Specifier

With the public access specifier, an external function can access the public members of the class. The following code illustrates this (read the explanation below):

```
#include <iostream>

using namespace std;

class Calculator

{

      public:

         int num1;
```

Notes

```
        int num2;
        int add()
            {
                 int sum = num1 + num2;
                 return sum;
            }
    };
     int myFn(int par)
        {
             return par;
        }
int main()
    {
         Calculator obj;
         obj.num1 = 2;
         obj.num2 = 3;
         int result = obj.add();
         cout << result; cout << "\n";
         int myVar = myFn(obj.num1);
         cout << myVar;

         return 0;
    }
```

There are two functions in the code: myFn() and main(). The first line in the main function instantiates a class object called, obj. In main, lines 2 and 3 use the properties of the class as identifiers. Because the class members are public, the main() function can access the members of the class. Line 4 of the main function also demonstrates this. In line 6 of the main function, the function, myFn() uses the property num1 of the class as its argument. It could do so because the member, num1 is public in the class.

### 4.5.2 The Private Access Specifier

With the private access specifier an external function cannot access the private members of the class. With the private specifier only a member of a class can access the private member of the class. The following code shows how only a member of a class can access a private member of the class (read the explanation below):

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

**Notes**

```
        private:
        int num1;
        int num2;

        public:
        int add()
            {
                num1 = 2;
                num2 = 3;
                 int sum = num1 + num2;
                 return sum;
            }
    };

int main()
    {
         Calculator obj;
         int result = obj.add();
         cout << result;

         return 0;
    }
```

The class has two private members (properties) and one public member (method). In the class description, the add() method uses the names of the private members as identifiers. So the add() method, a member of the class has accessed the private members of the class.

The main function definition (second line) has been able to access the add() method of the class because the add() method is public (it has a public access specifier).

The following code will not compile because the main function tries to access (use as identifier) a private member of the class:

```
#include <iostream>
using namespace std;
class Calculator
    {
        private:
        int num1;
        int num2;
        public:
        int add()
            {
```

Notes

```
                num1;
                num2 = 3;
                int sum = num1 + num2;
                return sum;
            }
    };
int main()
    {
         Calculator obj;
         obj.num1 = 2;
         int result = obj.add();
         cout << result;
         return 0;
    }
```

The second line in the main function is wrong because at that line, main tries to access (use as identifier) the private member, num1.

### 4.5.3 The Protected Access Specifier

If a member of a class is public, it can be accessed by an external function including a derived class. If a member of a class is private, it cannot be accessed by an external function; even a derived class cannot access it.

The question is, should a derived class not really be able to access a private member of its base class (since the derived class and base class are related)? Well, to solve this problem you have another access specifier called, protected. If a member of a class is protected, it can be accessed by a derived class, but it cannot be accessed by an external function. It can also be accessed by members within the class. The following code illustrates how a derived class can access a protected member of a base class:

```
#include <iostream>
using namespace std;
class Calculator
    {
         protected:
         int num1;
         int num2;
    };
class ChildCalculator: public Calculator
    {
         public:
         int add()
```

**Notes**

```
                {
                     num1 = 2;
                     num2 = 3;
                      int sum = num1 + num2;
                      return sum;
                }
     };
int  main()
     {
            ChildCalculator  myChildObj;
           int  result  =  myChildObj.add();
          cout << result;
          return 0;
     }
```

The base class has just two properties and no method; these properties are protected. The derived class has one method and no property. Inside the derived class, the protected properties of the base class are used as identifiers. Generally, when a derived class is using a member of a base class, it is a method of the derived class that is using the member, as in this example. The above code is OK.

The following code will not compile, because line 2 in the main() function tries to access a protected member of the base class:

```
#include  <iostream>
using  namespace  std;
class  Calculator
     {
          protected:
          int num1;
          int num2;
     };
class  ChildCalculator:  public  Calculator
     {
          public:
          int add()
              {
                   num1;
                   num2 = 3;
                    int sum = num1 + num2;
```

Notes

```
                return sum;
            }
    };
int  main()
    {
         Calculator obj;
         obj.num1 = 2;
          ChildCalculator myChildObj;
          int result = myChildObj.add();
         cout << result;;
         return 0;
    }
```

An external function cannot access a protected member of a class (base class); however, a derived class method can access a protected member of the base class.

*Notes* A member of a class can access any member of the same class independent of whether the member is public, protected or private.

You should now know the role of the access specifiers: public, protected and private as applied to classes. In one of the following parts of the series, we shall see the role of the access specifiers in the declarator of a derived class.

A public member of a class is accessible by external functions and a derived class. A private member of a class is accessible only by other members of the class; it is not accessible by external functions and it is not accessible by a derived class. A protected member of a class is accessible by a derived class (and other members of the class); it is not accessible by an external function.

## Self Assessment

Fill in the blanks:

12. The access-specifier determines the access to the names that follow it, up to the next access-specifier or the end of the class…… .
13. A public member of a class is accessible by …….. functions and a derived class.
14. If a member of a class is……, it cannot be accessed by an external function; even a derived class cannot access it.
15. With the …….. access specifier, an external function can access the public members of the class.

Notes



Caselet

## Change your Design, Midway

Changes happen all the time in a building project - the client asks for changes, the consultant asks for changes, you consider changes as you move the project through its later stages... there is no end to it. This could have been a mind-boggling issue some years ago, but may not be any more.

In the early 80s, when architects started using Computer-Assisted Drawing (CAD), the traditional layer-drafting technique was easily adapted to the layer-based CAD systems of the day. Within a few years, most construction documents and shop drawings were plotted from computers.

The use of CAD files was evolving towards communicating information about a building in ways that a plotted drawing could not. Forward thinking design firms adopted these tools, realising that the data in the object-oriented CAD files, if carefully structured and managed, could be used to automate certain documentation tasks.

But, object-oriented CAD systems remained rooted to building graphics, which were built on graphics-based CAD foundations. This resulted in the system not being optimised for creating and managing information about a building.

Another generation of purpose-built software solutions was required - a software that was both information-centric and provided building information modelling in place of building graphic modelling.

Architects, for instance, work on the information using the conventional graphic language of building design (such as plan, section and elevation), entering and reviewing information in a format that looks just like the architectural drawings they have worked with, for years. The fact remains that these people work on the building information through drawing rather than working directly on a drawing in the computer.

There was a need for a powerful building design and documentation systems for architects, interior designers, design-build teams and other building industry professionals.

Autodesk Inc, in the building design space, has introduced what it calls revolutionary software - Autodesk Revit - the parametric building modeller that is expected to capture all information about the design, even while the work is on.

**How is Revit different from the AutoCAD software?**

Imagine you are halfway through construction documents. You see a change that would make the project look even better. The Autodesk Revit parametric change engine, Autodesk sources say, will automatically coordinate the changes made anywhere - in model views or drawing sheets, schedules, sections, plans... . You name it. 'This engine supports all phases of the building process, preserving all information from beginning to end', say the sources.

This change management is said to be one of the fundamental characteristics of a building information modelling solution.

Because the coordination is assured by the system, embarrassing errors are minimised, and there's savings in time and needless costs in trying to hunt down the source and fix the problem.

Contd...

Notes

Revit is, therefore, considered an integrated building industry solution model, they say.

However, this change engine modeller is found to be more economical for large projects since such projects require careful organisation and management.

What about the others? This is left to be seen, considering the number of project developers who operate on a very small scale in India.

With the evolution of this design technology, all buildings could, in course of time, be represented as databases, not drawings! Just wait and see although it can take a long time coming... .

## 4.6 Summary

- A class represents a group of similar objects. A class in C++ binds data and associated functions together.
- It makes a data type using which objects of this type can be created. Classes can represent the real-world object which have characteristics and associated behavior.
- While declaring a class, four attributes are declared: data members, member functions, program access levels (private, public, and protected,) and class tag name.
- While defining a class its member functions can be either defined within the class definition or outside the class definition. The public member of a class can be accessed outside the class directly by using object of this class type.
- Private and protected members can be accessed with in the class by the member function only. The member function of a class is also called a method. The qualified name of a class member is a class name: : class member name.
- A global object can be declared only from a global class whereas a local object can be declared from a global as well as a local class.
- The object are created separately to store their data members. They can be passed to as well as returned from functions. The ordinary members functions can access both static as well as ordinary member of a class.

## 4.7 Keywords

***Class:*** Represents the real-world objects which have characteristics and associated behaviour.

***Global Class:*** A class whose definition occurs outside the bodies of all functions in a program. Objects of this class type can be declared from anywhere in the program.

***Local Class:*** A class whose definition occurs inside a function body. Objects of this class type can be declared only within the function that defines this class type.

***Private Members:*** Class members that are hidden from the outside world.

***Public Members:*** Class members (data members and member functions) that can be used by any function.

***Temporary Object:*** An anonymous short lived object.

Notes

## 4.8 Review Questions

1. Define class. What is the difference between structures and classes in C++?
2. How can you pass an object of a C++ class to/from a C function?
3. Can a method directly access the non-public members of another instance of its class?
4. What's the difference between the keywords struct and class?
5. Write a program to add two numbers defining a member getdata () and display () inside a class named sum and displaying the result.
6. How does C++ help with the tradeoff of safety vs. usability?
7. "The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory". Justify your statement.
8. Write a program to print the score board of a cricket match in real time. The display should contain the batsman's name, runs scored, indication if out, mode by which out, bowler's score (overs played, maiden overs, runs given, wickets taken). As and when a ball is thrown, the score should be updated.

   (print: Use separate arrays to store batsmen's and bowler's information)
9. If a member of a class is public, it can be accessed by an external function including a derived class. Explain with an example.
10. Write a program in which a class has one private member (properties) and two public member (method).

### Answers: Self Assessment

1. functions
2. class
3. access specifier
4. variable
5. declaration
6. defined
7. class definition
8. scope resolution operator
9. scope
10. objects
11. name
12. declaration
13. external
14. private
15. public

## 4.9 Further Readings

Notes



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://msdn.microsoft.com/en-us/library/h05xxt5d(v=vs.80).aspx

http://www.cplusplus.com/forum/beginner/6713/

**Notes**

# Unit 5: Static Members

**CONTENTS**

Objectives

Introduction

5.1 Static Member

5.1.1 Static Member Functions

5.2 The Const Keyword

5.2.1 The Class Keyword

5.3 Static Objects

5.4 Friend Functions

5.4.1 Friend Function in Empty Classes

5.4.2 Friend Function in Nested Classes

5.4.3 Friend Function in Local Classes

5.5 Summary

5.6 Keywords

5.7 Review Questions

5.8 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the static members
- Describe the const and class keyword
- Explain the static objects
- Discuss the friend function

## Introduction

Static data members of a class in namespace scope have external linkage. Static data members follow the usual class access rules, except that they can be initialized in file scope. Static data members and their initializers can access other static private and protected members of their class. The initializer for a static data member is in the scope of the class declaring the member.

A friend function is a function that is not a member of a class but has access to the class's private and protected members.

## 5.1 Static Member

Member data items of a class can be static. Static data members are data objects that are common to all objects of a class. They exist only once in all objects of this class. The static members are used when the information is to be shared. They can be public or private data. The main advantage

**Notes**

of using a static member is to declare the global data which should be updated while the program lives in memory.

When a static member is declared private, the non-member functions cannot access these members. But a public static member can be accessed by any member of the class. The static data member should be created and initialized before the main function control block begins.

For example, consider the class account as follows:

```
class account
{
private:
        int acc _no;
        static int balance; //static data declaration
public:
        void disp(int acc _no);
        void getinfo( );
};
```

The static variable balance is initialized outside main() as follows:

```
int account::balance = 0;    //static data definition
```

Consider the following Program which demonstrates the use of static data member count. The variable count is declared static in the class but initialized to 0 outside the class.

```
#include <iostream.h>
#include <conio.h>
class counter
{
private:
        static int count;
public:
        void disp();
};
int counter::count = 0;
void counter::disp()
{
        count++
        cout << "The present value of count is " << count << "\n";
}
main()
{
        counter cntl ;
        for(int i=0; i<5; i++)
                cnt1.disp();
        getche();
}
```

Notes

You should get the following output from this program.

The present value of count is 1

The present value of count is 2

The present value of count is 3

The present value of count is 4

The present value of count is 5



*Did u know?* **What is the use of static members?**

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created.

### 5.1.1 Static Member Functions

Just as data members can be static, member functions can also be static. A static function is a member function of a class and it manipulates only static data member of the class. It acts as global for members of its class without affecting the rest of the program. Static members serve the purpose of reducing the need for global variables by providing alternatives that are local to a class.

A static member function is not a part of objects of class. It is instance dependent and can be accessed directly by using the class name and scope resolution operator. If it is declared and defined in a class, the keyword static should be used only in declaration part.

The following program demonstrates the use of static member function which accesses static data. The data member count and the function disp( ) are declared static in the class counter. But the variable count is initialized to 0 outside the class.

```
#lnclude <iostream.h>
#lnclude <conin.h>
class counter
{
private:
        static int count;
public:
        counter()
        {
                ++count;
        }
        static void disp();
};
int counter::count=0;
void counter::disp()
```

Notes

```
{
        cout << count << "\n";
}
main()
{
cout << " Number of objects created before=";
counter::disp(); //Calling function disp() belonging to class counter
counter cnt1, cnt2, cnt3, cnt4, cnt5;
cout << " Number of objects created recently=";
counter::disp();
getche();
}
```

You should get the following output on running the program.

Number of objects created before=0

Number of objects created recently=5

In defining static functions you should take some precautions. A static member functions cannot be a virtual function.



*Caution* A static or non-static member function cannot have the same name and same arguments type.

### Self Assessment

Fill in the blanks:

1. The …….. should be created and initialized before the main function control block begins.
2. Class members can be declared using the ……. specifier static in the class member list.
3. Only one copy of the static member is shared by all …….. of a class in a program.
4. When you declare an object of a class having a static member, the static member is not part of the……... .
5. A static member function does not have a ……. pointer.

## 5.2 The Const Keyword

The keyword, const (for constant), precedes the data type of a variable.

***Syntax***

```
const <data type> <name of variable> = <value>;
```

Notes



*Example:*

```
const float g = 9.8;
cout << "The acceleration due to gravity is" << g << "m/s2";
g=g+4; // ERROR!! Const variables cannot be modified.
```

The const keyword specifies that the values of the variable will not change throughout the program. This prevents the programmer from changing the value of variable, like the one in the example given above. If the keyword, const, has been used while defining the variable, the compiler will report an error if the programmer tries to modify it.

### 5.2.1 The Class Keyword

Syntax for the class keyword is as follows.

```
class class_name
{
// access control keywords here
// class variables and methods declared here
};
```

You use the class keyword to declare new types. A class is a collection of class member data, which are variables of various types, including other classes. The class also contains class functions—or methods—which are functions used to manipulate the data in the class and to perform other services for the class. You define objects of the new type in much the same way in which you define any variable. State the type (class) and then the variable name (the object). You access the class members and functions by using the dot (.) operator. You use access control keywords to declare sections of the class as public or private. The default for access control is private. Each keyword changes the access control from that point on to the end of the class or until the next access control keyword. Class declarations end with a closing brace and a semicolon.



*Example:*

```
class Cat
{
public:
unsigned int Age;
unsigned int Weight;
void Meow();
};

Cat   Frisky;
Frisky.Age = 8;
Frisky.Weight = 18;
Frisky.Meow();
```

Notes

*Example:*

```
class  Car
{
public:                                  // the next five are public

void  Start();
void  Accelerate();
void  Brake();
void  SetYear(int  year);
int  GetYear();

private:                                 // the rest is private

int  Year;
Char  Model  [255];
};                                        // end of class declaration

Car OldFaithful;                         // make an instance of car
int bought;                               // a local variable of type int
OldFaithful.SetYear(84) ;                // assign 84 to the year
bought  =  OldFaithful.GetYear();       // set bought to 84
OldFaithful.Start();                      // call the start method
```

## 5.3 Static Objects

1. A static object is a class all of whose members (functions and data) are declared static. E.g., (source file)

```
public class Static Example {
static double x;                          // Static data
static void printx ()                     // Static method
{
System.out.println ("x = " + x);
}
public static void main (String[] argv)
{
x = 5.34;
```

Notes

```
printx ();
}
}
```

2. A static object is really just an encapsulation of data and related functions.
3. A static object does not have to have a main function, but to be useful it typically has at least one public function that can be called from somewhere else.



*Task* By default, an object or variable that is defined outside all blocks has static duration and external linkage. Justify.

### Self Assessment

Fill in the blanks:

6. Static duration means that the object or variable is allocated when the program starts and is …….. when the program ends.
7. You cannot declare the members of a …… as static.
8. The threads invoking methods on static objects must wait until the object is fully …….. .
9. A class is a collection of …….., which are variables of various types, including other classes.
10. A member function definition begins with the name of the class, followed by two colons, the name of the……, and its parameters.

## 5.4 Friend Functions

Object oriented programming paradigm secures data because of the data hiding and data encapsulation features. Data members declared as private in a class are restricted from access by non-member functions. The private data values can be neither read nor written by non-member functions. Any attempt made directly to access these members, will result in an error message as "inaccessible data-type" during compilation.

The best way to access a private data member by a non-member function is to change a private data member to a public group. But this goes against the concept of data hiding and data encapsulation. A special mechanism available known as friend function allows non-member functions to access private data. A friend function may be either declared or defined within the scope of a class definition. The keyword friend informs the compiler that it is not a member function nor the property of the class.

The general syntax of the friend function is:

```
friend <return_type> <function_name>(argument list);
```

friend is a keyword. A friend declaration is valid only within or outside the class definition. The following code snippet shows how a friend function is defined.

```
class sample
{
```

Notes

```
private:
              int x;
       public:
              void getdata( );
              friend void disp(sample abc);            //friend function
};
void disp(sample abc)             // non-member function without scope::
operator
{
cout << "value of x ="<< abc.x;
cout << endl;
}
```



*Notes* Note that the function is declared as friend in the class and is defined outside the class. The keyword friend should not be in both the function declaration and definition. The friend declaration is unaffected by its location in the class. It can be declared either in a public or a private section, which does not affect its access right.



*Example:* The following declarations of a friend function are valid:

1. The friend function disp( ) is declared in the public group

```
              class sample
     {
              private :
                     int x;
              public:
                     void getdata( );
                     friend void disp( );
              };
```

2. The friend function disp( ) is declared in the private group

```
              class sample
     {
                private:
                     int x;
                     friend void disp();
                public:
                     void getdata( );
              };
```

**Notes**

Since private data members are available only to the particular class and not to any other part of the program, a non-member function cannot access these private data. Therefore, the friend function is a special type of function which is used to access the private data of any class. In other words, they are defined as non-member functions with the ability to modify data directly or to call function members that are not part of the public interface. The friend class has the right to access as many members of its class. As a result the level of privacy of the data encapsulation gets reduced. Only if it is necessary to access the private data by non-member functions, then a class may have a friend function, otherwise it is not necessary.



*Did u know?* **Is friend function a member of a class?**

A friend function is a function that is not a member of a class but has access to the class's private and protected members.

Let us look at a sample program given below to access the private data of a class by non-member functions through friend function. The program declares a private class example representing variable x and function to input the value for the same. The friend function to display the value of x takes an object as argument with the help of which private variable x can be accessed.

```
#include <iostream.h>
#include <conio.h>
class example
{
private:
int x; .
public:
void getdata()
{
cout << "Enter the value ofx"<< "\n";
cin >> x;
}
friend void disp(example);
};
void disp(example eg)
{
              cout << "Display the entered number"<< eg.x<< "\n";
}
main()
{
              example eg1;
              eg1.getdata();
              disp(eg1);
              getche();
}
```

**Notes**

You should see the following output.

Enter the value of x

4

Display the entered number 4

There are also other areas of application for friend function. The friend function can also be defined within the scope of a class definition itself. Friend function may also have inline member functions. If the friend function is defined in the class, then the inline code substitution is done automatically. But if defined outside the class, then it is necessary to precede the return type with the keyword inline to make the inline code substitution.

The following program accesses the private data of a class through a friend function where the friend function is defined with inline code substitution.

```
#include <conio.h>
class example
{
private:
int x;
public:
inline void getdata();
friend void disp(example);
};
inline void example::getdata()
{
cout <<"Enter the value of x " << "\n";
cin >> x;
}
inline void disp(example eg)       //Note the use of the keyword inline
{
              cout << "Display the entered number" << eg.x << "\n";
}
main()
{
              example eg1 ;
              eg1.getdata();
              disp(eg1);
getche();
}
```

**Notes**

You should see the output as shown below:

Enter the value of x

20

Display the entered number 20

One class can be friendly with another class. Consider two classes, first and second. If the class first is friendly with the other class second, then the private data members of the class first are permitted to be accessed by the public members of the class second. But on the other hand, the public member functions of the class first cannot access the private members of the class second.



*Task* The friend declaration can be placed anywhere in the class declaration. Explain.

The following program demonstrates how a class one has granted its friendship to the class two. The class two is declared friendly in the class one.

```
#include <iostream.h>
#include <conio.h> .
class one
{
              friend class two;
       private:
int x;
       public:
void getdata();
};
class two
{
       public:
void disp(one); .
};
       inline void one::getdata()
{
              cout <<"Enter a number" ;
              cin>>x;
}
       inline void two::disp(one obj)
       {
int x;
```

**Notes**

```
            cout <<"Entered number is ";
            cout << obj.x;
}
void main()
{
            one obj1;
            two obj2;
            obj1.getdata();
            obj2.disp(objl);
getche();
}
```

You should see the output as shown below:

Enter a number 25

Entered number is 25

Though the class first has granted its friendship to the class second, it cannot access the private data of the class second through its public member function display( ) of the class first.

A non-member function can be friendly with one or more classes. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration as it needs to access the private members of both classes.

The general syntax of declaring the same friend function with more than one class is:

```
class second;
class first
{
private:
       //data members;
       //member functions;
public: .
       //data members;
       //member functions;
          friend <return_type> <fname>(class first, class second. . ..);
};
class second
{
private:
       //data members;
       //member functions;
public:
       //data members;
```

**Notes**

```
        //member functions;
        friend <return_type> <fname>(class first, class second. . ..);
};
```

The following program demonstrates the use of the friend function which is friendly to two classes. The function seem to calculate the sum of two objects is declared friendly in both the classes.

```
#include <iostream.h>
#include <conio.h>
class two;
class one
{
intx;
public:
void getdata()
        {
cout <<"Enter the value for x ";
cin >>x;
}
void disp()
{
               cout<< "The value of x entered is "<< x <<"\n".
}
friend int sum( one,two);
};
class two
{
inty;
public:
void getdata()
        {
cout <<"Enter the value for y ";
cin>>y;
}
void disp()
        {
cout<< "The value of y entered is"<< y<< "\n";}
friend int sum(one, two);
};
```

Notes

```
int sum(one obj1,two obj2)
{
return(objl.x + obj2.y);
}
void main()
{
one obj11;
two obj22;
obj11.getdata();
obj11.disp();
obj22.getdata();
obj22.disp();
cout<< "the sum of two private data variables x and y is";
int tot = sum(obj 11,obj22);
cout <<tot;
getche(); .
}
```

You should see the output as shown below.

Enter the value for x 20

The value of x entered is 20

Enter the value for y 11

the value of y entered is 11

the sum of two private data variables x & y is 31



*Notes* Note the forward declaration of class second. The non-member function sum() is declared friendly to class first and class second.

### 5.4.1 Friend Function in Empty Classes

```
#include <iostream>
using namespace std;
class MyClass {
  int a, b;
public:
  MyClass(int i, int j) { a=i; b=j; }
   friend int friendFunction(MyClass x); // a friend function
};
```

Notes

```
// friendFunction() is a not a member function of any class.
int friendFunction(MyClass x)
{
   /* Because friendFunction() is a friend of MyClass, it can
      directly access a and b. */
   int max = x.a < x.b ? x.a : x.b;
   return max;
}
int main()
{
   MyClass n(18, 111);
     cout << "friendFunction(n) is " << friendFunction(n) << "\n";
   return 0;
}
friendFunction(n) is 18
```

### 5.4.2 Friend Function in Nested Classes

Friend functions declared in a nested class are considered to be in the scope of the nested class, not the enclosing class. Therefore, the friend functions gain no special access privileges to members or member functions of the enclosing class. If you want to use a name that is declared in a nested class in a friend function and the friend function is defined in file scope, use qualified type names as follows:

```
// friend_functions_and_nested_classes.cpp
#include <string.h>
char *rgszMessage[255];
class BufferedIO
{
public:
    class BufferedInput
   {
    public:
         friend int GetExtendedErrorStatus();
        static char *message;
       int iMsgNo;
   };
};
char *BufferedIO::BufferedInput::message;
int GetExtendedErrorStatus()
```

Notes

```
{
    int iMsgNo = 1; // assign arbitrary value as message number
    strcpy( BufferedIO::BufferedInput::message,
            rgszMessage[iMsgNo] );
    return iMsgNo;
}
int main()
{
}
```

The preceding code shows the function GetExtendedErrorStatus declared as a friend function. In the function, which is defined in file scope, a message is copied from a static array into a class member. Note that a better implementation ofGetExtendedErrorStatus is to declare it as:

int GetExtendedErrorStatus( char *message )

With the preceding interface, several classes can use the services of this function by passing a memory location where they want the error message copied.



*Task* The friend function has access to the private data member of the Point object it receives as a parameter. Analyse.

### 5.4.3 Friend Function in Local Classes

The name of a friend function or class first introduced in a friend declaration is not in the scope of the class granting friendship (also called the enclosing class) and is not a member of the class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

If the name of a friend class has been introduced before the friend declaration, the compiler searches for a class name that matches the name of the friend class beginning at the scope of the friend declaration. If the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

The scope of a friend class name is the first nonclass enclosing scope. For example:

```
class A {
    class B { // arbitrary nested class definitions
       friend class C;
   };
};
is equivalent to:
```

**Notes**

```
class C;
class A {
    class B { // arbitrary nested class definitions
       friend class C;
   };
};
```

If the friend function is a member of another class, you need to use the scope resolution operator (::).



*Example:*

```
class A {
public:
  int f() { }
};
class B {
   friend int A::f();
};
```

Friends of a base class are not inherited by any classes derived from that base class. The following example demonstrates this:

```
class A {
   friend class B;
   int a;
};
class B { };
class C : public B {
   void f(A* p) {
//     p->a = 2;
  }
};
```

The compiler would not allow the statement p->a = 2 because class C is not a friend of class A, although C inherits from a friend of A.

Friendship is not transitive. The following example demonstrates this:

```
class A {
   friend class B;
   int a;
};
class B {
   friend class C;
```

Notes

```
};
class C {
   void f(A* p) {
//     p->a = 2;
   }
};
```

The compiler would not allow the statement p->a = 2 because class C is not a friend of class A, although C is a friend of A.

If you declare a friend in a local class, and the friend's name is unqualified, the compiler will look for the name only within the innermost enclosing nonclass scope. You must declare a function before declaring it as a friend of a local scope. You do not have to do so with classes.



*Notes* Note that a declaration of a friend class will hide a class in an enclosing scope with the same name.

The following example demonstrates this:

```
class X { };
void a();
void f() {
   class Y { };
   void b();
   class A {
      friend class X;
      friend class Y;
      friend class Z;
//      friend void a();
      friend void b();
//      friend void c();
   };
   ::X moocow;
//   X moocow2;
}
```

In the above example, the compiler will allow the following statements:

1. friend class X: This statement does not declare ::X as a friend of A, but the local class X as a friend, even though this class is not otherwise declared.
2. friend class Y: Local class Y has been declared in the scope of f().
3. friend class Z: This statement declares the local class Z as a friend of A even though Z is not otherwise declared.

Notes

4. friend void b(): Function b() has been declared in the scope of f().
5. ::X moocow: This declaration creates an object of the nonlocal class ::X.

The compiler would not allow the following statements:

1. friend void a(): This statement does not consider function a() declared in namespace scope. Since function a() has not been declared in the scope of f(), the compiler would not allow this statement.
2. friend void c(): Since function c() has not been declared in the scope off(), the compiler would not allow this statement.
3. X moocow2: This declaration tries to create an object of the local class X, not the nonlocal class ::X. Since local class X has not been defined, the compiler would not allow this statement.



*Caselet*

## History of a Hi-tech City (Bengaluru)

"To establish, maintain, conduct, provide, procure or make available services of every kind including commercial, statistical, financial, accountancy, medical, legal, management, educational, engineering, data processing, communication and other technological, social or other services."

"To carry on the business as importer, exporter, buyers, lessors, and sellers of and dealers in all types of electronic components and equipments necessary for attaining the above objects."

Thus reads a portion of a section titled 'the main objects of the company, as set out in the memorandum of association,' in The Birth of the Infosys Saga, a chapter included by Narendar Pani, Sindhu Radhakrishna and Kishor G. Bhat in Bengaluru, Bangalore, Bengaluru (www.sagepublications.com).

Another excerpt from 'Infosys IPO Prospectus 1993' is about DMAP (distributor management application package), which addresses 'the MIS needs of a distributor of consumer products,' by handling 'sales order processing, customer service, accounts receivable, allocation, invoicing, warehouse management, purchasing, inventory, sales analysis and forecasting.' The package, with over 600 programs and running on IBM AS/400, is used by Reebok, informs the description.

**Three Segments**

Under 'marketing' one reads about the three major segments from which Indian software companies derive business _ on-site services, turnkey projects, and products/ packages. "On-site is a low technology business involving selection of software personnel with appropriate educational qualifications and experience in developing software on the desired hardware platforms and sending them abroad to work on-site at client locations. A large number of software companies in India are in this segment of business."

Turnkey project management, the second segment, involves understanding of the clients' requirements, estimation of effort, time, resource input (both hardware installation and software personnel), data communication, and developing high quality software on

*Contd...*

Notes

schedule. "ITL is one of the few companies in India who have the capability to manage turnkey projects. The combined experience of the promoters in this area is over 100 man years," the prospects notes.

**Kempe Gowda's Search**

The book of 'imaginations and their times' begins by tracing old Bengaluru, seven miles south of Yelahanka, near what is now the urban village of Kodigehalli. And the context is the search by Kempe Gowda for a place to locate his fortress town, five hundred years ago, with an emphasis on early warning.

"The pace at which armies moved in the sixteenth century meant that being able to spot the enemy many miles away was a great advantage… And the high ground to the south of Yelahanka, with its hillocks and lakes must have seemed ideally suited for this purpose."

**Polarised Urban Society**

Fast-forward to the current times, in 'Governing change', an essay by Christoph Dittrich that acknowledges the shift from 'garden city' to 'pensioners' paradise' to 'hi-tech capital' in the 1990s based on Bangalore's meteoric rise to a global powerhouse for software development and other computer-based service industries.

The author finds that Bangalore's tremendous population growth, the extensive urban sprawl, and the authorities' policy and development priorities towards the ICT sector have induced profound changes in settlement patterns and in the urban social setting, creating new disparities and a highly fragmented and polarised urban society.

"Critics also point out that the top-down perspective followed by the government core agencies are reinforcing rather than tackling socio-economic polarisation and inequality. The challenge is to create favourable conditions for reversing the polarities and for more social cohesion…"

A work of importance, presenting a model that can be emulated in the study of other Indian cities.

**Project Mantras**

Information seeding is the third of 'the seven mantras' in Steering Project Success: Simple innovations in execution by Madhavan S. Rao (www.tatamcgrawhill.com). Consistently seed information in the customer's mind, proactively right from the beginning of a project, the author advises.

"This can be done through weekly or periodic status reports, conference calls, other customer interaction opportunities and internal team meetings. Seeding information will ensure that the customer and the team share the same understanding on the project execution details and milestones to be achieved."

A scenario that Rao paints in this context is of an offshore team receiving a complete outsourcing project of a relatively large application. And the customer's project manager (PM) wants that the challenging aspects of the SLA (Service Level Agreement) be undertaken in the initial stage of the project.

"The offshore team explained to the customer's PM the risks involved in doing the same. The team also convinced the PM that the SLA implementation should start only after the initial knowledge transfer phase. The team provided information and convinced the PM about the negative impact if the SLA was implemented in the initial stage of the work," reads the 'approach' that Rao recommends.

*Contd...*

Notes

**Bug Tracking**

Another situation is of a product support project, in which the team found the number of reported bugs to be high owing to the complexity of the system, new releases and features. "It was difficult to track the status and progress of each bug since the backlog of defects was high…"

Here comes the relevance of a bug tracking system, a database containing the entire history of the bug, as the author instructs. The system – with info about priority, author, owner, target release, abstract and so on – can be updated through mails based on the bug ID, and communicated automatically to the identified recipients, he adds.

"The mails keep the team updated on the new bugs and the respective solutions. Include this as a process improvement so that the projects would have an automated bug tracking system for education and information seeding,".

**Comfort Levels**

Mantra six calls for steering the comfort levels, such as enhancing the delight level of end customers. Sample this context, described by Rao: The customer, a bank, wanted to implement new software but wanted to retain the existing front-ends as the end users were comfortable with the same.

The software vendor began by seeking feedback from the users in the bank; finding that the user community was conversant and comfortable with the old screens, the vendor collected data from the front-ends of the old application and transferred the same to the database of the new application.

"Once the data transfer was completed, it was processed to retain the same front-ends on the new application. All this ran transparently." And the outcome, Rao concludes, was that end-users' comfort level could be kept very high, as they did not see any changes on the front-end screens of the new application.

**Instructive Read**

***Five Generations of Programming Language***

The Visual Studio development environment is 'an extremely powerful fifth-generation tool,' says Rod Stephens in Visual Basic 2010: Programmer's reference (www.wileyindia.com).

"It provides graphical editors to make building forms and editing properties easy and intuitive; IntelliSense to help developers remember what to type next; auto-completion so developers can use meaningful variable names without needing to waste time typing them completely by hand; tools that show call hierarchies indicating which routines call which others; and breakpoints, watches, and other advanced debugging tools that make building applications easier," he describes.

Fifth-generation languages or 5GLs emphasise on the development environment rather than the language itself; they provide powerful, highly graphical development environments to allow developers to use the underlying language in more sophisticated ways, informs the author in a discussion of the different generations.

**1-2-3-4**

The previous generation, of the 4GLs, was of 'natural languages' such as SQL. They let developers use a language that is sort of similar to a human language to execute programming tasks, explains Stephens. "For example, the SQL statement 'SELECT * FROM

Contd...

Notes

Customers WHERE Balance > 50' tells the database to return information about customers that owe more than $50."

Examples of 3GLs are Pascal and FORTRAN, which provided elements such as subroutines, loops, and data structures. And the 2GL was assembly language, providing terse mnemonics for machine instructions. At the beginning, as you would have guessed by now, was the language of 0s and 1s, the machine language. "You actually had to program some early computers by painstakingly toggling switches to enter 0s and 1s!"

### Self Assessment

Fill in the blanks:

11. A friend function is declared by the class that is granting …….. .
12. The friend declaration can be placed …….. in the class declaration.
13. To declare a friend function, simply use the …….. keyword in front of the prototype of the function you wish to be a friend of the class.
14. A function can be a friend of more than one class at the……... .
15. It is also possible to make an entire class a …… of another class.

## 5.5 Summary

- Member data items of a class can be static. Static data members are data objects that are common to all objects of a class.
- They exist only once in all objects of this class.
- The static members are used when the information is to be shared.
- They can be public or private data. The main advantage of using a static member is to declare the global data which should be updated while the program lives in memory.
- A static member function is not a part of objects of class.
- It is instance dependent and can be accessed directly by using the class name and scope resolution operator.
- If it is declared and defined in a class, the keyword static should be used only in declaration part.
- The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend.
- The friend function must have the class to which it is declared as friend passed to it in argument.

## 5.6 Keywords

***Friend Function:*** A function which is not a member of a class but which is given special permission to access private and protected members of the class.

***Static Data Members:*** Static data members are data objects that are common to all objects of a class.

***Static Member Functions:*** Functions that can access only the static members.

Notes

## 5.7 Review Questions

1. Why did C++ add the class keyword?
2. The const keyword specifies that the values of the variable will not change throughout the program. Analyze.
3. "The static keyword can be used to declare variables, functions, class data members and class functions". Justify with an example.
4. Does friend function violate encapsulation? Explain with an example.
5. What are the some advantagesd or isadvantages of using friend functions?
6. What does it mean that "friendship isn't inherited, transitive, or reciprocal"?
7. "Should my class declare a member function or a friend function?" Do you agree with this statement? Why or why not. Justify you answers with an example.
8. "A friend function is a function that can access the private members of a class as though it were a member of that class". Explain.
9. Write a program which is using the friend function in the nested classes and local classes.
10. Discuss why a friend function is used for accessing the non-public members of a class.

### Answers: Self Assessment

1. static data member
2. storage class
3. objects
4. class object
5. this
6. deallocated
7. union
8. constructed
9. class member data
10. function
11. access
12. anywhere
13. friend
14. same time
15. friend

## 5.8 Further Readings



*Online links*

http://en.wikipedia.org/wiki/Friend_function

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/topic/com.ibm.xlcpp8a.doc/language/ref/cplr038.htm

# Unit 6: Constructors and Destructors

Notes

CONTENTS

Objectives

Introduction

6.1 Need for Constructor and Destructor

6.1.1 Constructor

6.1.2 Destructor

6.2 Copy Constructor

6.3 Dynamic Constructors

6.4 Parameterized Constructors

6.5 Constructors with Default Arguments

6.6 Destructors

6.7 Constructor/Destructor with Static Members

6.8 Summary

6.9 Keywords

6.10 Review Questions

6.11 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the need for constructor and destructors
- Describe the copy constructor
- Explain the dynamic constructor
- Discuss the destructors
- Explain the constructor and destructors with static members

## Introduction

When an object is created all the members of the object are allocated memory spaces. Each object has its individual copy of member variables. However the data members are not initialized automatically. If left uninitialized these members contain garbage values. Therefore it is important that the data members are initialized to meaningful values at the time of object creation. Conventional methods of initializing data members have lot of limitations. In this unit you will learn alternative and more elegant ways initializing data members to initial values.

When a C++ program runs it invariably creates certain objects in the memory and when the program exits the objects must be destroyed so that the memory could be reclaimed for further use.

C++ provides mechanisms to cater to the above two necessary activities through constructors and destructors methods.

Notes

## 6.1 Need for Constructor and Destructor

### 6.1.1 Constructor

Constructor is public method that is called automatically when the object of a particular class is created. C++ provides a default constructor method to all the classes. This constructor method takes no parameters. Actually the default constructor method has been defined in system.object class. Since every class that you create is an extension of system.object class, this method is inherited by all the classes.

The default constructor method is called automatically at the time of creation of an object and does nothing more than initializing the data variables of the object to valid initial values.



*Notes* A Programmer can also define constructor methods for a class if he/she so desires.

While writing a constructor function the following points must be kept in mind:

1. The name of constructor method must be the same as the class name in which it is defined.
2. A constructor method must be a public method.
3. Constructor method does not return any value.
4. A constructor method may or may not have parameters.

Let us examine a few classes for illustration purpose. The class abc as defined below does not have user defined constructor method.

```
class  abc
{
       int  x,y;
}
main()
{
       abc  myabc;
       …;
}
```

The main function above an object named myabc has been created which belongs to abc class defined above. Since class abc does not have any constructor method, the default constructor method of C++ will be called which will initialize the member variables as:

```
myabc.x=0
and
myabc.y=0.
```

Let us now redefine myabc class and incorporate an explicit constructor method as shown below:

```
class  abc
```

Notes

```
{
        int  x,y;
        public:
                abc(int,  int);
}
abc::abc(int  a,  int  b)
{
        x=a;
        y=b;
}
```



*Task* 'The constructor also usually holds the initializations of the different declared member variables of its object'. Explain this statement.

Observed that myabc class has now a constructor defined to except two parameters of integer type. We can now create an object of myabc class passing two integer values for its construction, as listed below:

```
main()
{
        abc  myabc(100,200);
        ---;
}
```

In the main function myabc object is created value 100 is stored in data variable x and 200 is stored in data variable y. There is another way of creating an object as shown below.

```
main()
{
        myabc=abc(100,200);
        ---;
}
```

Both the syntaxes for creating the class are identical in effect. The choice is left to the programmer.

There are other possibilities as well. Consider the following class differentials:

```
class  abc
{
        int  x,y;
        public:
                abc();
}
```

**Notes**

```
abc::abc()
{
        x=100;
        y=200;
}
```

In this class constructor has been defined to have no parameter. When an object of this class is created the programmer does not have to pass any parameter and yet the data variables x,y are initialized to 100 and 200 respectively.

Finally, look at the class differentials as given below:

```
class  abc
{
        int  x,y;
        public:
                abc();
abc(int);
abc(int,  int);
}
abc::abc()
{
        x=100;
        y=200;
}
abc::abc(int  a)
{
        x=a;
        y=200;
}
abc::abc(int  a)
{
        x=100;
        y=a;
}
```

Class myabc has three constructors having no parameter, one parameter and two parameters respectively. When an object to this class is created depending on number of parameters one of these constructors is selected and is automatically executed.

Notes



*Notes* Note that C++ selects one constructor by matching the signature of the method being called. Also, once you define a constructor method, the default constructor is overridden and is not available to the class. Therefore you must also define a constructor method resembling the default constructor method having no parameters.

### 6.1.2 Destructor

When a program no longer needs an instantiated object it destroys it. If you do not supply a destructor function, C++ supplies a default destructor for you, unknown to you. The program uses the destructor to destroy the object for you.

When should you define your own destructor function? In many cases you do not need a destructor function. However, if your class created dynamic objects, then you need to define your own destructor in which you will delete the dynamic objects. This is because dynamic objects cannot be deleted on their own. So, when the object is destroyed, the dynamic objects are deleted by the destructor function you define.



*Did u know?* **Should we explicitly call a destructor on a local variable?**

No!

The destructor will get called again at the close } of the block in which the local was created. This is a guarantee of the language; it happens automatically; there's no way to stop it from happening. But you can get really bad results from calling a destructor on the same object a second time! Bang! You're dead!

A destructor function has the same name as the class, and does not have a returned value. However you must precede the destructor with the tilde sign, which is ~ .

The following code illustrates the use of a destructor against dynamic objects:

```
#include <iostream>

using namespace std;


class Calculator

    {

         public:

         int *num1;

         int *num2;


          Calculator(int ident1, int ident2)

            {

                 num1 = new int;

                 num2 = new int;
```

Notes

```
                    *num1 = ident1;
                    *num2 = ident2;
                }

            ~Calculator()
                {
                    delete num1;
                    delete num2;
                }

            int add ()
                {
                    int sum = *num1 + *num2;
                    return sum;
                }
    };
int main()
    {
        Calculator myObject(2,3);
        int result = myObject.add();
        cout << result;

        return 0;
    }
```

The destructor function is automatically called, without you knowing, when the program no longer needs the object. If you defined a destructor function as in the above code, it will be executed. If you did not define a destructor function, C++ supplies you one, which the program uses unknown to you. However, this default destructor will not destroy dynamic objects.



*Notes* An object is destroyed as it goes out of scope.

## Self Assessment

Fill in the blanks:

1. A constructor is a special method that is created when the …… is created or defined.
2. A constructor is declared without a return value, that also excludes……..
3. You cannot declare a constructor as virtual or static, nor can you declare a constructor as ……, volatile, or const volatile.

Notes

4. …….. are called at the point an object is created.
5. Specifying a constructor with a return type is an error, as is taking the …….. of a constructor.
6. A destructor takes no …… and has no return type.

## 6.2 Copy Constructor

A copy constructor method allows an object to be initialized with another object of the same class. It implies that the values stored in data members of an existing object can be copied into the data variables of the object being constructed, provided the objects belong to the same class. A copy constructor has a single parameter of reference type that refers to the class itself as shown below:

```
abc::abc(abc  &  a)

{

        x=a.x;

        y=a.y;

}
```

Suppose we create an object myabc1 with two integer parameters as shown below:

```
abc  myabc1(1,2);
```

Having created myabc1, we can create another object of abc type, say myabc2 from myabc1, as shown below:

```
myabc2=abc(&  myabc1);
```

The data values of myabc1 will be copied into the corresponding data variables of object myabc2. Another way of activating copy constructor is through assignment operator. Copy constructors come into play when an object is assigned another object of the same type, as shown below:

```
abc  myabc1(1,2);

abc  myabc2;

myabc2=myabc1;
```

Actually assignment operator(=) has been overloaded in C++ so that copy constructor is invoked whenever an object is assigned another object of the same type.



*Did u know?* **What is the difference between the copy constructor and the assignment operator?**

(a) If a new object has to be created before the copying can occur, the copy constructor is used.

(b) If a new object does not have to be created before the copying can occur, the assignment operator is used.

Notes

## 6.3 Dynamic Constructors

Allocation of Memory during the creation of objects can be done by the con structures too.

The memory is saved as it allocates the right amount of memory for each object (Objects are not of the same size). Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. New operator is used to allocate memory.

The following program concatenates two strings. The constructor function initializes the strings using constructor function which allocates memory during its creation.

```
#include <iostream.h>
#include <string.h>
class string
{
        char * name;
        intlength;
        public:
        string 0
{
        length = 0;
        name = new char [length = 1];
};
string (char*s)
{
length = strlen (s);
name = new char [length + 1];
strcpy(name,s );
};
void display (void)
{
        cout<<"\n Name :- "<<name;
};
void join (string & a, string & b)
{
        length = a.length + b.length;
        delete name;
        name = new char [length + 1];
        strcpy (name,a.name);
        strcat (name," ");
        strcat (name,b.name);
};
};
main()
```

Notes

```
{
        char * FirstName= "Mohan";
        string Fname(First name);
        string Mname("Kumar");
        string Sname("Singh");
        string Halfname, Fullname;
//Joining FirstName with Surname
Halfname.join (Fname, Sname);
//Joining Firstname with Middlename & Surname
Fullname.join (Halfname, Mname);
Fname.display ();
Mname.display ();
Sname.display ();
Halfname.display();
Fullname.displayO();
}
You should see the following output.
/*
        Name :- Ram
        Name :- Kumar
        Name :- Singh
        Name:- Mohan .Singh Name :- Mohan.Singh .Kumar
*/
```

The above program uses new operator to allocate memory. The first constructor is an empty constructor that allows us to declare an array of string. The second constructor initializes length of the string, allocates necessary space for the string to be stored and creates the string itself. The member function join ( ) concatenates 2 strings.

It actually adds the length of 2 strings and then allocates the memory for the combined string. After that the join function uses inbuilt string functions strcpy & strcat to fulfil the action.

The output of the program will be in Full Name and Half name.

That is Mohan Singh Kumar and Ram Singh respectively.

Another example of the dynamic constructor is the matrix program. In two Dimensional matrix we need to allocate the memory for the values to be stored in. Using constructor we can allocate the memory for the matrix.



*Task* 'Basically, it's a way of constructing an object based on the run-time type of some existing object.' Explain.

**Notes**

This program declares a class matrix to represent the number of rows and columns of matrix as well as two dimensional array to store the contents of matrix. The constructor function used to initialize the objects allocates the required amount of memory for the matrix.

```
#include<iostream.h>
#include<conio.h>
//Class  Definition
class  matrix
{
       int  **p;      //declaring  two  dimensional  array
       int  dl,d2;
public:
       matrix(intx,inty);
       void  get_value(  void);
       void  dis_value(  void);
       void  square(void);
       void  cube(void);
};
matrix:  :  matrix(  int  x,int  y)
{
       dl=x;
       d2=y;
       p=  new  int  *[d1];
       for  (int  i  =0;i<d1  ;i++)
       p[i]=new  int[d2];
} ;
void  matrix::  get_value(void)
{
       for(int  i  =  0;i<d1;i++)
       for(int  j=0;j<d2;j++)
{
       cout<<"Please  Enter  A  No  At"<<i<<j<<  "Position  :-";
       cin>>p[i][j];
}
} ;
voidmatrix::  dis_value(void)
{
       cout<  <"The  Matrix  Entered";
       for  (int  i  =  0;i<d1  ;i++)
```

**Notes**

```
{
        cout< <"\n";
        for(int  j=0;j<d2;j++)
        cout<<"\t"<<p[i]  [j];
}
};
void  matrix::  square(void)
{
        cout  <  <"\n  The  Squared  Matrix";
        for  (int  i  =  0;i<d1  ;i++  )
{
        cout<<"\n";.  for(intj=0;j<d2;j++)
cout<<"\t"  <<p[i][j]*p[i]  [j];
}
};
void  matrix::  cube(void)  {
        cout  <  <"\nThe  Cubed  Matrix";
        for  (int  i  =  0;i<d1;i++)
{
        cout<<"\n";
        for(int  j=0;j<d2;j++  )
        cout<<"\t"<<p[i][j]  *  p[im]  *  p[i][j];
}
};
//Start  Of  Main  Program
main()
{
        intm,n;
        elrser();
        cout<<"Enter  Size  Of  Matrix  :-  ";
        cin>  >rn>>n;
        matrix  mat(m,n);
        mat.get_value();
        mat.dis_value();
        getch();
        mat.square();
        mat.cube();
```

**Notes**

```
        getch();
}
```

Let us assume we run this program with the following input.

/*

Enter Size of matrix :- 2 2

Please Enter A No At Position 00 :- 1

Please Enter A No At Position 01 : - 1

Please Enter A No At Position 10 : - 2

Please Enter A No At Position 11 :- 2

*1

You should see the following output.

/*

The Matrix Entered

1 1

2 2

The Squared Matrix

1 1

4 4

The Cubed Matrix

1 1

8 8

*/

The Constructor first creates a vector pointer to an int of size dl. Then it allocates, iteratively, an int type vector of size d2 pointed at each element p[i]. Thus space for the element of a d1x d2 matrix is allocated from free store.

## Self Assessment

Fill in the blanks:

7. A copy constructor method allows an object to be initialized with another object of the …….. class.
8. A copy constructor has a …… parameter of reference type that refers to the class itself.
9. The memory is saved as it ……. the right amount of memory for each object.
10. Allocation of …… to objects at the time of their construction is known as dynamic construction of objects.

## 6.4 Parameterized Constructors

Notes

If it is necessary to initialize the various data elements of different objects with different values when 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 'Parameterized constructors.' The definition and declaration are as follows:

```
class dist
{
        int m, cm;
public:
        dist(int x, int y);
};
dist::dist(int x, int y)
{
        m = x;  n = y ;
}
main()
{
dist d(4,2);
d. show ();
}
```



*Task* C++ permits us to achieve this objects but passing argument to the constructor function when the object are created. Discuss with a proper example.

## 6.5 Constructors with Default Arguments

This method is used to initialize object with user defined parameters at the time of creation.

Consider the following Program that calculates simple interest. It declares a class interest representing principal, rate and year. The constructor function initializes the objects with principal and number of years. If rate of interest is not passed as an argument to it the Simple Interest is calculated taking the default value of rate of interest

```
#include <iostream.h>
#include <conio.h>
class interest
{ int principal, rate, year;
        float amount;
        public
```

**Notes**

```
        interest (int p, int n, int r = 10);
        void cal (void);
};
        interest::interest (int p, int n, int r = 10)
{ principal = p; year = n; rate = r;
};
void interest::cal (void)
{
        cout<< "Principal" <<principal;
        cout << "\ Rate" <<rate;
        cout<< "\ Year" <<year;
        amount = (float) (p*n*r)/100;
        cout<< "\Amount" <<amount;
};
main ( )
{
        interest i1(1000,2);
        interest i2(1000, 2,15);
        clrscr( );
        il.cal();
        i2.cal();
}
```

*Notes* The two objects created and initialized in the main( ) function.

```
interest i1(1000,2);
interest i2(1000,2, 15);
```

The data members principal and year of object i1 are initialized to 1000 and 2 respectively at the time when object i1 is created. The data member rate takes the default value 10 whereas when the object 12 is created, principal, year and rate are initialized to 1000,2 and 15 respectively.

It is necessary to distinguish between the default

```
constructor::construct();
```

and default argument constructor

```
construct::construct(int = 0)
```

The default argument constructor can be called with one or no arguments. When it is invoked with no arguments it becomes a default constructor. But when both these forms are used in a class, it causes ambiguity for a declaration like construct C1;

The ambiguity is whether to invoke construct: : construct ( ) or construct: : construct (int=O)

## 6.6 Destructors

Notes

Constructors create an object, allocate memory space to the data members and initialize the data members to appropriate values; at the time of object creation. Another member method called destructor does just the opposite when the program creating an object exits, thereby freeing the memory.

A destructive method has the following characteristics:

1. Name of the destructor method is the same as the name of the class preceded by a tilde(~).
2. The destructor method does not take any argument.
3. It does not return any value.

The following codes snippet shows the class abc with the destructor method;

```
class abc
{
  int x,y;
  public:
         abc();
abc(int);
abc(int, int);
~abc()
{
cout << "Object being destroyed!!";
         }
}
```

Whenever an object goes out of the scope of the method that created it, its destructor method is invoked automatically. However if the object was created using new operator, the destructor must be called explicitly using delete operator. The syntax of delete operator is as follows:

```
delete(object);
```



*Notes* Whenever you create an object using new keyword, you must explicitly destroy it using delete keyword, failing which the object would remain homed in the memory and in the course of program execution there may come a time when sufficient memory is not available for creation of the more objects. This phenomenon is referred to as memory leak. Programmers must consider memory leak seriously while writing programs for the obvious reasons.



*Did u know?* **What's the order that objects in an array are destructed?**

In reverse order of construction: First constructed, last destructed.

In the following example, the order for destructors will be a[9], a[8], ..., a[1], a[0]:

```
void userCode()
{
   Fred a[10];
   ...
}
```

Notes

## 6.7 Constructor/ Destructor with Static Members

A CLR type, such as a class or struct, can have a static constructor, which can be used to initialize static data members. A static constructor will be called at most once, and will be called before the first time a static member of the type is accessed.

An instance constructor will always run after a static constructor.

The compiler cannot inline a call to a constructor if the class has a static constructor. The compiler cannot inline a call to any member function if the class is a value type, has a static constructor, and does not have an instance constructor. The common language runtime may inline the call, but the compiler cannot.

A static constructor should be defined as a private member function, as the static constructor is only meant to be called by the common language runtime.

To get the equivalent of a static constructor, you need to write a separate ordinary class to hold the static data and then make a static instance of that ordinary class.

```
class  StaticStuff
{
       std::vector<char>  letters_;
public:
      StaticStuff()
     {
          for  (char  c  =  'a';  c  <=  'z';  c++)
               letters_.push_back(c);
     }
      // provide  some  way  to  get  at  letters_
};
class  Elsewhere
{
    static StaticStuff staticStuff; // constructor runs once, single instance
};
```



*Example:*

```
class A {
int a;
public:
A(int i) {a=i;}
~A() {cout << "bye\n";}
};
A x(3); // Static constructor
main() {
// do stuff..
```

```
}
// static destructor of x called now.
```

Notes



*Caselet*

## Clearly Making a Point

XEROX-PARC (Palo Alto Research Centre, where the world's first graphic user interface and the SmallTalk-80 object-oriented programming was developed) has an unusual project on in India. This technology company has been researching methods of making technology more human-friendly. The brief was simple: "Why should humans always adapt to tech? Why not make tech easy and natural for human beings to use?"

**Feeling at Home**

The group came up with a variety of interface devices that were a lot more instinctive than the ordinary keyboard-mouse devices.

Devices with names such as e-Shiva egg, the e-rickshaw and the 360 degree Tilty screen, have been developed which can be used by just anyone. "No one needs to know anything about technology. You can get information instinctively," says Mustafa Siddiqui, a team member.

The research group's demo chose to base its projects on subjects that were very Indian, such as Benaras ("which is a microcosm of India") and Shiva. "We wanted to produce technology in India in an Indian way," explains Siddiqui.

The "Interactive Physical Icons", are real physical objects such as a wooden trishul, ring and so on which can be placed on the cursor. The system understands this as a command for more information on the trishul.

The e-rickshaw (a real rickshaw with a screen fitted on the seat) demonstrates best the kind of instinctive interaction built into the products. Someone who has never seen a keyboard and a mouse would be bewildered by a PC as we know it, but anyone looking at the e-rickshaw would instinctively turn the handle bars, the key to steering through a video on Benaras in the demo project. Ringing the rickshaw bell would bring more pictures and information on the screen.

The e-Shiva egg, an egg-shaped device that can be cradled in a palm with buttons on the side, works like any palmtop. The demo device has information on Shiva and so was shaped to feel like Shiva.

The 360-degree swivel display can be turned around a full circle and the screen shows various Benaras skylines as if through a handycam.

The Crossing Project, as it was called, was the brainchild of Ranjit Makkuni, a multimedia researcher at Xerox-PARC, who is also a designer and musician. Makkuni continues to explore the "non-button pushing, gesture-based interfaces" and tries to bridge the traditional and the contemporary.

It was begun purely as a research project and the team has exhibited 21 products in Mumbai at the National Gallery of Modern Art, says Siddiqui.

Three exhibitions later, the team has some commercial projects on hand. Some of the requests have come in from the tourism sector, a large media group, a GSM operator and even a large public sector unit.

Notes

### Self Assessment

Fill in the blanks:

11. C++ permits us to achieve this objective by passing arguments to the constructor function when the …….. are created.
12. A static constructor should be defined as a …….. member function.
13. The …….. is called every time an object goes out of scope or when explicitly deleted by the programmer (using operator delete).
14. The constructors that can take arguments are called ……. constructors.
15. A destructor is called for a class object when that object passes out of scope or is …….. deleted.

## 6.8 Summary

- A constructor is a member function of a class, having the same name as its class and which is called automatically each time an object of that class it created.
- It is used for initializing the member variables with desired initial values. A variable (including structure and array type) in C++ may be initialized with a value at the time of its declaration.
- The responsibility of initialization may be shifted, however, to the compiler by including a member function called constructor.
- A class constructor, if defined, is called whenever a program created an objects of that class. Constructors are public member unless otherwise there is a good reason against.
- A constructor may take argument (s). A constructor may taking no argument(s) is known as default constructor.
- A constructor may also have parameter (s) or argument (s), which can be provided at the time of creating an object of that class.
- C++ classes are derived data types and so they have constructor (s). Copy constructor is called whenever an instance of same type is assigned to another instance of the same class.
- If a constructor is called with less number of arguments than required an error occurs. Every time an object is created its constructor is invoked.
- The functions that is automatically called when an object is no more required is known as destructor. It is also a member function very much like constructors but with an opposite intent.

## 6.9 Keywords

***Constructor:*** A member function having the same name as its class and that initializes class objects with legal initial values.

***Copy Constructor:*** A constructor that initializes an object with the data values of another object.

***Default Constructor:*** A constructor that takes no arguments.

***Destructor:*** A member function having the same name as its class but preceded by ~ sign and that deinitializes an object before it goes out of scope.

Notes

***Friend Function:*** A function which is not a member of a class but which is given special permission to access private and protected members of the class.

***Static Member Functions:*** Functions that can access only the static members.

***Temporary Object:*** An anonymous short lived object.

## 6.10 Review Questions

1. Write a program to calculate prime number using constructor.
2. Is there any difference between List x; and List x();? Explain.
3. Can one constructor of a class call another constructor of the same class to initialize the this object? Justify your answers with an example.
4. Should my constructors use "initialization lists" or "assignment"? Discuss.
5. What is the "Named Constructor Idiom"?
6. Does return-by-value mean extra copies and extra overhead?
7. What about returning a local variable by value? Does the local exist as a separate object, or does it get optimized away?
8. Why cannot we pass an object by value to a copy constructor?
9. Why are classes with static data members getting linker errors?
10. Spot out the error in the following code and correct it.

```
class  one
{
                int  x,y;
         public:
                one(int);
one(int);
one(int,  int);
~one(int)
{
cout  <<  "Object  being  destroyed!!";
                }
}
```

11. How is a copy constructor different from a constructor? Illustrate with suitable examples.
12. Debug the following code:

```
class  interest
{
int  principal,  rate,  year;
         float  amount;
         public:
```

Notes

```
                    interest(int p=1000, int n, int r = 10);
            };
            interest::interest(int p=1000, int n, int r = 10)
{
principal = p; year = n; rate = r;
};
```

13. What is the purpose of having default arguments in a constructor? Explain with suitable examples.

### Answers: Self Assessment

1. object
2. void
3. const
4. Constructors
5. address
6. arguments
7. same
8. single
9. allocates
10. memory
11. objects
12. private
13. destructor
14. Parameterized
15. explicitly

## 6.11 Further Readings



*Books*

E Balagurusamy; *Object Oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://www.cprogramming.com/tutorial/constructor_destructor_ordering.html

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101 index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr374.htm

# Unit 7: Operator Overloading

Notes

CONTENTS

Objectives

Introduction

7.1 Defining Operator Overloading

7.2 Rules for Overloading Operators

7.3 Overloading Unary Operators

7.4 Overloading Binary Operators

7.4.1 Using Friend Function

7.4.2 Using Member Function

7.5 Manipulation of Strings using Operator Overloading

7.6 Summary

7.7 Keywords

7.8 Review Questions

7.9 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the operator overloading
- Describe the rules for operator overloading
- Explain the overloading of unary operators
- Discuss the various binary operators with friend function and member function

## Introduction

C++ provides a rich collection of operators. You have already seen the meaning and uses of many such operators in previous units. One special feature offered by C++ is operator overloading. This feature is necessary in a programming language supporting objects oriented features.

Overloading an operator simply means attaching additional meaning and semantics to an operator. It enables an operator to exhibit more than one operation polymorphically, as illustrated below:

You know that addition operator (+) is essentially a numeric operator and therefore, requires two number operands. It evaluates to a numeric value, which is equal to the sum of the two operands. Evidently this cannot be used in adding two strings. However, we can extend the operation of addition operator to include string concatenation. Consequently, the addition operator would work as follows:

"COM" + "PUTER"

Notes

should produce a single string

"COMPUTER"

This act of redefining the effect of an operator is called operator overloading. The original meaning and action of the operator however remains as it is. Only an additional meaning is added to it.

Function overloading allows different functions with different argument list having the same name. Similarly an operator can be redefined to perform additional tasks.

Operator overloading is accomplished using a special function, which can be a member function or friend function. The general syntax of operator overloading is:

```
<return_type> operator <operator_being_overloaded>(<argument list>);
```

Here, operator is the keyword and is preceded by the return_type of the operation.

To overload the addition operator (+) to concatenate two characters, the following declaration, which could be either member or friend function, would be needed:

```
char * operator + (char *s2);
```

## 7.1 Defining Operator Overloading

In C++ the overloading principle applies not only to functions, but to operators too. That is, of operators can be extended to work not just with built-in types but also classes. A programmer can provide his or her own operator to a class by overloading the built-in operator to perform some specific computation when the operator is used on objects of that class. Is operator overloading really useful in real world implementations? It certainly can be, making it very easy to write code that feels natural. On the other hand, operator overloading, like any advanced C++ feature, makes the language more complicated. In addition, operators tend to have very specific meaning, and most programmers don't expect operators to do a lot of work, so overloading operators can be abused to make code unreadable. But we won't do that.



*Did u know?* **What is the advantage of operator overloading?**

By using operator overloading we can easily access the objects to perform any operations.

An Example of Operator Overloading

```
Complex a(1.2,1.3);   //this class is used to represent complex numbers

Complex b(2.1,3); //notice the construction taking 2 parameters for the
real and imaginary part

Complex c = a+b; //for this to work the addition operator must be overloaded
```

The addition without having overloaded operator + could look like this:

```
Complex c = a.Add(b);
```

This piece of code is not as readable as the first example though—we're dealing with numbers, so doing addition should be natural. (In contrast to cases when programmers abuse this technique, when the concept represented by the class is not related to the operator—ike using + and - to add and remove elements from a data structure. In this cases operator overloading is a bad idea, creating confusion.)

Notes

In order to allow operations like Complex c = a+b, in above code we overload the "+" operator. The overloading syntax is quite simple, similar to function overloading, the keyword operator must be followed by the operator we want to overload:

```
class Complex
{
public:
       Complex(double re,double im)
                  :real(re),imag(im)
                 {};
           Complex operator+(const Complex& other);
           Complex operator=(const Complex& other);
private:
         double real;
         double imag;
};
Complex Complex::operator+(const Complex&   other)
{
      double result_real = real + other.real;
      double result_imaginary = imag + other.imag;
      return Complex( result_real, result_imaginary );
}
```

The assignment operator can be overloaded similarly. Notice that we did not have to call any accessor functions in order to get the real and imaginary parts from the parameter other since the overloaded operator is a member of the class and has full access to all private data. Alternatively, we could have defined the addition operator globally and called a member to do the actual work. In that case, we'd also have to make the method a friend of the class, or use an accessor method to get at the private data:

```
friend Complex operator+(Complex);
Complex operator+(const Complex   &num1, const Complex &num2)
{
      double result_real = num1.real + num2.real;
      double result_imaginary = num1.imag + num2.imag;
      return Complex( result_real, result_imaginary );
}
```

**Notes**



*Task* Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables. Discuss.

Why would you do this? when the operator is a class member, the first object in the expression must be of that particular type. It's as if you were writing:

```
Complex a( 1, 2 );

Complex a( 2, 2 );

Complex c = a.operator=( b );
```

when it's a global function, the implicit or user-defined conversion can allow the operator to act even if the first operand is not exactly of the same type:

```
Complex c = 2+b;          //if the integer 2  can be converted by the Complex
class, this expression is valid
```

By the way, the number of operands to a function is fixed; that is, a binary operator takes two operands, a unary only one, and you can't change it. The same is true for the precedence of operators too; for example the multiplication operator is called before addition. There are some operators that need the first operand to be assignable, such as : operator=, operator(), operator[] and operator->, so their use is restricted just as member functions (non-static), they can't be overloaded globally. The operator=, operator& and operator, (sequencing) have already defined meanings by default for all objects, but their meanings can be changed by overloading or erased by making them private.

Another intuitive meaning of the "+" operator from the STL string class which is overloaded to do concatenation:

```
string prefix("de");

string word("composed");

string composed = prefix+word;
```

Using "+" to concatenate is also allowed in Java, but note that this is not extensible to other classes, and it's not a user defined behavior. Almost all operators can be overloaded in C++:

```
+   -   *   /   %   ^   &   |

~   !   ,   =   <   >   <=  >=

++  --  <<  >>  ==  !=  &&  ||

+=  -=  /=  %=  ^=  &=  |=  *=

<<= >>= []  ()  ->  ->* new delete
```

The only operators that can't be overloaded are the operators for scope resolution (::), member selection (.), and member selection through a pointer to a function(.*). Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general pointers. The standard behavior of operators for built-in (primitive) types cannot be changed by overloading, that is, you can't overload operator+(int,int).

The logic (boolean) operators have by the default a short-circuiting way of acting in expressions with multiple boolean operations. This means that the expression:

```
if(a && b && c)
```

Notes

will not evaluate all three operations and will stop after a false one is found. This behavior does not apply to operators that are overloaded by the programmer.

Even the simplest C++ application, like a "hello world" program, is using overloaded operators. This is due to the use of this technique almost everywhere in the standard library (STL). Actually the most basic operations in C++ are done with overloaded operators, the IO(input/output) operators are overloaded versions of shift operators(<<, >>). Their use comes naturally to many beginning programmers, but their implementation is not straightforward. However a general format for overloading the input/output operators must be known by any C++ developer. We will apply this general form to manage the input/output for our Complex class:

friend ostream &operator<<(ostream &out, Complex c)  //output

```
{
        out<<"real part: "<<real<<"\n";
        out<<"imag part: "<<imag<<"\n";
       return out;
}
```

friend istream &operator>>(istream &in, Complex &c)  //input

```
{
        cout<<"enter real part:\n";
       in>>c.real;
        cout<<"enter imag part: \n";
       in>>c.imag;
       return in;
}
```



*Notes* Note that the use of the friend keyword in order to access the private members in the above implementations. The main distinction between them is that the operator>> may encounter unexpected errors for incorrect input, which will make it fail sometimes because we haven't handled the errors correctly.

A important trick that can be seen in this general way of overloading IO is the returning reference for istream/ostream which is needed in order to use them in a recursive manner:

```
Complex a(2,3);
Complex b(5.3,6);
cout<<a<<b;
```

## Self Assessment

Fill in the blanks:

1. The …….. operator works by giving the value of one variable to another variable of the same type or closely similar.
2. Operators are overloaded in C++ by creating operator functions either as a member or as a …….. of a class.

Notes

3. Overloading an operator simply means attaching additional meaning and semantics to an…….. .
4. Overloading assumes you specify a behavior for an operator that acts on a user defined type and it can't be used just with general……. .
5. The operator keyword declares a function specifying what operator-symbol means when applied to instances of a…….. .

## 7.2 Rules for Overloading Operators

To overload any operator, we need to understand the rules applicable Let us revise some of them which have already been explored.

Following are the operators that cannot be overloaded.

**Table 7.1: Operators that cannot be Overloaded**

| Operator | Purpose |
|---|---|
| . | Class member access operator |
| .* | Class member access operator |
| :: | Scope Resolution Operator |
| ?: | Conditional Operator |
| Sizeof | Size in bytes operator |
| # | Preprocessor Directive |
| = | Assignment operator |
| 0 | Function call operator |
| 0 | Subscripting operator |
| -> | Class member access operator |

1. Operators already predefined in the C++ compiler can be only overloaded. Operator cannot change operator templates that is for example the increment operator ++ is used only as unary operator. It cannot be used as binary operator.
2. Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```
class integer
{intx, y;
  public:
  int operator + ();
}
int integer: : operator + ( )
{
  return (x-y);
}
```

3. Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).

Notes

4. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

**Table 7.2**

| Operator to Overload | Arguments passed to the Member Function | Argument passed to the Friend Function |
|---|---|---|
| Unary Operator | No | 1 |
| Binary Operator | 1 | 2 |

5. Overloaded operators must either be a non-static class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type.



*Example:*

```
class  Point
{
public:
      Point  operator<( Point  & );   // Declare a member operator
                                    //  overload.
      // Declare addition operators.
    friend Point operator+( Point&, int );
      friend Point operator+( int, Point& );
};
int  main()
{
}
```

## 7.3 Overloading Unary Operators

In case of unary operator overloaded using a member function no argument is passed to the function whereas in case of a friend function a single argument must be passed.

Following program overloads the unary operator to negate an object. The operator function defined outside the class negates the individual data members of the class integer.

```
#include  <iostream.h>
#include  <conio.h>
class  integer
{
       int  x,y,z;
       public:
       void  getdata(int  a,  int  b,  int  c);
```

**Notes**

```
        void disp(void);
        void operator- ();   // overload unary operator minus
};
void integer::getdata(int a, int b, int c)
{
        x=a; y=b; z=c;
}
void integer::disp(void)
{
        cout << x << " ";
        cout<< y<<" ";
        cout<< z<< "\n";
}
void integer::operator- () // Defining operator- ()
{
        x = -x;  y = -y;  z = -z;
}
void main ()
{
        integer S;
        S.getdata(11,-21,-31);
        Cout<< "S: ";
        S.disp();
        -S;
        cout<<"-S : ";
        S.disp();
        getch();
}
```

You should see the following output.

S: 11 -21 -31

-S:-11 21 31



*Caution* The function written to overload the operator is a member function. Hence no argument is passed to the function. In the main() function, the statement -S invokes the operator function which negates individual data elements of the object S.

The same program can be rewritten using friend function. This is demonstrated in the following Program. In this program we define operator function to perform unary subtraction using a friend function.

**Notes**

```
#include <iostream.h>
#include <conio.h>
class integer
{
       intx;
       int y;
       intz;
       public:
       void getdata(int a, int b, int c);
       void disp(void);
       friend void operator- (integer &s );     // overload unary minus
};
void integer::getdata (int a, int b, int c)
{
       x = a; y = b; z = c;
}
void integer::disp(void)
{
       cout << x << " ";
       cout << y <<" ";
       cout << z << "\n";
}
void operator- (integer &s )              // Defining operator- ()
{
       s.x = -s.x;
       s.y = -s.y;
       s.z = -s.z;
}
void main ()
{
       integer S;
       S.getdata(11 ,-21 ,-31);
       Cout<< "S: ";
       S.disp();
       -S;                          //activates operator-()
       cout<<"-S: ";
       S.disp();
       getch();
}
```

Notes

You should see the following output.

S: 11 -21 -31

-S: -11 21 31



*Notes* Note that how only one argument is passed to the friend function. The operator function declared as friend is not the property of the class. Hence when we define this friend function, we should pass the object of the class on which it operates.

### Self Assessment

Fill in the blanks:

6. By overloading operators, we can control or define how an operator should operate on …….. with respect to a class.
7. Overloaded operators must either be a …….. class member function or a global function.
8. Operators obey the precedence, grouping, and number of operands dictated by their typical use with……. .
9. The positive (+), negative (–) and logical not (!) operators all are unary operators, which means they only operate on one……... .

## 7.4 Overloading Binary Operators

Binary Operators are operators, which require two operands to perform the operation. When they are overloaded by means of member function, the function takes one argument, whereas it takes two arguments in case of friend function. This will be better understood by means of the following program.

The following program creates two objects of class integer and overloads the + operator to add two object values.

```
#include <iostream.h>

#include <como.h>

class integer

{

private:

int val;

public:

integer();

integer(int one );

integer operator+ (integer objb);

void disp();
```

**Notes**

```
};
integer::integer()
{
        val = 0;
}
integer::integer(int one)
{
        val = one;
}
integer integer::operator+ (integer objb)
{
        integer objsum;
        objsum.val = val + objb.val;
        return (objsum);
}
void integer::disp()
{
        cout<< "value ="<< val<< endl;
}
void main()
{
        integer obj1(11);
        integer obj2(22);
        integer objsum;
        objsum = obj1 + obj2;
        obj1.disp();
        obj2.disp();
        objsum.disp();
        getch();
}
```

You should see the following output.

value = 11

value = 22

value = 33

Notes

### 7.4.1 Using Friend Function

The following program is the same as previous one. The only difference is that we use a friend function to find the sum of two objects. Note that an argument is passed to this friend function.

```
#include <iostream.h>
#include <conio.h>
class integer
{
private:
int val;
public:
integer();
integer(in tone);
friend integer operator+ (integer obja, integer objb);
void disp();
};
integer::integer()
{
       val = 0;
}
integer:: integer(int one)
{
       val = one;
}
integer operator+ (integer obja, integer objb)
{
       integer objsum;
       objsum.val = obja.val + objb.val;
       return(objsum);
}
void integer::disp()
{
       cout<< "value ="<< val <<endl;
}
void main()
{
       integer obj1(11);
       integer obj2(22);
```

Notes

```
        integer objsum;
        objsum = obj1 + obj2;
        obj1.disp();
        obj2.disp();
        objsum.disp();
        getch();
}
```

You should see the following output.

value = 11

value = 22

value = 33

Friend function being a non-member function does not belong to any class. This function is invoked like a normal function. Hence the two objects that are to be added have to be passed as arguments exclusively.



*Task* Analyze the uses of friend function in operator overloading.

## 7.4.2 Using Member Function

In the unit on overloading the arithmetic operators, you learned that when the operator does not modify it's operands, it's best to implement the overloaded operator as a friend function of the class. For operators that do modify their operands, we typically overload the operator using a member function of the class.

Overloading operators using a member function is very similar to overloading operators using a friend function. When overloading an operator using a member function:

1. The leftmost operand of the overloaded operator must be an object of the class type.
2. The leftmost operand becomes the implicit *this parameter. All other operands become function parameters.

Most operators can actually be overloaded either way, however there are a few exception cases:

3. If the leftmost operand is not a member of the class type, such as when overloading operator+(int, YourClass), or operator<<(ostream&, YourClass), the operator must be overloaded as a friend.
4. The assignment (=), subscript ([]), call (()), and member selection (->) operators must be overloaded as member functions.

### *Overloading the unary negative (–) operator*

The negative operator is a unary operator that can be implemented using either method. Before we show you how to overload the operator using a member function, here's a reminder of how we overloaded it using a friend function:

```
class Cents
```

**Notes**

```
{
private:
int m_nCents;
public:
Cents(int nCents) { m_nCents = nCents; }
// Overload -cCents
friend Cents operator-(const Cents &cCents);
};
// note: this function is not a member function!
Cents operator-(const Cents &cCents)
{
return Cents(-cCents.m_nCents);
}
```

Now let's overload the same operator using a member function instead:

```
class Cents
{
private:
int m_nCents;
public:
Cents(int nCents) { m_nCents = nCents; }
// Overload -cCents
Cents operator-();
};
// note: this function is a member function!
Cents Cents::operator-()
{
      return Cents(-m_nCents);
}
```

You'll note that this method is pretty similar. However, the member function version of operator- doesn't take any parameters! Where did the parameter go? In the lesson on the hidden this pointer, you learned that a member function has an implicit *this pointer which always points to the class object the member function is working on. The parameter we had to list explicitly in the friend function version (which doesn't have a *this pointer) becomes the implicit *this parameter in the member function version.



*Caution* Remember that when C++ sees the function prototype Cents Cents::operator-();, the compiler internally converts this to Cents operator-(const Cents *this), which you will note is almost identical to our friend version Cents operator-(const Cents &cCents)!

### *Overloading the binary addition (+) operator*

Notes

Let's take a look at an example of a binary operator overloaded both ways. First, overloading operator+ using the friend function:

```
class Cents
{
private:
int m_nCents;
public:
Cents(int nCents) { m_nCents = nCents; }
// Overload cCents + int
friend Cents operator+(Cents &cCents, int nCents);
int GetCents() { return m_nCents; }
};
// note: this function is not a member function!
Cents operator+(Cents &cCents, intnCents)
{
return Cents(cCents.m_nCents + nCents);
}
```

Now, the same operator overloaded using the member function method:

```
class Cents
{
private:
int m_nCents;
public:
Cents(int nCents) { m_nCents = nCents; }
// Overload cCents + int
Cents operator+(int nCents);
int GetCents() { return m_nCents; }
};
// note: this function is a member function!
Cents Cents::operator+(int nCents)
{
return Cents(m_nCents + nCents);
}
```

Our two-parameter friend function becomes a one-parameter member function, because the leftmost parameter (cCents) becomes the implicit *this parameter in the member function version.

Notes

Most programmers find the friend function version easier to read than the member function version, because the parameters are listed explicitly. Furthermore, the friend function version can be used to overload some things the member function version can not. For example, friend operator+(int, cCents) can not be converted into a member function because the leftmost parameter is not a class object.

However, when dealing with operands that modify the class itself (eg. operators =, +=, -=, ++, -, etc…) the member function method is typically used because C++ programmers are used to writing member functions (such as access functions) to modify private member variables. Writing friend functions that modify private member variables of a class is generally not considered good coding style, as it violates encapsulation.

### Self Assessment

Fill in the blanks:

10. When the function defined for the binary operator overloading is a friend function, then it uses …… arguments.
11. Operator overloading adds new functionality to its existing …….. .
12. The program will be efficient and readable only if …….. is used only when necessary.
13. Binary operator overloading, as in unary operator overloading, is performed using a …… operator.

## 7.5 Manipulation of Strings using Operator Overloading

ANSI C implements strings using character assays, pointers and string functions. There are no operators for manipulating the strings. There is no direct operator that could act upon the strings. Although, these limitations exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers for e.g. we shall be able to use the statement like

```
String3 = string1 + string2;
```

The following program to overload the '+' operator to append one string to another

```
#include<iostream.h>
#include<conio.h>
class str
{
char *name;
public:
        str();
        str(char *);
        void get();
        void show();
        str operator + (str);
};
```

**Notes**

```
str::str()
{
        name = new char[1];
}
str::str(char *a)
{
        int i = strlen(a);
        name = new char[i + 1];
        strcpy(name, a);
}
str str::operator + (str s)
{
        int i = strlen(name) + strlen(s.name);
        str tmp;
        tmp.name = new char[i+1];
        strcpy(tmp.name, name);
        strcat(tmp.name, s.name);
        return tmp;
}
void str::get()
{
cin >> name;
}
void str::show()
{
cout << name;
}
void main()
{
clrscr();
        str S3;
        str S1("hello");
        str S2("monty");
        S3 = S1 + S2;
        S3.show();
}
```

Notes



*Did u know?* **How to find a String within a String?**

Searching a string within another string is a breeze with std::string. The Standard Library defines several specialized overloaded versions of string::find() that take const string&, const char *, or just char as a sought-after value (the substring).

## Self Assessment

Fill in the blanks:

14. There is no direct operator that could act upon the……... .

15. String objects are a special type of container, specifically designed to operate with sequences of……... .



*Caselet*

### Every Inch of Space Counts

At 6:30 p.m., Ramesh, a 55-year-old-accountant, is leaving the office and his son Pratik is heading to work at the call centre unit of the same company. They both work in the same building but meet each other only on weekends.

Top executives of their departments have independently discussed their requirements with the CFO for approving a larger facility due to the growth in business in their units. In an ideal scenario, they should expand into each other's existing location. That is the magic of "seat utilization".

Companies have always preferred to keep the seat utilization ratio (employees/seats) less than 1.0 and built the extra capacity in the facility to accommodate future expansion, minimum for the next two years.

The concept of working in multiple shifts has existed right from the early days of mass production, where the most expensive heavy machinery needed to be run 24×7 for maximum utilization.

If we take 160 hours(8 hrs day for 20 days) as the office hours per person per month, and each seat is available for 720 hrs (24 hrs per day for 30 days), it means an absolute mathematical maximum limit seat utilization of 4.5 (720/160).

By apportioning the leaves of the employee throughout the week (even during the week days) and thus enforcing forced absenteeism in every working day, one of the large teams in a leading BPO player has reached the seat utilization of more than 3.

**Top 4 Cost Components**

Currently, in the IT/ITES sectors, the top-four cost components are salary, space, technology infrastructure and support services.

Salary and technology infrastructure are market-driven factors and comparable in most companies of equal status.

The cost of support services is directly proportional to the employee strength and space. Real-estate prices have jumped more than 200 per cent in the last three years in all major

*Contd...*

Notes

cities in India; allowing more people to work in the same area reduces the per unit cost of space and support services.

Most of the ITES (BPO/Call Centre) players have achieved an average seat utilization ratio of 1.4 (author's estimate).

However, IT services (Software) has not been much affected by seat utilization as it continues to attract higher billing rates and earns enough margins to sustain the large facility.

With the dollar weakening every month, demand of salary hike in Indian rupees, customers unwilling for price escalation and scarcity of quality infrastructure, every one needs to think towards higher seat utilization.

Ever since the launch of C++ programming in 1990, the popular object-oriented software, there are numerous examples where a project is broken into independent modules and each module is worked with different teams across geographies to shorten delivery time. If one can work on modules across geographies, what prevents a business from working in multiple shifts?

IT companies need to design attractive incentive packages for those employees who are willing to work in night shift, so that they can free lots of valuable space for growth.

**Change of Mindset Required**

Moving towards higher seat utilisation requires change of management mindset, especially in judging the performance of employees, with well-defined and measurable traits, to track productivity and quality, instead of duration in the office.

If an employee spends longer duration in office, then one will notice a drop in productivity and too less time will result over productivity, which he/she needs to continue.

A team that has continuous over-utilisation, more than 100 per cent, would need immediate management attention, else either it will result in service breakdown or abnormal attrition.

Higher seat utilisation not only helps in keeping costing control but also results in indirect advantages, such as improving traffic decongestion and preserving trees, on which new facilities might have come up.

**Extension to other Fields**

The concept of seat utilization can be extended to many fields, for instance, sharing of doctors' chambers; since visiting hours are limited to four-five hours per day, the same premises can be shared by non-competing areas of expertise (Ophthalmologists and Orthopaedics). Schools run for six hours a day and can be easily extended to two shifts. India has land/people ratio 11 times lower compared to the US, it makes more sense to optimally utilize every inch of space.

## 7.6 Summary

- In this unit, we have seen how the normal C++ operators can be given new meanings when applied to user-defined data types.
- The keyword operator is used to overload an operator, and the resulting operator will adopt the meaning supplied by the programmer.
- Closely related to operator overloading is the issue of type conversion. Some conversions take place between user defined types and basic types.

Notes

- Two approaches are used in such conversion: A one argument constructor changes a basic type to a user defined type, and a conversion operator converts a user-defined type to a basic type.
- When one user-defined type is converted to another, either approach can be used.

## 7.7 Keywords

***Operator Overloading:*** Attaching additional meaning and semantics to an operator. It enables to exhibit more than one operations polymorphically.

***Strings:*** The C++ strings library provides the definitions of the basic_string class, which is a class template specifically designed to manipulate strings of characters of any character type.

***Unary Operators:*** Unary operators operate on one operand (variable or constant). There are two types of unary operators- increment and decrement.

## 7.8 Review Questions

1. Overload the addition operator (+) to assign binary addition. The following operation should be supported by +.

   110010 + 011101 = 1001111

2. What will be the output of the following program snippet? Explain.

```
int  x;
        float  y  =  11.1883;
        x=y;
cout<<x;
cout<<y;
```

3. Which operators are not allowed to be overloaded?
4. What are the differences between overloading a unary operator and that of a binary operator? Illustrate with suitable examples.
5. Why is it necessary to convert one data type to another? Illustrate with suitable examples.
6. How many arguments are required in the definition of an overloaded unary operator?
7. When used in prefix form, what does the overloaded + + operator do differently from what it does in postfix form?
8. Write the complete definition of an overloaded + + operator that works with the string class from the STRPLUS example and has the effect of changing its operand to uppercase. You can use the library function toupper ( ), which takes as its only argument the character to be changed, and returns the changed character.
9. Write a note on unary operators.
10. What are the various rules for overloading operators?

## Answers: Self Assessment

Notes

1. assignment
2. Friend Function
3. operator
4. pointers
5. class
6. data
7. non-static
8. built-in types
9. operand
10. two
11. operators
12. operator overloading
13. keyword
14. strings
15. characters

## 7.9 Further Readings



*Books*

E Balagurusamy; *Object-Oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://www.mochima.com/tutorials/strings.html

http://www.exforsys.com/tutorials/c-plus-plus/operator-overloading-part-ii.html

**Notes**

# Unit 8: Type Conversion

**CONTENTS**

Objectives

Introduction

8.1 Type Conversions

8.2 Basic Type to Class Type

8.3 Class Type to Basic Type

8.4 Class Type to another Class Type

8.5 Summary

8.6 Keywords

8.7 Review Questions

8.8 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the type conversions
- Describe the basic type to class type
- Explain the class type to basic type
- Discuss the class type to another type

## Introduction

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler. Explicit type conversions (casts) are used when a programmer want to get around the compiler's typing system; for success in this endeavour, the programmer must use them correctly. Problems which the compiler avoids may arise, such as if the processor demands that data of a given type be located at certain addresses or if data is truncated because a data type does not have the same size as the original type on a given platform. Explicit type conversions between objects of different types lead, at best, to code that is difficult to read.

## 8.1 Type Conversions

In a mixed expression constants and variables are of different data types. The assignment operations cause automatic type conversion between the operand as per certain rules.

The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left.

Consider the following example:

```
int  x;
```

Notes

```
float y = 20.123;
x=y;
```

This converts float variable y to an integer before its value assigned to x. The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type.



*Caution* But of objects are of different data types we must apply conversion rules for assignment.

There are three types of situations that arise where data conversion are between incompatible types.

1. Conversion from built in type to class type.
2. Conversion from class type to built in type.
3. Conversion from one class type to another.

### Self Assessment

Fill in the blanks:

1. Implicit type conversion is done automatically by the …… whenever data from different types is intermixed.
2. The type of data to the right of an assignment operator is automatically converted to the data type of variable on the ……....
3. An implicit conversion is performed automatically by the compiler when an expression needs to be …….. into one of its compatible types.
4. The desired data type is simply placed in …….. to the left of the expression that needs to be converted.
5. That cast is therefore more likely to execute subtle conversion ……. if used incorrectly.
6. If a user-defined conversion can give rise to exceptions or loss of information, then that conversion should be defined as an …….. conversion.

## 8.2 Basic Type to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defect type conversion from the argument's type to the constructor's class type

Consider the following constructor:

```
string::string(char*a)
 {
   length = strlen(a);
   name=new char[len+1];
```

Notes

```
    strcpy(name,a);
  }
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the constructor in the class string, it can be used for conversion from char* type to string type.



*Example:*

string s1, s2;

char* name1 = "Good Morning";

char* name2 = "STUDENTS" ;

s1 = string(name1);

s2 = name2;

The program statement

s1 = string (name1);

first converts name1 from char* type to string type and then assigns the string type values to the object s1. The statement

s2 = name2;

performs the same job by invoking the constructor implicitly.

Consider the following example

```
class  time
{
        int  hours;
        int  minutes;
         public:
        time  (int  t)  II  constructor
          {
         hours  =  t  /  60;                    //t  is  inputted  in  minutes
            minutes  =  t  %  60; .
          }
        };
```

In the following conversion statements:

```
time T1;                    //object T1 created
int period = 160;
T1 = period;                //int to class type
```

The object T1 is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours of T1 will have value 2 arid minutes will have a value of 40 denoting 2 hours and 40 minutes.

Notes

*Notes* Note that the constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded = operator.

### Self Assessment

Fill in the blanks:

7. You can define a member function of a class, called a conversion function that converts from the type of its class to another…….. .
8. Classes, enumerations , …….. , function types, or array types cannot be declared or defined in the conversion_type.
9. The conversion operator can be defined either inside the …….. type.
10. Where the operator is the required keyword and type is the required…….. .

## 8.3 Class Type to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename()
{
        //Program statement
}
```

This function converts a class type data to type name. For example, the operator double() converts a class object to type double, in the following conversion function:

```
vector:: operator double()
    {
        double sum = 0;
            for(int I = 0; i<size; i++)
               sum = sum + v[i] * v[i ];   //scalar magnitude
             return sqrt(sum);
    }
```

The casting operator should satisfy the following conditions.

1. It must be a class member.
2. It must not specify a return type.
3. It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function. As a result function does not need an argument.

Notes

In the string example discussed earlier, we can convert the object string to char* as follows:

```
string::operator  char*()
{
        retum(str);
  }
```



*Did u know?* **What is dynamic_cast?**

Dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

### Self Assessment

Fill in the blanks:

11. Conversion functions have no …… , and the return type is implicitly the conversion type.
12. ……. functions can be inherited.
13. C++ allows us to define an overloaded …….. operator that convert a class type data to basic type.

## 8.4 Class Type to another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.



*Example:*

Objl = Obj2 ; //Obj1 and Obj2 are objects of different classes.

Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

We studied that the casting operator function

Operator typename()

Converts the class object of which it is a member to typename. The type name may be a built-in type or a user defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source class and the result is given to the destination class object.

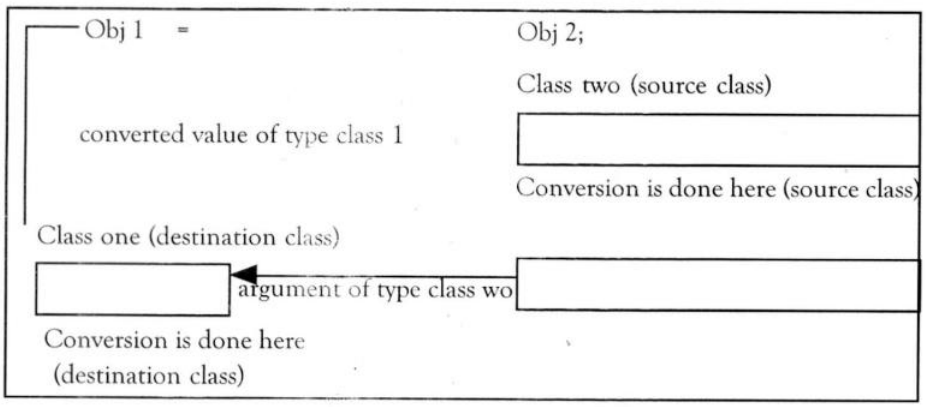
Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. The argument belongs

Notes

to the source class and is passed to the destination class for conversion. Therefore the conversion constructor must be placed in the destination class.

The following Figure 8.1 illustrates the above two approaches.

**Figure 8.1: Conversions between Objects**




The Table 8.1 summarizes all the three conversions. It shows that the conversion from a class to any other type (or any other class) makes use of a casting operator in the source class. To perform the conversion from any other type or class to a class type, a constructor is used in the destination class.

**Table 8.1**

| Conversion | Conversion takes place in | |
|---|---|---|
| | Source class | Destination class |
| Basic → class | Not applicable | Constructor |
| Class → Basic | Casting operator | Not applicable |
| Class → class | Casting operator | Constructor |

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument.

*Task* 'Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.' Explain this statement.

Consider the following example of an inventory of products in a store. One way of keeping record of the details of the products is to record their code number, total items in the stock and the cost of each item. Alternatively we could just specify the item code and the value of the item

**Notes**

in the stock. The following program uses classes and shows how to convert data of one type to another.

```
#include<iostream.h>
#inc1ude<conio.h>
class stock2;
class stockl
{
int code,item;
float price;
public:
stockl(int a,int b,float c)
{
code=a;
item=b;
price=c;
}
void disp()
{
cout<< "code" <<code <<"\n";
cout<< "Items" <<item <<"\n";
cout<< "Price per item ₹" <<price <<"\n";
}
int getcode()
{return code;}
int getitem()
{return item;}
int getprice()
{return price;}
operator float()
{
retum(item*price);
}
};
class stock2
{
int code;
float val;
```

**Notes**

```
public:
stock20
{
code=0;val=0;
}
stock2(int  x,float  y)
{
code=x;val=y;
}
void  disp()
{
cout<<  "code"  <<code  <<"\n";
cout<<"Total  Value  ₹"  <<val  <<"\n";
}
stock2(stockl  p)
{
code=p.getcode();
val=p.getitem()  *  p.getprice();
}
};
void  main()
{
stock1  i1(101,  10,125.0);
stock2  i2;  .
float  tot_val;
tot_val=i1  ;
i2=i1  ;
cout<<"  Stock  Details-stockl-type"  <<"\n";
i1.disp();
cout<<  "  Stock  value"  <<  "\n";
cout<<  tot_val<<  "\n";
cout<<  "  Stock  Details-stock2-type"  <<  "\n";
i2.disp();
getch();
}
```

Notes

You should get the following output.

Stock Details-stock1-type

code 101

Items 10

Price per item ₹125

Stock value

1250

Stock Details-stock2-type

code 10 1

Total Value ₹1250

## Self Assessment

Fill in the blanks:

14. Typecasting is making a variable of one type, such as an int, act like another type, a …….. , for one single operation.

15. One special case of implicit type conversion is type promotion, where the compiler automatically expands the …….. representation of objects of integer or floating-point types.



*Caselet*

### Learn to Think Like the Computer Thinks

To learn to program a computer you need to learn to think like the computer thinks, advises Jim Messinger in Starting Out With Programming Logic & Design, and introduces readers first to friendly definitions of terms such as variable, constant, pseudocode, counter and so on.

Thus, you'd learn that 'flag' is related to flagging of data. It is similar to flagging in car races where the flag tells you to take a specific action, explains Messinger.

"In a computer, the flag is on when it is 1 and off when it is zero. In car races, there are a variety of flags: yellow, white, checkered, and so on. Each of these could be represented in a computer by an appropriately named variable and could be set to 0 or 1 to indicate to the computer program the state of the car race."

Again, when explaining modules as collections of related tasks, the author uses the automotive example: "Think of a car as a program. Its modules could be the engine, transmission, and body." Remember that modules can become complex, which is when you need sub-modules, such as "fuel system, ignition system, and exhaust system" for the engine.

With Lord Falkland's quote - "When it not necessary to make a decision, it is necessary not to make a decision" - begins a chapter on `selection logic'. If you're familiar with flowchart symbols, you may know selection as what you do when a `diamond' is encountered.

*Contd...*

Notes

The selection structure allows for decisions to be made, explains Messinger. "It has only one entry point, as do all the other structures in computer programming; but after the decision, we can proceed in one of two independent directions."

Somebody who thinks logically is a nice contrast to the real world, is the 'Law of Thumb', as one learns in another chapter on 'compound logic'.

And yet another chapter, which discusses loops, begins with George Eliot's thought, "Iteration, like friction, is likely to generate heat instead of progress."

Hamlet's words, "Yea, from the table of my memory I'll wipe away all trivial fond records," introduce readers to `tables'. Tables are contiguous memory locations in RAM that have a common name, explains the author. A simple example of a four-dimension table is book, he writes, referring to chapter, page, line and column. "A common application of multi-dimension tables is the market survey."

Procedural programming paradigm is not the only choice, and so the book explains two other popular ones, viz. visual or graphical user interface programming, and object-oriented programming.

Easy read, replete with exercises and 'enrichments'.

AIMED at 'empowering productivity for the Java developer', here is the third edition of Beginning J2ME: From Novice to Professional, by Sing Li and Jonathan Knudsen.

The book is about programming "mobile phones, pagers, PDAs, and other small devices", and MIDP (Mobile Information Device Profile), which is part of the Java 2 Platform.

J2ME isn't a specific piece of software or specification, explain the authors. "All it means is Java for small devices."

The market is expanding rapidly for two reasons: "First, developers can write code and have it run on dozens of small devices, without change. Second, Java has important safety features for downloadable code."

MIDP applications are called MIDlets, rhyming with applets and servlets. "Writing MIDlets is relatively easy for a moderately experienced Java programmer," is an enticingly reassuring line. Please note that "the actual development process, however, is a little more complicated for MIDlets than it is for J2SE applications," because of `some additional tweaking and packaging' required.

To make MIDlets as compact as possible, 'obfuscator' is used. This is a tool, "originally designed to foil attempts to reverse engineer compiled bytecode"; it renames classes, member variables, and methods to more compact name; removes unused classes and so on; and inserts illegal or questionable data to confuse decompilers.

A killer application for the wireless is SMS, says the duo. "The ability to send short text messages (up to 160 characters in most cases) between cell-phone users inexpensively is compelling enough. The possibility to send messages directly between J2ME applications running on cellular phones is even more exciting," opine the authors.

A chapter is devoted to 'Bluetooth and OBEX'. The former is "a radio connectivity technology designed for creating Personal Area Networks (PANs)" to help you connect things that are next to you; and the latter is short for Object Exchange, "a communication protocol that enables applications to talk to one another easily over infrared".

Bluetooth networks are formed ad hoc and dynamically when Bluetooth-enabled devices come into proximity of one another, as Li and Knudsen explain. "Technically, a Bluetooth

*Contd...*

Notes

network is a piconet, and it can consist of one master device and up to seven slave devices."

The book guides you to write a MIDlet to create "a simple Bluetooth dating service", something to help break the ice when there are people in the same room who are attracted to each other.

This is how: "You tell the MIDlet the type of date you're looking for, and the MIDlet will use Bluetooth to query anybody who comes near yours for compatibility. Once compatibility is established, your potential date's e-mail address is displayed on your phone."

What happens thereafter is beyond the scope of the book. However, be prepared for failing gracefully, when tuning performance. For, "given the paucity of memory in a typical MIDP device, your application should be prepared for disappointment each time it asks for memory."

And there's a tip as a 'possible strategy in production programming' - that you may "attempt to allocate all memory up front, when the application first starts".

## 8.5 Summary

- A type conversion may either be explicit or implicit, depending on whether it is ordered by the programmer or by the compiler. Explicit type conversions (casts) are used when a programmer want to get around the compiler's typing system; for success in this endeavour, the programmer must use them correctly.
- Used another constructor to build a string type object from a char* type variable.
- The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename()
{
  //Program statement
}
```

- Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

## 8.6 Keywords

***Implicit Conversion:*** An implicit conversion sequence is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration.

***Operator Typename():*** Converts the class object of which it is a member to typename.

## 8.7 Review Questions

1. "In a mixed expression constants and variables are of different data types." Justify this statement with an example.
2. The constructor functions do not support conversion from a class to basic type. Explain with a example.

Notes

3. Discus which statements are does casting operator should satisfy.
4. Explicit type conversions between objects of different types lead, at best, to code that is difficult to read. Discuss.
5. The assignment operations cause automatic type conversion between the operand as per certain rules. Describe.
6. Two standard conversion sequences or two user-defined conversion sequences may have different ranks. Do you agree with this statement? Why or why not?
7. Scrutinize the standard conversion sequences.
8. Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. Explain with and example.
9. There are three types of situations that arise where data conversion are between incompatible types. What are three situations explain briefly.
10. Write a program which the conversion of class type to basic type conversion.

### Answers: Self Assessment

1. compiler
2. left
3. converted
4. parentheses
5. errors
6. explicit
7. specified type
8. typedef names
9. class or struct
10. return type
11. arguments
12. Conversion
13. Casting
14. Char
15. binary

## 8.8 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://www.bogotobogo.com/cplusplus/typecast.php

http://en.wikipedia.org/wiki/Type_conversion

Notes

# Unit 9: Inheritance

**CONTENTS**

Objectives

Introduction

9.1 Defining Derived Class

9.2 Forms of Inheritance

9.2.1 Single Inheritance

9.2.2 Multilevel Inheritance

9.2.3 Multiple Inheritance

9.2.4 Hierarchical Inheritance

9.2.5 Hybrid Inheritance

9.3 Ambiguity in Multiple and Multipath Inheritance

9.3.1 Ambiguity in Multiple Inheritance

9.3.2 Ambiguity in Multipath Inheritance

9.4 Virtual Base Classes

9.5 Overriding Member Function

9.6 Constructors under Inheritance

9.7 Destructors under Inheritance

9.8 Making a Private Member Inheritable

9.9 Summary

9.10 Keywords

9.11 Review Questions

9.12 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the inheritance
- Describe the different types of inheritance
- Explain the ambiguity in multiple and multipath inheritance
- Discuss the virtual base class
- Identify the overriding member function
- Demonstrate the execution of constructor and destructor with inheritance

## Introduction

Reusability of code is a characteristic feature of OOP. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and

Notes

tested, it can be adopted by other programmers. New classes can be defined reusing the properties of existing ones.

The mechanism of deriving a new class from an old one is called 'INHERITANCE'. This is often referred to as 'IS-A' relationship because every object of the class being defined "is" also an object of inherited class type. The old class is called 'BASE' class and the new one is called 'DERIVED' class or sub-class.

## 9.1 Defining Derived Class

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class derivedclassname : access_specifier baseclassname
{
        ……
        ……      // members of derivedclass
};
```

The colon (:) indicates that the derivedclassname class is derived from the baseclassname class. The access_specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the accessibility status of derived features. For example,

```
class xyz                   //base class
{
        //members of xyz
};
class ABC: public xyz              //public derivation
{
//members of ABC
};
class ABC : XYZ             //private derivation (by default)
{
        //members of ABC
};
```

In inheritance, some of the base class data elements and member functions are inherited into the derived class and some are not. We can add our own data and member functions and thus extend the functionality of the base class.



*Notes* Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Notes

### Self Assessment

Fill in the blanks:

1. Inheritance is a mechanism of …….. and extending existing classes without modifying them, thus producing hierarchical relationships between them.
2. Inheritance is almost like embedding an …… into a class.
3. C++ allows you to use one class declaration, known as a base class, as the basis for the declaration of a second class, known as a ……..
4. A member marked as …….. is accessible from member functions of the class and also from member functions of any classes derived from that class.

## 9.2 Forms of Inheritance

### 9.2.1 Single Inheritance

When a class inherits from a single base class, it is referred to as single inheritance. Following program shows single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
int age;
        char name [10];
public:
        void get();
};
void worker::get()
{
        cout << "your name please";
        cin >> name;
        cout << "your age please";
        cin >> age;
}
void worker::show()
{
        cout<<"\nMy name is :"<<name<<"\nMy age is :"<<age;
}
class manager : public worker            //derived class (publicly)
{
```

**Notes**

```
        int now;
public:
        void get();
        void show();
};
void manager::get()
{
        worker::get();              //calling base class get function
        cout<<"\nNumber of workers under you";
        cin >> now;
}
void manager::show()
{
        worker::show();             //calling base class show function
        cout>>"No. of workers under me is: "<<now;
}
main()
{
        clrscr();
        worker W1;
        manager M1;
        M1.get();
        M1.show();
}
```

If you input the following to this program:

```
Your name please
        Ravinder
Your age please
        27
number of workers under you
        30
```

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me is : 30

**Notes**



*Did u know?* **What is directed acyclic graphs?**

Directed acyclic graphs are not unique to single inheritance. They are also used to depict multiple-inheritance graphs.

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker                //Base class declaration
{
       int age;
       char name[10];
public:
       void get();
       void show();
};
void worker::get()
{
       cout<<"\nYour name please:";
       cin>>name;
       cout<<"\nyour age please:";
       cin>>age;
}
void worker : show()
{
cout << "\nMy name is: "<<name<< "\n" << "My age is: "<<age;
}
class manager : worker      //Derived class (privately by default)
{
       int now;
public:
       void get();
       void show();
};
void manager::get()
{
       worker::get();       //calling the get function of base
       cout<<"number of worker under you";
```

Notes

```
        cin>>now;
}
void manager::show()
{
worker::show();
cout << "\n no. of worker under me is : "<<now;
}
main()
{
        clrscr();
        worker w1;
        manager m1;
        m1.get();
        m1.show();
}
```



*Notes* When a derived class publicly inherits the base class, all the public members of the base class also become public to the derived class and the objects of the derived class can access the public members of the base class.

The following program shows the single inheritance using protected derivation

```
        #include<conio.h>
        #include<iostream.h>
        class worker                    //Base class declaration
        {
protected:
                int age; char name[20];
        public:
                void get();
                void show();
        };
        void worker::get()
        {
        cout >> "your name please";
        cin >> name;
        cout << "your age please";
        cin >> age;
        }
```

**Notes**

```
void worker::show()
{
cout << "\n My name is: "<< name << "\nMy age is "<<age
}
class manager : protected worker  // protected inheritance
{
        int now;
public:
        void get();
        void show();
};
void manager::get()
{
        cout << "please enter the name\n";
        cin >> name;
        cout << "please enter the age\n";
//Directly inputting the data
        cin >> age;
//members of base class
        cout <<" please enter the no. of workers under you:";
        cin >> now;
}
void manager::show()
{
cout << "your name is : "<<name<<" and age is : "<<age;
cout <<"\n no. of workers under your are : "<<now;
main()
{
        clrscr();
        manager m1;
        m1.get();
        cout << "\n \n";
        m1.show();
}
```

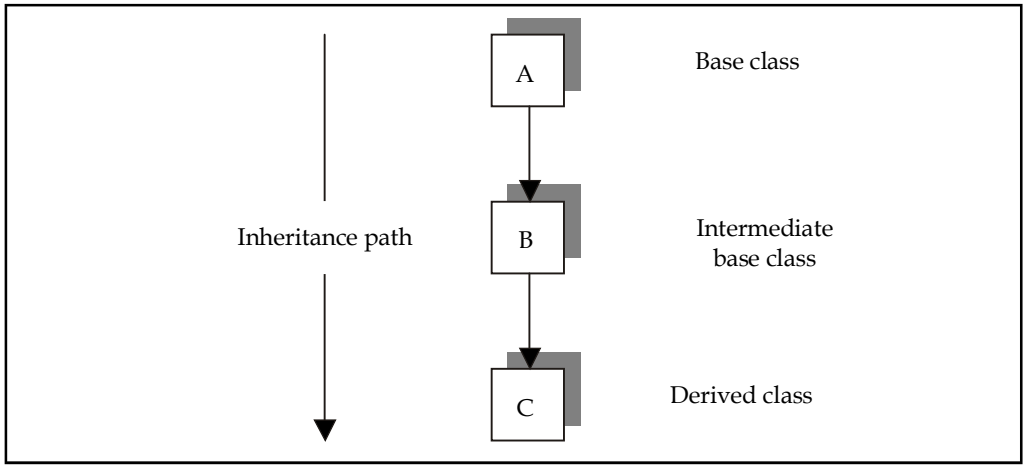


*Task* In a group of four try to analyze the advantages of single inheritance.

### 9.2.2 Multilevel Inheritance

Notes

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITANCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'INHERITANCE*PATH' for e.g.




The declaration for the same would be:

```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.

*Task* In your words write down the definition of Intermediate Base Class.

Following program exhibits the multilevel inheritance.

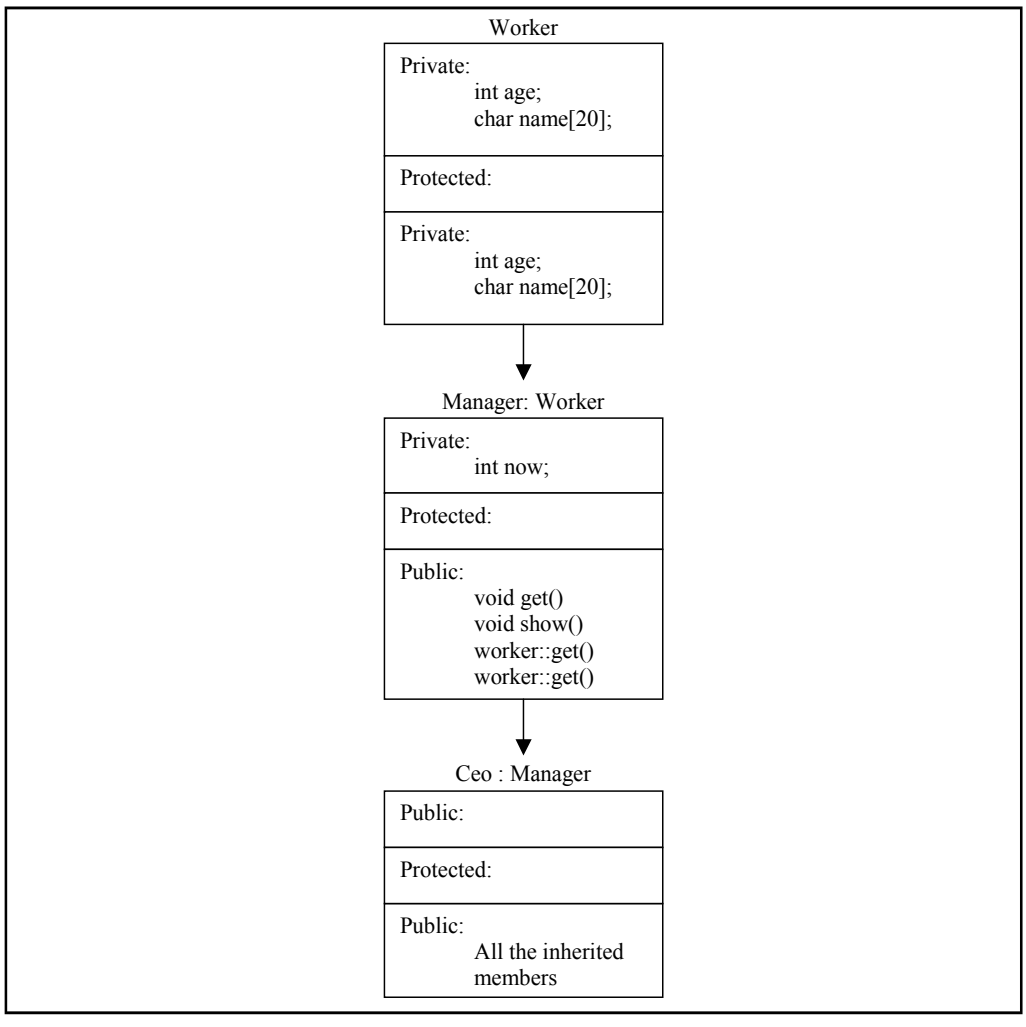
```
# include < iostream.h>
# include < conio.h>
class worker                //Base class declaration
{
```

**Notes**

```
int age;
char name [20];
public;
       void get();
       void show();
}
void worker: get()
{
       cout << "your name please";
       cin >> name;
       cout << "your age please";
}
void worker::show()
{
cout << "In my name is : "<<name<<" In my age is : "<<age;
}
class manager : public worker     //Intermediate base class derived
{                           //publicly from the base class
int now;
public:
       void get();
       void show();
};
void manager::get()
{
worker::get();              //calling get ( ) fn. of base class
cout << "no. of workers under you:";
cin >> now;
}
void manager : : show ( )
{
worker : : show ( );        //calling show ( ) fn. of base class
cout << "In no. of workers under me are: "<< now;
}
class ceo: public manager          //declaration of derived class
{                                  //publicly inherited from the
int nom;                           //intermediate base class
public:
       void get ( );
       void show ( );
}
       void ceo : : get ( )
```

**Notes**

```
{
        manager : : get ( );
        cout << "no. of managers under you are:"; cin >> nom;
}
        manager : : show ( );
        cout << "In the no. of managers under me are: In";
        cout << "nom;
}
main ( )
{
clrscr ( );
        ceo c1;
        c1.get ( ); cout << "\n\n";
        c1.show ( );
}
```



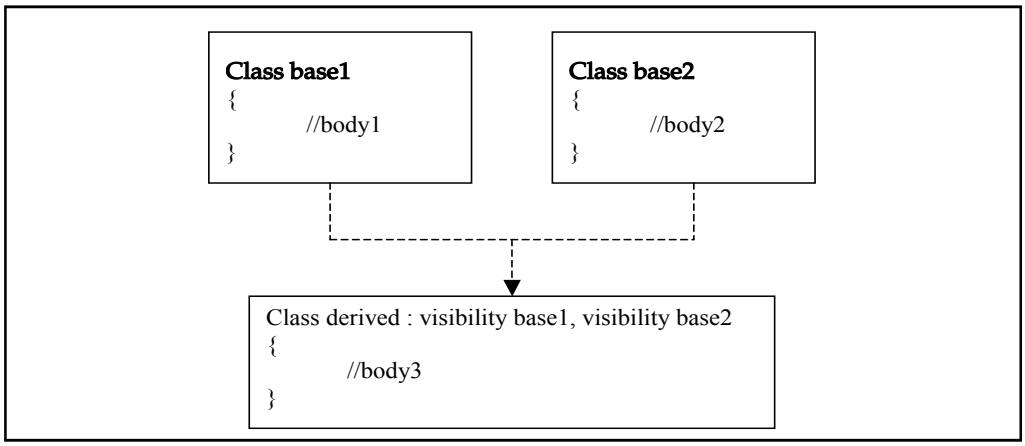
Notes

### 9.2.3 Multiple Inheritance

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITANCE'. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes.

*Caution* It is like the child inheriting the physical feature of one parent and the intelligence of another.

The syntax of the derived class is as follows:




*Task* A direct base class cannot appear in the base list of a derived class more than once. Discuss.

Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

```
# include < iostream.h>
# include < conio.h>
class father                //Declaration of base class1
{
int age;
char name [20];
public:
       void get ( );
       void show ( );
};
void father : : get ( )
{
       cout << "your father name please";
       cin >> name;
```

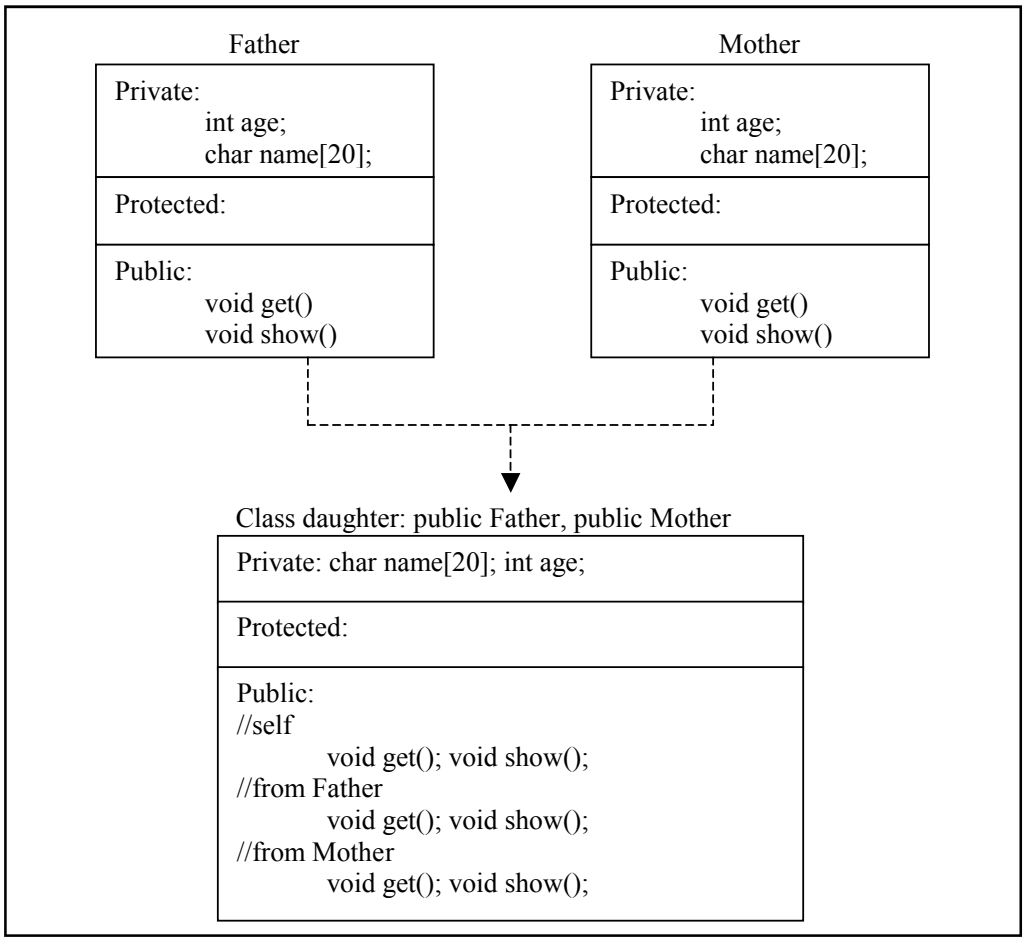
**Notes**

```
                cout << "Enter the age";
                cin >> age;
        }
        void father : : show ( )
        {
        cout<< "In my father's name is:"<<name<< "In my father's age
is: <<age;
        }
        class mother                //Declaration of base class 2
        {
        char name [20];
        int age;
        public:
                void get ( )
                {
                cout << "mother's name please" << "In";
                cin >> name;
                cout << "mother's age please" << "in";
                cin >> age;
                }
                void show ( )
                {
                cout << "In my mother's name is: "<<name;
                cout << "In my mother's age is: "<<age;
class daughter : public father, public mother //derived class:
inheriting
{                                                   //publicly
char name [20];             //the features of both the base class
int std;
public:
        void get ( );
        void show ( );
};
void daughter :: get ( )
{
        father :: get ( );
        mother :: get ( );
        cout << "child's name: ";
        cin >> name;
        cout << "child's standard";
```

**Notes**

```
        cin >> std;
}
void daughter :: show ( )
{
        father :: show ( );
        nfather :: show ( );
        cout << "In child's name is : "<<name;
        cout << "In child's standard: "<<std;
}
main ( )
{
clrscr ( );
        daughter d1;
        d1.get ( );
        d1.show ( );
}
```

Diagrammatic Representation of Multiple Inheritance is as follows:

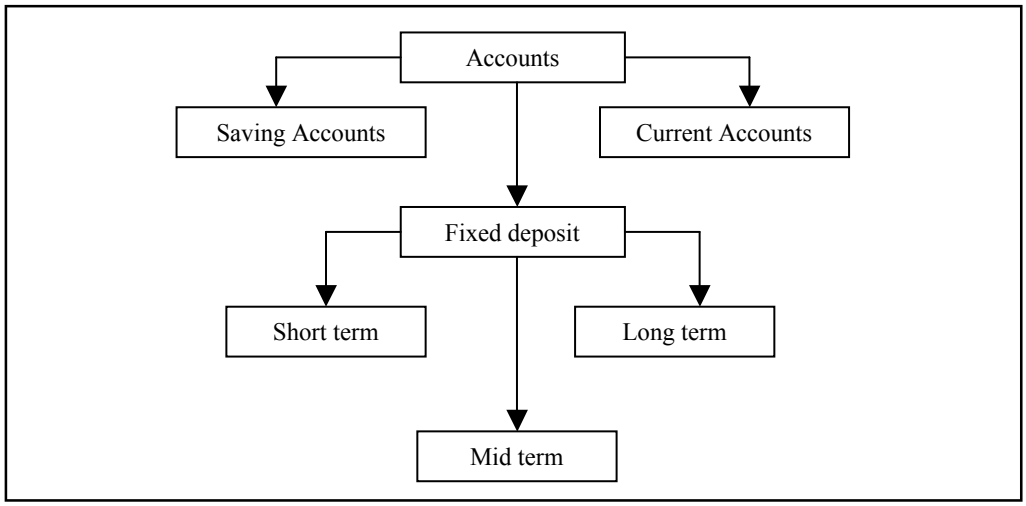


Notes

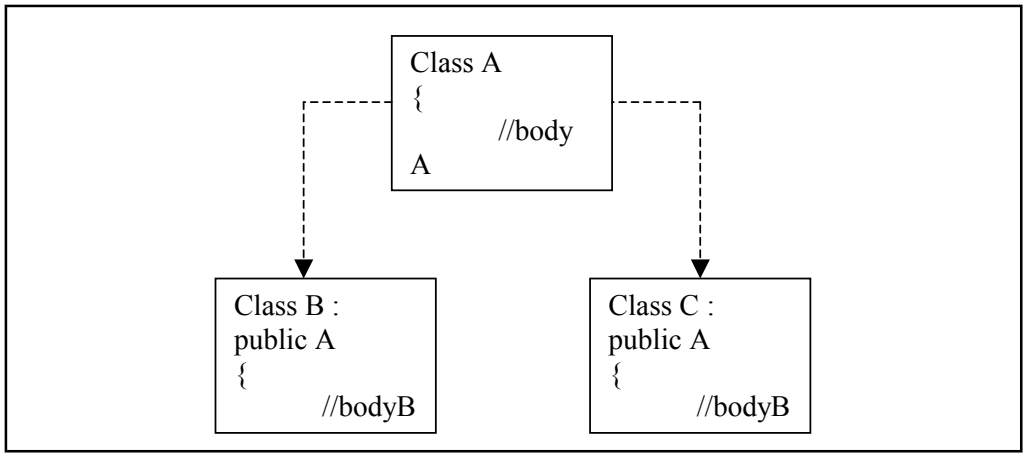
*Example:* If two base classes have a member with the same name, the derived class cannot implicitly differentiate between the two members. Note that, when you are using multiple inheritance, the access to names of base classes may be ambiguous.

### 9.2.4 Hierarchical Inheritance

Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.




In general the syntax is given as




In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

**Notes**



*Did u know?* **What is Polymorphic functions?**

Polymorphic functions are functions that can be applied to objects of more than one type.

//Program to show the hierarchical inheritance

```
# include < iostream.h>
# include < conio.h>
class father                //Base class declaration
{
int age;
char name [15];
public:
        void get ( )
        {
        cout << "father name please"; cin >> name;
        cout << "father's age please"; cin >> age;
        }
        void show ( )
        {
        cout << "In father's name is : "<<name;
        cout << "In father's age is: "<< age;
        }
};
class son : public father          //derived class 1
{
        char name [20];
        int age;
        public;
                void get ( );
                void show ( );
};
void son :: get ( )
{
        father :: get ( );
        cout << "your (son) name please" <<"in"; cin >>name;
        cout << "your age please" <<"In"; cin>>age;
}
void son :: show ( )
{
        father : : show ( );
        cout << "In my name is : "<<name;
        cout << "In my age is : "<< age;
```

Notes

```
}
class daughter : public father          //derived class 2.
{
        char name [15];
        int age;
        public:
                void get ( )
                {
                father : : get ( );
                cout << "your (daughter's) name please In"
cin>>name;
                cout << "your age please In"; cin >>age;
                }
                void show ( )
                {
                father : : show ( );
                cout << "in my father name is: " << name << "
                        In and his age is : "<<age;
                }
};
main ( )
{
        clrscr ( );
                son S1;
                daughter d1;
                S1.get( );
                D1.get( );
                S1.show( );
                D1.show( );
}
```

### 9.2.5 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

//Program to show the simple hybrid inheritance

```
# include < isostream.h>
# include < conio.h>
class student                       //base class declaration
{
```

**Notes**

```
protected:
        int r_no;
public:
        void get _n (int a) {
                r_no - a; }
        void put_n (void)
        {
        cout << "Roll No.: "<< r_no;
        cout << "In";
        }
};
class test : public student
{                                   //Intermediate base class
protected : int part1, par 2;       //(base for result)
public :
        void get_m (int x, int y) {
                part1 = x; part 2 = y; }
        void put_m (void) {
                cout << "marks obtained: "<< "In"
                        << "Part 1 = "<< part1 << "In"
                        << "Part 2 = "<< part2 << "In";
        }
};
class sports                // base for result
{
protected : int score;
public:
        void get_s (int s) {
                score = S1 }
        void put_s (void) {
                cout << "sports wt. : "<< score << "\n\n";
        }
};
class result : public test, public sports//Derived from test
                                                        & sports
{
int total;
```

Notes

```
public:
        void display (void);};
        void result : : display (void)
{
        total = part1 + part2 + score;
        put_n ( );
        put_m ( );
        put_s ( );
        cout << "Total score: "<<total<< "\n";
}
main ( )
{
clrscr ( );
        result S1;
        S1.get_n (347);
        S1.get_m (30, 35);
        S1.get_s (7);
        S1.dciplay ( );
}
```



*Did u know?* **What are the advantages of inheritance?**

Inheritance offers the following advantages:

Development model closer to real life object model with hierarchical relationships

***Reusability*** — facility to use public methods of base class without rewriting the same

***Extensibility*** — extending the base class logic as per business logic of the derived class

***Data hiding*** — base class can decide to keep some data private so that it cannot be altered by the derived class.

## Self Assessment

Fill in the blanks:

5. In "single inheritance," a common form of inheritance, classes have only one ……. class.
6. When a derived class privately inherits a base class, all the public members of the base class become …….. for the derived class.
7. The base class from which each class is derived is declared before the declaration of the …… class.
8. An indirect base class is a base class that does not appear directly in the ……. of the derived class but is available to the derived class through one of its base classes.

Notes

9. A direct base class is a base class that appears directly as a ..... in the declaration of its derived class.

10. ..... inheritance is the combination of types of inheritance.

## 9.3 Ambiguity in Multiple and Multipath Inheritance

### 9.3.1 Ambiguity in Multiple Inheritance

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

typedef int HANDS;
enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;

class Animal
{
   public:
      Animal(int);
      virtual ~Animal() { cout << "Animal destructor...\n"; }
      virtual int GetAge() const { return itsAge; }
      virtual void SetAge(int age) { itsAge = age; }
   private:
     int itsAge;
};

Animal::Animal(int age):
itsAge(age)
{
     cout << "Animal constructor...\n";
}

class Horse : public Animal
{
   public:
      Horse(COLOR color, HANDS height, int age);
      virtual ~Horse() { cout << "Horse destructor...\n"; }
      virtual void Whinny()const { cout << "Whinny!... "; }
      virtual HANDS GetHeight() const { return itsHeight; }
```

**Notes**

```
      virtual COLOR GetColor() const { return itsColor; }
   protected:
      HANDS itsHeight;
      COLOR itsColor;
};

Horse::Horse(COLOR color, HANDS height, int age):
Animal(age),
itsColor(color),itsHeight(height)
{
     cout << "Horse constructor...\n";
}

class Bird : public Animal
{
   public:
      Bird(COLOR color, bool migrates, int age);
      virtual ~Bird() {cout << "Bird destructor...\n";   }
      virtual void Chirp()const { cout << "Chirp... ";   }
      virtual void Fly()const{ cout << "fly! "; }
      virtual COLOR GetColor()const { return itsColor; }
      virtual bool GetMigration() const { return itsMigration; }
   protected:
      COLOR itsColor;
      bool itsMigration;
};

Bird::Bird(COLOR color, bool migrates, int age):
Animal(age),
itsColor(color), itsMigration(migrates)
{
     cout << "Bird constructor...\n";
}

class Pegasus : public Horse, public Bird
{
```

**Notes**

```
    public:
       void Chirp()const { Whinny(); }
       Pegasus(COLOR, HANDS, bool, long, int);
       virtual ~Pegasus() {cout << "Pegasus destructor...\n";}
       virtual long GetNumberBelievers() const
       { return  itsNumberBelievers; }
       virtual COLOR GetColor()const { return Horse::itsColor; }
       virtual int GetAge() const { return Horse::GetAge(); }
    private:
       long itsNumberBelievers;
};

Pegasus::Pegasus(
    COLOR aColor,
    HANDS height,
    bool migrates,
    long NumBelieve,
    int age):
    Horse(aColor, height,age),
    Bird(aColor, migrates,age),
    itsNumberBelievers(NumBelieve)
{
    cout << "Pegasus constructor...\n";
}

int main()
{
    Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
    int age = pPeg->GetAge();
    cout << "This pegasus is " << age << " years old.\n";
    delete pPeg;
    return 0;
}
```



*Did u know?* **How the ambiguities in multiple inheritance are removed?**

You can also avoid this ambiguity by using the base specifier virtual to declare a base class

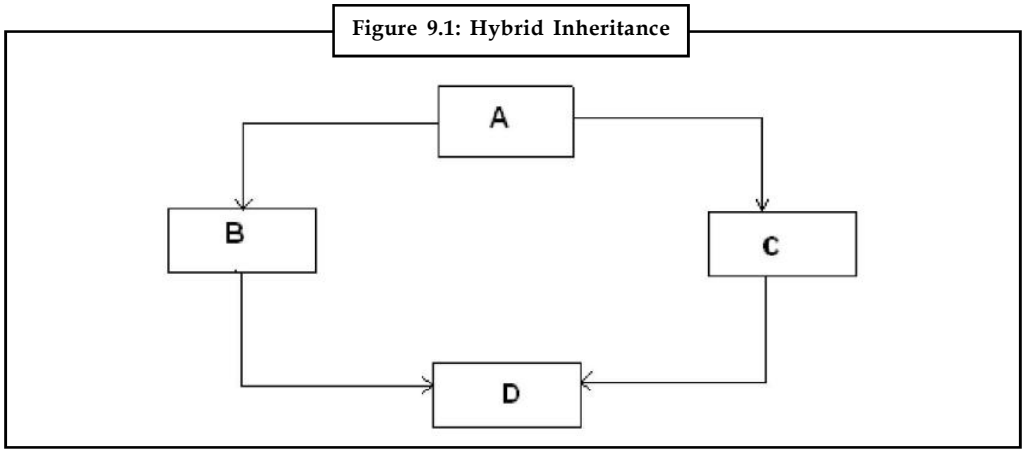
Notes

### 9.3.2 Ambiguity in Multipath Inheritance

Sometimes we need to apply two or more types of inheritance to design a program and this type of inheritance are called hybrid inheritance. But in hybrid inheritance, if multiple paths exit between a base class and derived class through different intermediate classes, then derived class inherits the members of the base class more than one time and it results in ambiguity. Here, base class is called indirect base class and intermediate base classes are called direct base classes.

To avoid this ambiguity, the indirect base class is made virtual base class and to define base class as virtual, a keyword virtual is appended when extending the direct base classes from the indirect base class as given in the example. In this way, when we define the indirect base class virtual, compiler take care that only one copy of that class is inherited, regardless of how many inherited paths exist between the virtual base class and the derived class.

**Figure 9.1: Hybrid Inheritance**




*Example:*

```
class  student
{
   protected:
           char  course[20];
           char  name[20];
           int  enroll;
};

class  exam:virtual  public  student
{
     protected:
           char  etype[20];
           int  msub1;
           int  msub2;
           int  msub3;
};
```

**Notes**

```
class sports:virtual public student
{
     protected:
          char sname[20];
          int smarks;
};

class result:public exam, public sports
{
   private:
          int tmarks;
   public:
          void input();
          void total();
          void display();
};

void result::input()
{
       cout<<"enter enroll:";
       cin>>enroll;
       cout<<"enter name:";
       cin>>name;
       cout<<"enter course:";
       cin>>course;
       cout<<"enter exam type:";
       cin>>etype;
       cout<<"enter marks of subject1:";
       cin>>msub1;
       cout<<"enter marks of subject2:";
       cin>>msub2;
       cout<<"enter marks of subject3:";
       cin>>msub3;
       cout<<"enter sports name:";
       cin>>sname;
       cout<<"enter sports marks:";
       cin>>smarks;
```

Notes

```
}

void  result::total()
{
       tmarks=msub1+msub2+msub3+smarks;
}

void  result::display()
{
       cout<<"enroll:"<<enroll;
       cout<<"\nname:"<<name;
       cout<<"\ncourse:"<<course;
       cout<<"\nexam  type:"<<etype;
       cout<<"\ntotal  marks:"<<tmarks;
}

void  main()
{
    clrscr();
    result  r1;
     r1.input();
     r1.total();
     r1.display();
    getch();
}
```

### Self Assessment

Fill in the blanks:

11. Deriving a class from more than one direct base class is called …… inheritance.
12. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a…… .

## 9.4 Virtual Base Classes

We have just discussed a situation which would require the use of both multiple and multi level inheritance. Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved. Let us say the 'child' has two direct base classes 'parent1' and 'parent2' which themselves has a common base class 'grandparent'. The child inherits the traits of 'grandparent' via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as 'INDIRECT BASE CLASS'. Now, the

**Notes**

inheritance by the child might cause some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. So, there occurs a duplicacy which should be avoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class: for e.g.

```
                    class  g_parent
                    {
//Body
                    };
                    class  parent1:  virtual  public  g_parent
                    {
                           //  Body
                    };
                    class  parent2:  public  virtual  g_parent
                    {
                           //  Body
                    };
                    class  child  :  public  parent1,  public  parent2
                    {
                           //  body
                    };
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class.



*Notes* Note that keywords 'virtual' and 'public' can be used in either order.

//Program to show the virtual base class

```
#  include  <  iostream.h>
#  include  <  conio.h>
class  student                  //  Base  class  declaration
{
protected:
       int  r_no;
public:
       void  get_n  (int  a)
       {  r_no  =  a;}
       void  put_n  (void)
```

**Notes**

```
        { cout << "Roll No.: "<< r_no<<"In";}
};
class test : virtual public student          //Virtually
                                             declared common
{                                  //base class 1
protected:
        int para1;
        int para2;
public:
        void get_m (int x, int y)
        {       part1= x; part2=y;}
                void put.m (void)
        {
                cout <<"marks obtained: "<< "In";
                cout << "part1 = " << part1 <<"In";
                cout << "part2 = "<< part2 << "In";
        }
};
class sports: public virtual student     //virtually
                                            declared common
{                                        //base class 2
protected:
        int score;
public:
        void get_s (int a) {
                score = a;
        }
        void put_s (void)
        { cout << "sports wt.: "<<score<< "\n";}
};
class result: public test, public sports     //derived class
{
private : int total;
public:
        void show (void);
};
void result : : show (void)
{       total = part1 + part2 + score;
```

**Notes**

```
        put_n ( );
        put_m ( );
        put_s ( );   cout <<"\n total score= "<<total<<"\n";
}
main ( )
{
clrscr ( );
        result S1;
        S1.get_n (345)
        S1.get_m (30, 35);
        S1.get-S (7);
        S1.show ( );
```



*Did u know?* **How the problem of duplicate sub-objects is resolved?**

The problem of duplicate sub-objects is resolved with virtual inheritance. When a base class is inherited as virtual, only one sub-object will appear in the derived class—a process called virtual base-class inheritance.

//Program to show hybrid inheritance using virtual base classes

```
# include < iostream.h>
# include< conio.h>
Class A
{
protected:
        int x;
public:
        void get (int);
        void show (void);
};
void A :: get (int a)
        { x = a; }
void A :: show (void)
        { cout << X;}
Class A1 : Virtual Public A
{
protected:
        int y;
public:
```

**Notes**

```
        void get (int);
        void show (void);
};
void A1 :: get (int a)
        { y = a;]
void A1 :: show (void)
{
cout <<y;
{
class A2 : Virtual public A
{
protected:
        int z;
public:
        void get (int a)
        { z = a;}
        void show (void)
        { cout << z;]
};
class A12 : public A1, public A2
{
int r, t;
public:
        void get (int a)
        { r = a;}
        void show (void)
        { t = x + y + z + r;
                cout << "result ="<< t;
        }
};
main ( )
{
clrscr ( );
        A12 r;
        r.A : : get (3);
        r.A1: : get (4);
        r.A2 : : get (5);
```

Notes

```
                    r.get  (6);
                    r.show();
                }
```

### Self Assessment

Fill in the blanks:

13. ..... offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.
14. Each non-virtual object contains a copy of the ..... defined in the base class.

## 9.5 Overriding Member Function

If there are two functions with the same name in the base class and derived class and even the address of derived class is assigned to the pointer of base class; it executes the function of base class. But by declaring the base class function as virtual, pointer to base class executes the function of derived class. And in this way base class pointer behaves differently at different situation and applies the concept of polymorphism. This is also called run time or dynamic polymorphism because when a function 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.

```
class  base
{
    public:
          virtual  void  display();
          virtual  void  show();
};

void  base::display()
{
       cout<<"\nbase  class  display()  function";

}

void  base::show()
{
       cout<<"\nbase  class  show()  function";

}
```

**Notes**

```
class derived: public base
{
   public:
          void display();
          void show();
};

void derived::display()
{
       cout<<"\nderived class display() function";

}

void derived::show()
{
       cout<<"\nderived class show() function";
}

void main()
{
    clrscr();
    base *b;
    derived d;
    b=&d;
     b->display();
     b->show();
     getch();
}
```

We can define data member and member function with the same name in both base and derived class. When the function with same name exists in both class and derived class, the function in the derived class will get executed. This means, the derived class will get executed. This means, the derived class function overrides the base class function.



*Example:*

```
Class base A

{
```

**Notes**

```
Public:

Void getdata ()

{

_____

_____

}

};

Class derived B: public base A

{

Public:

Void getdata ()

{

______

______

}

};

Void main ()

{

Derived B obj;
```

**Notes**

```
Obj. getdata ();

getdata ();

}
```

When the statement obj. getdata (); get executed, the function getdata () of the derived class i.e. of derived B get executed. This means, the derived class function overrides the base class function.

The scope resolution (:: ) operator can be used to access base class function through an object of the derived class.



*Example:*

Derived B obj;

Obj. Base A:: getdata ();

The above statements specify that the getdata () of base A is to be called.

A base class function can be accessed from within the derived class by as follows also:

Class derived B: public base A

```
{

Public:

Void getdata ()

{

Base A:: getdata ();    // call getdata () of base A

}

};
```

Notes

## 9.6 Constructors under Inheritance

As we know, the constructors play an important role in initializing objects. We did not use them earlier in derived classes for the sake of simplicity. As long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory 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.

It 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 a multi-level 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.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class.



*Caution* The base constructors are called and executed before executing the statements in the body of the derived constructor.

Let us consider an example:

Derived (int a1, int a2, float b1, float b2, int d1):

```
        A(a1, a2), B (b1, b2)
{
        d = d1;
}
```

In the above constructor named derived, supplies the five arguments. The A(a1, a2) invokes the base constructor A( ) and B(b1, b2) invokes another base constructor B ( ). So the derived ( ) constructor has one argument d1 of its own the derived constructor contains two parts separated by a colon (:). The first part provides the declaration of the arguments that are passed to the derived constructor and the second part lists the function calls to the base constructors.

The constructor derived (.) may be invoked as follows:

……

derived obj (10, 20, 15.0, 17.5, 40);

……

The values are assigned to various parameters by the constructor derived ( ) as follows:

a1 ? 10

a2 ? 20

b1 ? 15.0

b2 ? 17.5

d1 ? 40

Notes

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 constructor is executed.

**Table 9.1: Execution of base Class Constructors**

| Methods of Inheritance | Order of execution |
|---|---|
| Class X: public y { }; | Y( ); base constructor<br>X ( ); derived constructor |
| Class X: public y, public Z<br>{ }; | Y( ); base constructor (first)<br>Z( ); base constructor (second)<br>X( ); derived constructor |
| Class X: public Y, virtual public Z<br>{ }; | Z( ); virtual base constructor<br>Y ( ); ordinary base constructor<br>X( ); derived constructor |

Let us consider a program using constructors:

```
# include <iostream.h>
class A
{
        int a1;
        public:
                A (int x)
        {
                a1 = x:
                cout << "constructor of A\n";
        }
        void display (void)
        {
        cout << "a1 = " << a1 << "\n";
        }
}
class B
{
        float b1;
        public:
                B( float y)
                {
                        b1 = y;
                        cout << " constructor of B\n";
                }
```

**Notes**

```
                void display 1 (void)
                {
                cout << "b1 = " <<b1 << "\n";
                }
}
Class C: public B, public A
{
        int p, q;
        public c:
        c (int a, float b, int c, int d): A(a), B(b)
        {
                p = c;
                q = d;
cout << "constructor of c \n";
}
void display 2 (void)
{
        count << "p = " << p << "\n";
        cout << "q = " << q << "\n";
}
}
main ( )
{
        c obj (10, 20.57, 40, 70);
        cout << "\n";
        obj . display ( );
        obj . display 1( );
        obj . display 2 ( );
}
```

The out of the above code would be as follows:

Constructor of B

Constructor of A

Constructor C

a1 = 10

b1 = 20.57

b = 40

q = 70

Notes

*Notes* Note that B is initialized/called first, although it appears second in the derived constructor. This is because it has been declared first in the derived class header line. Also note that A(a1) and B(b1) are function calls. Therefore the parameter should not include types.

## 9.7 Destructors under Inheritance

If a derived class doesn't explicitly call a constructor for a base class, the default constructor for the parent class. In fact, the constructors for the parent classes will be called from the ground up. For example, if you have a base class Vehicle and Car inherits from it, during the construction of a Car, it becomes a Vehicle and then becomes a Car.

Destructors for a base class are called automatically in the reverse order of constructors.

One thing to think about is with inheritance you may want to make the destructors virtual:

```
class Vehicle {
public:
    ~Vehicle();
};
```

This is important if you ever need to delete a derived class when all you have is a pointer to the base class. Say you have a "Vehicle *v" and you need to delete it. So you call:

```
delete v;
```

If Vehicle's destructor is non-virtual and this happens to actually by a Car *", the destructor will not call Car::~Car() - just Vehicle::~Vehicle()

### *The Destructor Execution Order*

1. As the objects go out of scope, they must have their destructors executed also, and since we didn't define any, the default destructors will be executed.
2. Once again, the destruction of the base class object named unicycle is no problem, its destructor is executed and the object is gone.
3. The sedan_car object however, must have two destructors executed to destroy each of its parts, the base class part and the derived class part. The destructors for this object are executed in reverse order from the order in which they were constructed.
4. In other words, the object is dismantled in the opposite order from the order in which it was assembled. The derived class destructor is executed first, then the base class destructor and the object is removed from the allocation.
5. Remember that every time an object is instantiated, every portion of it must have a constructor executed on it. Every object must also have a destructor executed on each of its parts when it is destroyed in order to properly dismantle the object and free up the allocation. Compile and run this program.

Notes

### Self Assessment

Fill in the blanks:

15. The destruction of the base class object named unicycle is no problem, its destructor is executed and the …….. is gone.

16. In a multi-level inheritance, the …… will be executed in the order of inheritance.

## 9.8 Making a Private Member Inheritable

The members of base class which are inherited by the derived class and their accessibility is determined by visibility modes. Visibility modes are:

1. ***Private:*** When a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.

2. ***Public:*** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

3. ***Protected:*** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited.

**Table 9.2: Execution of base Class Constructors**

| Base Class Visibility ↓ | Derived Class Visibility | | |
|---|---|---|---|
| | **Public** | **Private** | **Protected** |
| Private | X | X | X |
| Public | Public | Private | Protected |
| Protected | Protected | Private | Protected |

The private and protected members of a class can be accessed by:

1. A function i.e. friend of a class.

2. A member function of a class that is the friend of the class.

3. A member function of a derived class.

### Self Assessment

Fill in the blanks:

17. A member declared as …… is accessible by the member functions within its class and any class immediately derived from it.

Notes



*Caselet*

## Lisping IT, with ABC

The vocabulary of a generation reflects the times that they live in. A generation ago, a mouse, a monitor, an apple or windows stood for a rodent, a crocodile, a fruit or an opening in the wall.

Today's four-year-olds know it's much more than that.

And today's thirteen year-olds have already integrated computer science into their school curriculum.

In the last couple of years, almost all the players in the IT education field have announced tie-ups with schools across the country.

What kind of courses are they offering and how useful are these courses in the real world?

Says Asha Devi, Director, FutureSchools, a Chennai-based IT education institute, "Our courses are a blend of computer basics for the IT novice, which later move on to advanced concepts and software, so as to help students discover their strengths in the IT field and decide their career choices.''

In its programme for school children, FutureSchools offers integrated theme-based projects which focus on primary technology and ongoing technology. It has tied up with 33 schools all over the country.

Suresh Kumar, Director, IT Kids, another Chennai-based institute, feels a child should not experience any stress while learning technology and the courses that IT Kids offers aim toward that. "Our courses are specially designed to integrate information technology with academics and value education. Therefore, our courses are unique from others offered as adult curriculum.''

At IT Kids, which has tied up with 60 schools across the country, the children are taught both programming languages and application tools. In fact, last year, the institute initiated the 'School of Tomorrow' programme where apart from introducing the children to basic computer lessons, IT will be integrated into their curriculum.

The programme will provide class-appropriate content and application-led modules which are project-oriented. On the technology side, the programme deals with subjects such as operating environment, word processing, graphics, spread sheets, databases, multimedia and animation and programming languages.

The content is boosted with each class level and is integrated with the school syllabi.

On the academic side, the subjects covered include Physics, Chemistry, Mathematics, Social Sciences and languages. At each class level, the students will be trained to integrate computer skills learnt with lessons in other subjects through sample exercises.

FutureSchools believes in integrating its services in four areas of a child's development: courses for the teachers, students, parents and school managements.

And for the child there are four options: Technology Programmes, Academic Programmes, Enrichment Programmes and GATE (Gifted and Talented Child Education) Programme.

In addition to this, FutureSchools has tied up with Cambridge International Examinations which is a department of the Cambridge University to offer technology training and

*Contd...*

Notes

assessment between the ages 6-16. And finally they are awarded the Cambridge Starter Award Certification in technology skills and applications.

Most of these courses are meant to equip students with computer skills needed in the school environment. According to Asha Devi, "These courses are career-oriented in the sense they help the students decide their area of interest and develop their skills accordingly.''

Says Suresh Kumar, "Today, IT application is an inherent part of any professional course <147,1,0>and therefore, children will find that these have helped them prepare for careers within and outside the IT industry.''

Integration of academics and computer education is what most schools are looking at as this will help in the familiarisation process. At IT Kids, says Kumar, "there is a lot of integration of academics and value education, which is useful to all children alike, irrespective of their ultimate choice of profession.''

The learning packages and skill development packages vary according to the age. For instance, a Grade I student will be exposed to programming skills such as Logo while a Grade VIII or IX student will have to learn C and HTML.

And how do these programmes cover the school subjects? At IT Kids, at each class level, students are trained to integrate computer skills learnt with lessons in other subjects through sample exercises. A Grade V student will tackle prime and composite numbers in Maths, Scramble words in English, Continents in Geography and Time and Distance in Physical Sciences.

Asha Devi says at FutureSchools, the goal is to help each child think different and go beyond what has been achieved. "The child is centric and places the teacher as a facilitator in the learning environment. This lets the learning happen at the educable moment when the grasping and understanding peak in a child.''

But the traditional method of reaching to children through their teachers is well-understood by all these institutes. FutureSchools, in fact, conducted 'Futurecast' an event for teachers where technology infusion was the main focus.

The event provided teachers with a platform to infuse technology into classroom teaching. Teams of teachers from various departments interacted with one another to learn new methods of applying technology in the curricula.

FutureSchools also offers teachers Professional Development Programmes on teaching methodologies. IT Kids did a survey and found that 90 per cent of the school teachers were aware of the latest developments in IT and related areas.

Realising that teachers are the best resource people, Intel Asia has launched a programme for teachers. Called Intel@Teach to Future, this worldwide training initiative plans to cover 100,000 teachers by the end of 2002 and till date has covered more than 50,000 teachers.

It also has a programme for students to encourage them to enter technical careers. Called Intel@Innovation Education, the programme aims to help students improve science, math and engineering knowledge through the use of IT.

In this forceful era of IT revolution, it's obvious the urban children are well-initiated into the system. What happens to children in rural areas? Says Asha Devi, "We have tied up with several schools in rural areas and are providing computer education to them at a nominal fee, thereby accomplishing our mission to help schools become Future Schools,

Contd...

Notes

to help children become Futurekids, creating a worldwide learning community that has integrated the power of technology to facilitate teachers to improve the students' performance.''

But what is the awareness level among these students of the revolution that is taking place?

Says Suresh Kumar, "Contrary to popular belief, the awareness level is high among the rural children. The attitude of rural children is variably different from that of urban children, who are motivated by learning for grades and marks, whereas the rural children learn out of interest.''

He feels the biggest challenge for players such as IT Kids is to fine-tune the curriculum to interest the rural children.

"We are in the process of doing this,'' he states.

FutureSchools has also been counselling and helping students realise their career dreams.

"In fact, we have been doing career counselling in many schools for Standards 9 and 10. Here we have concentrated on Government and rural schools where the awareness among parents is less.''

Both schools assert that all their courses are reasonably priced. While IT Kids charges between ₹ 40 and ₹ 60 per student per month, FutureSchools courses are between ₹ 40 and ₹ 180 per month per student.

The rural and government schools are given a discount to encourage the IT movement in the country.

As Ramya Srinivas, a Bangalore school teacher, puts it: "These courses have become a part of our curriculum. The success of these courses depends on how well integrated they are into the mainstream education.''

## 9.9 Summary

- Inheritance is the capability of one class to inherit properties from another class.
- It supports reusability of code and is able to simulate the transitive nature of real life objects. Inheritance has many forms: Single inheritance, multiple inheritance, hierarchical inheritance, multilevel inheritance and hybrid inheritance.
- A subclass can derive itself publicly, privately or protectedly. The derived class constructor is responsible for invoking the base class constructor, the derived class can directly access only the public and protected members of the base class.
- When a class inherits from more than one base class, this is called multiple inheritance.
- A class may contain objects of another class inside it.
- his situation is called nesting of objects and in such a situation, the contained objects are constructed first before constructing the objects of the enclosing class.
- Single Inheritance: Where a class inherits from a single base class, it is known as single inheritance.
- Multilevel Inheritance: When the inheritance is such that the class. A serves as a base class for a derived class B which is turn serves as a base class for the derived class C. This type of inheritance is called 'Multilevel Inheritance.'

Notes

- Multiple Inheritance: A class inherit the attributes of two or more classes. This mechanism is known as Multiple Inheritance.'
- Hybrid Inheritance: The combination of one or more types of the inheritance.

## 9.10 Keywords

***Abstract Class:*** A class serving only a base class for other classes and no objects of which are created.

***Base class:*** A class from which another class inherits. (Also called super class)

***Containership:*** The relationship of two classes such that the objects of a class are enclosed within the other class.

***Derived class:*** A class inheriting properties from another class. (also called sub class)

***Inheritance:*** Capability of one class to inherit properties from another class.

***Inheritance Graph:*** The chain depicting relationship between a base class and derived class.

***Visibility Mode:*** The public, private or protected specifier that controls the visibility and availability of a member in a class.

## 9.11 Review Questions

1. Consider a situation where three kinds of inheritance are involved. Explain this situation with an example.
2. What is the difference between protected and private members?
3. Scrutinize the major use of multilevel inheritance.
4. Discuss a situation in which the private derivation will be more appropriate as compared to public derivation.
5. Write a C++ program to read and display information about employees and managers. Employee is a class that contains employee number, name, address and department. Manager class and a list of employees working under a manager.
6. Differentiate between public and private inheritances with suitable examples.
7. Explain how a sub-class may inherit from multiple classes.
8. What is the purpose of virtual base classes?
9. What will be the output of following code?

```
Class  one
{
        public:
        one()
{
        cout<<"Creating  class  ONE";
}
        };
```

Notes

```
Class two : one
{
        public:
                two()
{
        cout<<"Creating class TWO";
}
        };
        two myclass = new two();
```

10. How will you make a private member inheritable?

## Answers: Self Assessment

| | | | |
|---|---|---|---|
| 1. | reusing | 2. | object |
| 3. | derived class | 4. | protected |
| 5. | base | 6. | private |
| 7. | derived | 8. | declaration |
| 9. | base specifier | 10. | Hybrid |
| 11. | Multiple | 12. | Comma |
| 13. | Virtual base classes | 14. | data members |
| 15. | object | 16. | constructors |
| 17. | protected | | |

## 9.12 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://msdn.microsoft.com/en-us/library/wcz57btd(v=vs.80).aspx

http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/

Notes

# Unit 10: Virtual Functions and Polymorphism

**CONTENTS**

Objectives

Introduction

10.1 Virtual Functions

10.1.1 Late Binding

10.2 Pure Virtual Functions

10.3 Abstract Classes

10.4 Polymorphism

10.5 Summary

10.6 Keywords

10.7 Review Questions

10.8 Further Readings

## Objectives

After studying this unit, you will be able to:

- Demonstrate the virtual functions
- Recognize the pure virtual functions
- Describe the abstract classes
- Explain the polymorphism

## Introduction

By default, C++ matches a function call with the correct function definition at compile time. This is called static binding. You can specify that the compiler match a function call with the correct function definition at run time; this is called dynamic binding. You declare a function with the keyword virtual if you want the compiler to use dynamic binding for that specific function.

## 10.1 Virtual Functions

Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_defined_name{
private:
public:
virtual return_type function_name1 (arguments);
virtual return_type function_name2(arguments);
```

Notes

```
virtual return_type function_name3( arguments);
_______
_______
};
```

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 precedes the 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.



*Example:* The following declaration of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that 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.



*Example:*

```
class point {
intx;
inty;
public:
virtual void display ();
};
virtual void point: : display () //error
{
Function Body
}
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point {
int x;
int y;
public:
virtual static int length (); //error
```

**Notes**

```
};
int point: : length ( )
{
Function body
{
```

A virtual function cannot have a constructor member function but it can have the destructor member function.

```
class point {
int x;
int y;
public:
virtual point (int xx, int yy); // constructors, error
void display ( );
int length ( );
} ;
```

A destructor member function does not take any argument and no return type can be specified for it not even void.

```
class point {
int x;
int y;
public:
virtual _ point (int xx, int yy); //invalid
void display ( );
int length ( );
```

It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual Method in the base class.

```
class base {
int x,y;
public:
virtual int sum (int xx, int yy ); //error
} ;
class derived: public base {
intz;
public:
virtual float sum (int xx, int yy);
};
```

The above declarations of two virtual functions are invalid. Even though these functions take identical arguments note that the return data types are different.

Notes

virtual int sum (int xx, int IT); //base class

virtual float sum (int xx, int IT); //derived class

Both the above functions can be written with int data types in the base class as well as in the derived class as

virtual int sum (int xx, int yy); //base class

virtual int sum (int xx, int yy); //derived class

Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declared virtual.

```
virtual void display () //error, nonmember function
{
Function body
}
```

### 10.1.1 Late Binding

Late binding means selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility. The late binding is implemented through virtual functions as a result we have to declare an object of a class either as a pointer to a class or a reference to a class.



*Did u know?* **What is static binding?**

By default, C++ matches a function call with the correct function definition at compile time. This is called static binding.

For example the following shows how a late binding or run time binding can be carried out with the help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( );
int sum ( );
};
class derivedD : public baseA
{
private:
int x;
float y;
public:
```

**Notes**

```
void display (); //virtual
int sum ( );
};
void main ( )
{
baseA *ptr;
derivedD objd;
ptr = &objd;
Other Program statements
ptr->display (); //run time binding
ptr->sum ( ); //compile time binding
}
```

Note that the keyword virtual is be followed by the return type of a member function if a run time is to be bound. Otherwise, the compile time binding will be effected as usual. In the above program segment, only the display ( ) function has been declared as virtual in the base class, whereas the sum () is non-virtual. Even though the message is given from the pointer of the base class to the objects of the derived class, it will not access the sum ( ) function of the derived class as it has been declared as non-virtual. The sum ( ) function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic binding.

```
#include <iostream.h>
#include <conio.h>
class baseA {
public:
virtual void display () {
cout<< "One \n";
}
};
class derivedB : public baseA
{
       public:
       virtual void display 0 {
       cout<< "Two\n"; }
};
class derivedC: public derivedB
{
       public:
       virtual void display ( ) {
```

Notes

```
        cout<< "Three \n"; }

};
void main ( ) {
        //define three objects
        baseA obja;
        derivedB objb;
        derivedC objc;
        base A *ptr [3]; //define an array of pointers to baseA
        ptr [0] = &obja;
        ptr [1] = &objb;
        ptr [2] = &objc;
        for (int i = 0; i <=2; i ++ )
        ptr [i]->display (); //same message for all objects
        getche ( );
}
```

Output

One

Two

Three

The program listed below illustrates the static binding of the member functions of a class. In program there are two classes student and academic. The class academic is derived from class student. The two member function getdata and display are defined for both the classes. *obj is defined for class student, the address of which is stored in the object of the class academic.



*Notes* The functions getdata ( ) and display ( ) of student class are invoked by the pointer to the class.

```
# include <iostream.h>
#include <conio.h>
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );
};
```

**Notes**

```
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( );
} ;
void student:: getdata ( )
{
       cout<< "enterrollno\n";
       cin>> rollno;
       cout <<"enter     name \n";
       cin > > name;
}
void student:: display ( )
{
       cout<< "the student's roll number is "<<rollno<<"and name is"
<<name;
       cout<< endl;
}
void academic :: getdata ()
{
       cout<< "enter stream of a student? \n";
       cin >>stream;
}
void academic :: display () {
       cout<< "students stream \n";
       cout <<stream<< end!;
}
void main ( )
{
       student *ptr;
       academic obj;
       ptr=&obj;
       ptr->getdata ();
       ptr->display ();
       getche ( );
}
```

Notes

output

enter rollno

25

enter name

raghu

the student's roll number is 25 and name is raghu



*Did u know?* **What is dynamic binding?**

You can specify the compiler match a function call with the correct function definition at run time; this is called dynamic binding.

The program listed below illustrates the dynamic binding of member functions of a class. In this program there are two classes student and academic. The class academic is derived from student. Student function has two virtual functions getdata ( ) and display (). The pointer for student class is defined and object for academic class is created. The pointer is assigned the address of the object and function of derived class are invoked by pointer to student.

```
#include <iostream.h>
#include <conio.h>
class student {
private:
introllno;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class academic: public student {
private :
char stream[10];
public:
void getdata ( );
void display ( );
};
void s_dent:: getdata ( )
{
        cout<< "enter rollno\n";
        cin >> rollno;
        cout<< "enter name \n";
        cin >>name;
```

**Notes**

```
}
void student:: display ()
{
       cout<< "the student's roll number is "<<rollno<<"and name is"
<<name;
       cout<< end1;
}
void academic:: getdata ( )
{
       cout << "enter stream of a student? \n";
       cin>> stream;
}
void academic:: display 0
{
       cout<< "students stream \n";
       cout<< stream << end1;
}
void main ()
{

       student *ptr;
       academic obj;
       ptr = &obj;
       ptr->getdata ( );
       ptr->dlsplay ( );
       getche ( )
}
```

output

enter stream of a student?

Btech

students stream

Btech

## Self Assessment

Fill in the blanks:

1. The return type of an …….. virtual function may differ from the return type of the overridden virtual function.

Notes

2. A virtual function is a member function of a base class and relies on a specific object to determine which implementation of the …… is called.
3. If a function is declared …… in its base class, you can still access it directly using the scope resolution (::) operator.
4. If you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the……....
5. If you declare a base class destructor as virtual, a derived class destructor will …… that base class destructor, even though destructors are not inherited.

## 10.2 Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task.

The following program demonstrates how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class. In the example there are two classes employee and grade. The class employee is base class and the grade is derived class. The functions getdata ( ) and display ( ) are declared for both the classes. For the class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class.



*Caution* The methods of the derived class are invoked by the pointer to the base class.

```
#include <iostream.h>
#include<conio.h>
class employee {

int code;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class grade: public employee
{
        char grd [90];
        float salary;
public:
        void getdata ( );
        void display ( );
};
void employee :: getdata ()
```

**Notes**

```
{
}
void employee:: display ( )
{
}
void grade:: getdata ()
{
        cout<<" enter employee's grade ";
        cin> > grd;
        cout<< "\n enter the salary ";
        cin>> salary;
}
void grade:: display ( )
{
        cout<<" Grade salary \n";
        cout<< grd<<" "<< salary<< end1;
}
void main ( )
{
        employee *ptr;
        grade obj;
        ptr = &obj;
        ptr->getdata ( );
        ptr->display ( );
        getche();
}
```

Output

enter employee's grade A

enter the salary 250000

Grade salary

A 250000

Another format can have a pure virtual function when a virtual function is declared within the class declaration itself. The virtual function may be equated to zero if it does not have a function definition. Such functions are also called as do-nothing functions.



*Task* A Pure Virtual Function is a Virtual function with no body. Explain with an example.

Notes

```
//pure virtual functions '
#include <iostream.h>
class base {
private :
int x, y;
pubic:
virtual void getdata ( ) =0;
virtual void display ( )=0;
};
class derivedB : public base {
Data Members
Member Functions
};
```

### Self Assessment

Fill in the blanks:

6. Since pure virtual function has no body, the programmer must add the notation ……. for declaration of the pure virtual function in the base class.
7. A pure virtual function simply acts as a placeholder that is meant to be redefined by …….. classes.
8. A pure virtual function is declared, but not necessarily defined, by a …….. class.

## 10.3 Abstract Classes

An abstract class is a class containing a pure virtual function. They cannot be instantiated into an object directly. Abstract classes are designed to be specifically used as base classes. An abstract class contains at least one pure virtual function. Only a subclass of an abstract class can be instantiated directly if all inherited pure virtual methods have been implemented by that class or a parent class.

Abstract classes provide a mechanism to implement multiple inheritance. In other OOP languages the concept of interface does the same thing. Pure virtual functions are also used where the method declarations are being used to define an interface for which derived classes will supply all implementations.

An interface can be constructed with an abstract class having only pure virtual functions, and no data members or ordinary methods. Use of purely abstract classes as interfaces works in C++ as it supports multiple inheritance.



*Notes* Many OOP languages including Java do not support multiple inheritance and therefore they provide a separate interface mechanism.

**Notes**

An additional feature of C++ abstract class is that it allows pure virtual methods in an abstract class to contain an implementation in their declaring class. This mechanism providing fallback or default behavior that a derived class can fall upon if needed.

The following code snippet defines an abstract class:

```
class  Abst

{

public:

              virtual  void  AbstFunc()  =  0;

};
```

Function Abst::AbstFunc is a pure virtual function. Note that a function declaration cannot have both a pure specifier and a definition. Due to this reason the following code is incorrect. The compiler will issue an error message for this.

```
Class  Abst

{

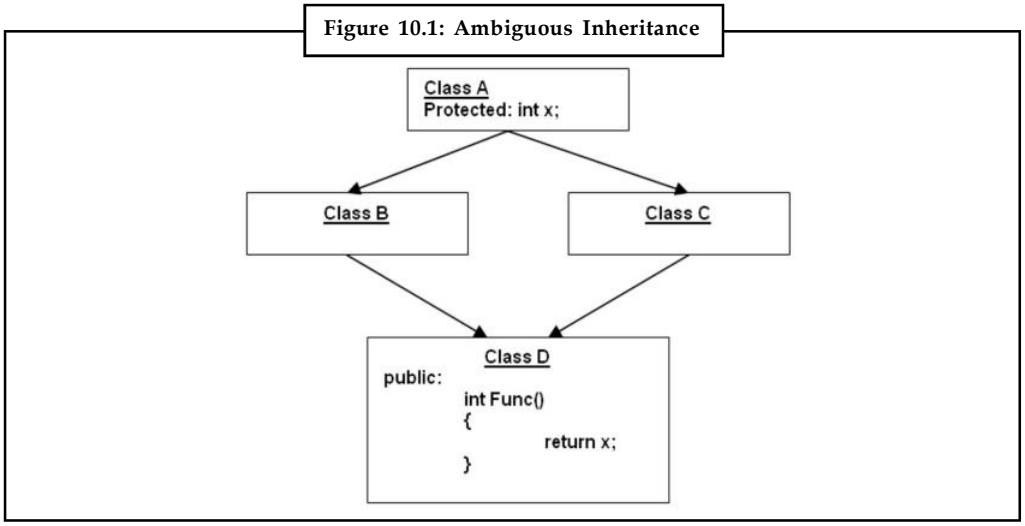
       virtual  void  AbstFunc()  {  }  =  0;   //Incorrect!  {}  and  =0  cannot  be
used  together

};
```

There are other restrictions also. You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class.

Care should be taken when cyclic inheritance is employed as shown in Figure 10.1.

**Figure 10.1: Ambiguous Inheritance**

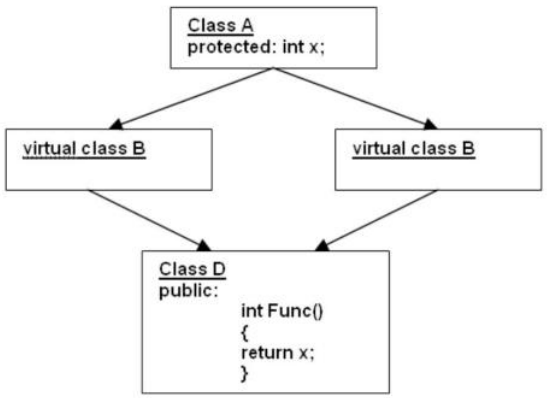



In this case a compiler error takes place. The member function Func() of class D attempts to access member data x of class A. The error results because the derived classes B and C derived from base class A create copies of A called. Each of the copies have A member data and member functions and each has one copy of member data x. When the member function of the class D tries to access member data x, confusion arises as to which of the two copies it must access since it has been derived from both derived classes.

Notes

To avoid this situation virtual or abstract base class is used. Both of the derived classes B and C are created as virtual base classes, meaning they should share a common copy of member in their base class. The modified definition is shown in Figure 10.2.

**Figure 10.2: Use of Virtual Base Classes to Disambiguation of Inheritance**




## Self Assessment

Fill in the blanks:

9. A base class containing one or more pure virtual member functions is called an …… class.
10. An abstract class contains at least one…….. .
11. A class derived from an abstract base class will also be abstract unless you …….. each pure virtual function in the derived class.
12. Only a subclass of an abstract class can be instantiated directly if all ……. pure virtual methods have been implemented by that class or a parent class.

## 10.4 Polymorphism

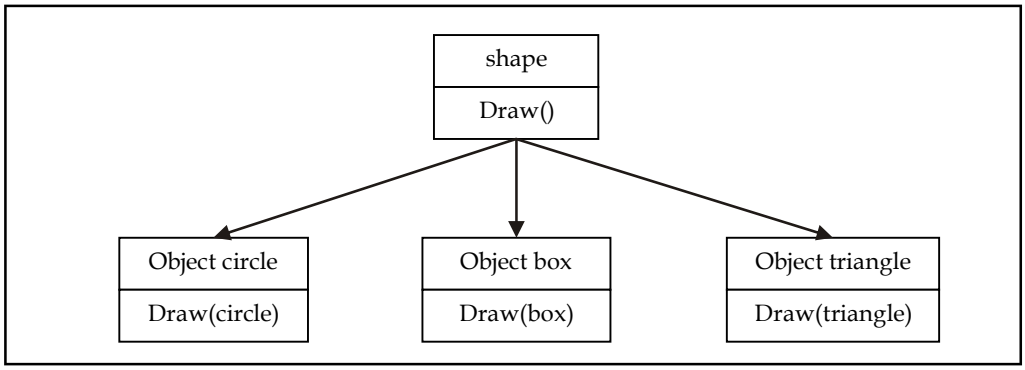
Polymorphism is an important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For tow numbers, the operation will generate a sum. If the operands are strings, then the operation will produce a third string by contention. The diagram given below, illustrates that a single function name can be used to handle different number and types of arguments. This is something similar to a particular word having several different meanings depending on the context.

*Task* Polymorphism means that some code or operations or objects behave differently in different contexts. In a group of four analyze the meaning of this statement with an example.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be

**Notes**

accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance as shown below.




Polymorphism can be implemented using operator and function overloading, where the same operator and function works differently on different arguments producing different results. These polymorphisms are brought into effect at compile time itself, hence is known as early binding, static binding, static linking or compile time polymorphism.

However, ambiguity creeps in when the base class and the derived class both have a function with same name. For instance, let us consider the following code snippet.

```
Class  aa
{
        Int  x;
        Public:
                Void  display()  {……}   //display  in  base  class
};
Class  bb  :  public  aa
{
        Int  y;
        Public:
                Void  display()  {……}   //display  in  derived  class
};
```

Since, both the functions aa.display() and bb.display() are same but at in different classes, there is no overloading, and hence early binding does not apply. The appropriate function is chosen at the run time – run time polymorphism.

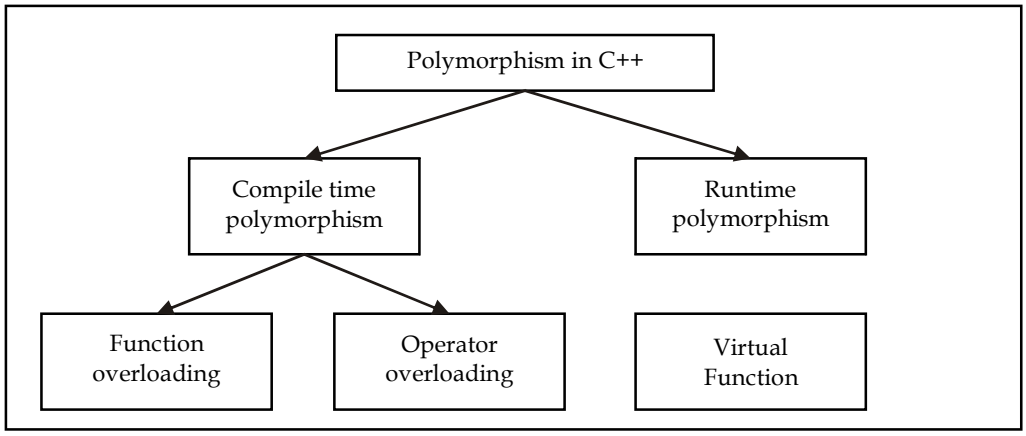
C++ supports run-time polymorphism by a mechanism called virtual function. It exhibits late binding or dynamic linking.

As stated earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. Therefore, an essential feature of polymorphism is the ability to refer to objects without any regard to their classes. It implies that a single pointer variable may refer to object of different classes.

However, a base pointer, even if is made to contain the address of the derived class, always executes the function in the base class. The compiler ignores the content of the pointer and

Notes

chooses the member function that matches the type of the pointer. Thus, the polymorphism stated above cannot be implemented by this mechanism.






*Did u know?* **What is the disadvantage of polymorphism?**

The biggest disadvantage of polymorphism is creation of reusable codes by programmers. Classes once written, tested and implemented can be easily reused without caring about what's written in the cases.

C++ implements the runtime object polymorphism using a function type known as virtual function. When a function with the same name is used both in the base class and the derived class, the function in the base class is declared virtual by attaching the keyword virtual in the base class preceding its normal declaration. Then 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, one can execute different definitions of the virtual function as given in the program below.

```
#include <iostream.h>
class base
{
       public:
              void display()
              {
                     cout<<"\n print base";
              }
              virtual void show()         //virtual function
              {
                     cout<<"\n show base";
              }
};
class derived : public base
{
```

**Notes**

```
public:
        void display()
        {
                cout<<"\n display derived";
        }
                void show()
                {
                        cout<<"\n show derived";
                }
};
main()
{
        base bb;
        derived dd;
        base *baseptr;
        cout <<"\nbaseptr points to the base \n";
        baseptr = &bb;
        baseptr -> display();       //calls base function display()
        baseptr -> show();  //calls base function show()
        cout <<"\n\nbaseptr points to the derived \n";
        baseptr = &dd;
        baseptr -> display();       //calls derived function display()
        baseptr -> show();  //calls derived function show()
}
```

The output of this program would be:

Baseptr points to base

Display base

Show base

Baseptr points to derived

Display derived

Show derived

Here, we see that the same object pointer points to two different objects of different classes and yet selects the right function to execute. This is implementation of function polymorphism. Remember, however, that runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It is also interesting to note that since, all the C++ classes are derived from the Object class, a pointer to the Object class can point to any object of any class in C++.

## Self Assessment

Notes

Fill in the blanks:

13. A class that declares or inherits a virtual function is called a …….. class.

14. Polymorphism plays an important role in allowing …….. having different internal structures to share the same external interface.

15. An essential feature of polymorphism is the ability to refer to objects without any regard to their…….. .



*Caselet*

### Betting on the Internet, PPP AIMS High

Betting on the Internet to level the playing field, a small Chennai-based firm is setting its sights high. The company, Plans Proposals & Projects (PPP), run by three brothers (who also constitute its entire staff), is looking to appoint distributors worldwide for a software product it has developed using the Java programming language. The software, PPPshar, enables several PC users to simultaneously access the Internet using a single dial-up connection.

According to Mr. Parameshwar Babu, the product's chief developer and the technical expert in the trio, PPPshar could be used to hook up all the PCs (running Windows 95 and a standard Web browser software) in a corporate network to the Internet using a single-modem and a single phone connection. The software, priced at ₹ 5,000 per terminal (on the user network), needs to be loaded only on the machine connected to the modem. It can also be customised based on the user's specific requirements.

PPPshar can be used with a dial-up TCP/IP Internet connection and there is no need for a permanent Internet Protocol (IP) address as in the case of leased line connections. Provided the product performs as claimed, the savings derivable from the software are quite obvious for any organisation which wants to provide multiple Net connections to its members: VSNL, currently the sole Internet Service Provider (ISP) in India, charges about ₹ 10 lakhs annually for a high speed (64 kbps kilo bits per second) leased line connection compared to ₹ 15,000 for a dial-up TCP/IP account (offering a maximum speed of 28.8 kbps).

Mr. Babu says that PPPshar would ensure that the speeds of the individual terminals in the network do not suffer when multiple users are logged on to the Net. According to him, the software keeps track of the "idle time" commonly encountered while surfing the Web (that is, while locating and loading Web pages) and divides this time among the different terminals.

"In any case, the individual speeds available to seven terminals simultaneously connected through a 28.8 kbps modem compares favourably with 16 users splitting up a 64 kbps leased line between them," he says.

According to Mr. Babu, PPPshar can also be used by system managers to selectively control access to certain types of information on the Net. For example, for a Chennai-based medical college-cum-hospital, PPP has customised the software to ensure that Web sites relevant to medicine can be accessed by the individual users.

*Contd...*

Notes

The company is targeting the product at corporates and educational institutions. It has already booked orders to install PPPshar, which was released on April 14, at five Chennai-based firms, including Grundig Electronics (India), the Indian subsidiary of the German consumer electronics giant.

The brothers point out that the product is also ideal for entrepreneurs wanting to set up "cybercafes" (in which visitors can access the Internet by paying a per hour fee) at a low capital investment. "There are not too many companies in the country who can afford to set up cybercafes using leased lines as the costs are too high. With PPPshar even small companies with a few PCs can afford to get into this business," points out Mr. Babu.

In fact, one of the first customers for PPPshar is the Chennai- based Quality Business Management (QBM), a private company providing electronic desktop publishing, Public Call Office and E-mail services at a prominent business location in the city. Using PPPshar, QBM has set up a 'Cyber Circle' in the same facility by hooking up five PCs to the Net to provide Internet-related information services.

A dozen Chennai-based firms and an equal number from the other metros have approached PPP for signing up as distributors for the product. According to Mr. Babu, five companies from other countries have also evinced interest based on the information they had obtained from the company's Web site (www.pppindia.com).

According to the brothers, Grundig (India) officials are so impressed with the product that they had offered to help the company market it in Germany and other parts of the world. "In fact, one of the directors in the parent company was so impressed with the product that he said we should set up shop in Germany rather than continuing here," says Mr. B.Shrinivas, the eldest among the three.

As for competition from similar products from overseas companies, Mr. Babu says he is aware of just one – 'WebShare' from Canada-based Protec Microsystems Inc. However, he points out that Web Share allows simultaneous shared Internet access for only three PCs.

The company is currently working on another Java-based software, PPPftp, which is a File Transfer Protocol (FTP) software for uploading and downloading information from the Net. According to Mr. Babu, an important feature of this software, not available to other FTP client software, is the ability to resume interrupted file transfers. "This enables users to download large software from the Internet. Even if the line gets disconnected in between, PPPftp will resume from where it left off during the last at tempt," he says.

Apart from product development, PPP also designs and hosts Web pages for clients on its Web site. The company's Web site at tracts visitors by posting several free resources – a trade bulletin board, a matrimonial service, a "herbal petrol" forum (which discusses the Ramar Pillai episode), a law forum for NRIs investing in India and a Web forum for Internet users in Chennai.

## 10.5 Summary

- Virtual functions do not really exist but it appears real in some parts of a program. 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.
- Early binding refers to the events that occur at compile time while late binding means selecting function during the execution. The late binding is implemented through virtual function.

Notes

- An abstract base class will not be used to create object, but exist only to act as a base class of other classes.
- Polymorphism allows the program to use the exactly same function name with exactly same arguments in both a base class and its subclasses. It refers to the implicit ability of a function to have different meanings in different contexts.

## 10.6 Keywords

***Abstract Base Class:*** An abstract base class will not be used to create object, but exist only to act as a base class of other classes.

***Late Binding:*** Selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility.

***Polymorphism:*** Polymorphism allows the program to use the exactly same function name with exactly same arguments in both a base class and its subclasses.

***Virtual Function:*** Virtual functions, one of advanced features of OOP is one that does not really exist but it appears real in some parts of a program.

## 10.7 Review Questions

1. Make the distinction between virtual functions and virtual base class?
2. How can C++ achieve dynamic binding yet also static typing?
3. What is the difference between virtual and non-virtual member functions?
4. What happens in the hardware when we call a virtual function? How many layers of indirection are there? How much overhead is there?
5. How can a member function in my derived class call the same function from its base class?
6. Write a program which uses a polymorphism with pointers.
7. Are virtual functions hierarchical? Give an example in support of your answer.
8. Write a program to use constructors and destructors in inheritance.
9. Debug the following program:

```
class one
{
     int a;
public:
     void set(int a)
     {
            a = a;
}
     void show()
     {
     cout << a;
```

Notes

```
        }
};
main()
{
     one O1, O2;
     O1.set(10);
     O2.show();
}
```

10. What is the purpose of a pure virtual function? Illustrate with a suitable example.

## Self Assessment

1. overriding
2. function
3. virtual
4. base class
5. override
6. =0
7. Derived
8. Base
9. Abstract
10. pure virtual function
11. override
12. inherited
13. polymorphic
14. objects
15. classes

## 10.8 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

Robert Lafore; *Object-oriented Programming in Turbo C++*; Galgotia.



*Online links*

http://www.artima.com/cppsource/pure_virtual.html

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101 index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr142.htm

# Unit 11: Pointers and Dynamic Memory Management

Notes

CONTENTS

Objectives
Introduction
11.1 Understanding Pointers
11.1.1 Pointer Variables
11.2 Declaring Pointers
11.2.1 Accessing Address of Variable
11.2.2 Pointer Expressions
11.2.3 Pointer Initialization
11.3 Pointer to Pointers
11.4 Pointer and Functions
11.4.1 Pointers to Functions
11.4.2 Function Returning Pointers
11.5 Dynamic Memory Management
11.5.1 New and Delete Operator
11.5.2 The this Pointer
11.6 Summary
11.7 Keywords
11.8 Review Questions
11.9 Further Readings

## Objectives

After studying this unit, you will be able to:

- Identify the understanding pointers
- Demonstrate the accessing address of a variable
- Recognize the declaring and initializing pointers
- Describe the pointer to pointer
- Explain the pointer to a function
- Discuss the dynamic memory management

## Introduction

Computers use their memory for storing instructions of the programs and the values of the variables. Memory is a sequential collection of storage cells. Each cell has an address associated with it. Whenever we declare a variable, the system allocates, somewhere in the memory, a memory location and a unique address is assigned to this location.

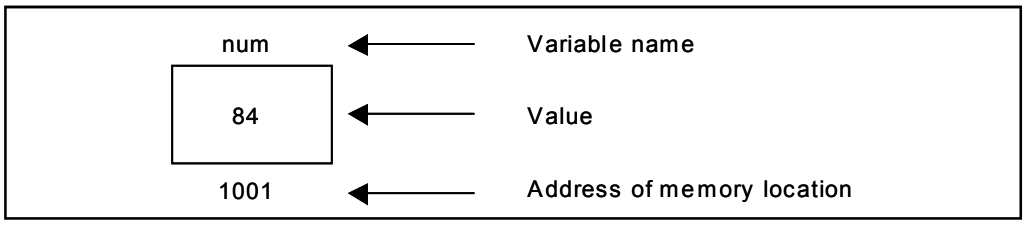
Notes

## 11.1 Understanding Pointers

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement:

```
int num = 84;
```

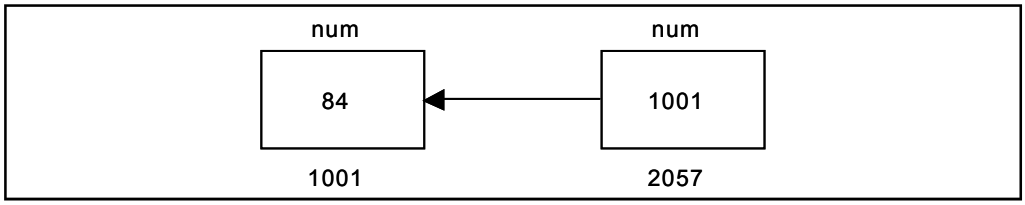
This statement instructs the system to reserve a 2-byte memory location and puts the value 84 in that location. Assume that a system allocates memory location 1001 for num. Diagrammatically, it can be shown as:




As the memory addresses are numbers, they can be assigned to some other variable.

Let ptr be the variable which holds the address of variable num.

Thus, we can access the value of num by the variable ptr. Thus, we can say "ptr points to num". Diagrammatically, it can be shown as






*Did u know?* **How does pointers helps us?**

Pointers help in allocating memory dynamically. Pointers improve execution time and saves space.

### 11.1.1 Pointer Variables

Pointers are variables that follow all the usual naming rules of regular, non-pointer variables. As with regular variables, you must declare pointer variables before you use them. A type of pointer exists for every data type in C++; you can use integer pointers, characters pointers, floating-point pointers, floating-point pointer, and so on. You can declare global pointer or local pointer, depending on where you declare them.

The only difference between pointer variables and regular variables is what they hold. Pointer do not contain values, but the address of a value.

C++ has two operators:

1. & - The "address of" operator
2. * - The de-referencing operator

Whenever you see '&' used with pointer, think of the phrase "address of". The '&' operator provides the memory address of whatever variable it precedes. The * operator, when used with

Notes

pointer, de-references the pointer's value. When you see * operator used, think of the phrase "value pointed to".

*Caution* The * operator retrieves the value that is "pointer to" by variable it proceeds.

### Self Assessment

Fill in the blanks:

1. A pointer is a variable that is used to store a …….. address.
2. There are two important pointer operators such as '*' and …….. .
3. The '*' operator is called the ……. operator.
4. Every variable is located under unique location within a computer's memory and this unique location has its own …… address, the memory address.

## 11.2 Declaring Pointers

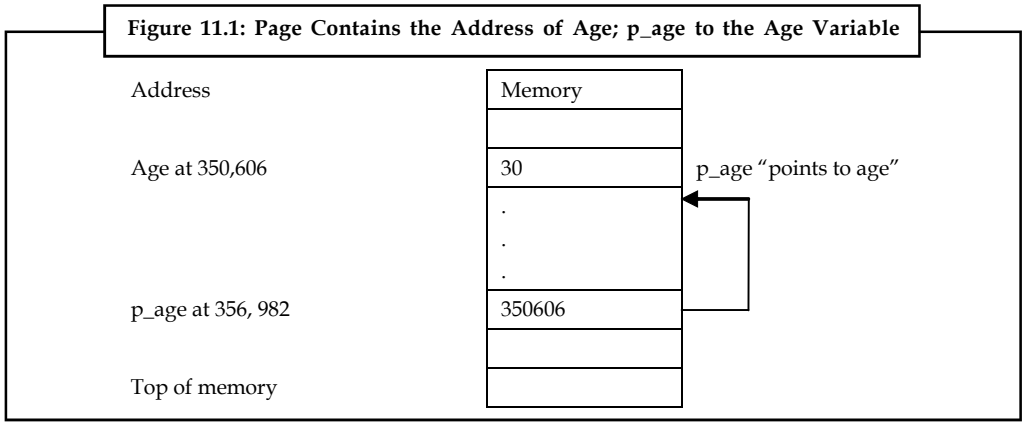
A pointer is a special variable that return the address of a memory. If you need to declare a variable to hold your age, you might use the following variable declaration:

int age = 18 // declares to variable to hold my age

Declaring age this way does several things. Because C++ knows that you need a variable called age, C++ reserves storage for that variable. C++ also knows that you will store only integers in age, not floating-point or double floating point data. You also have requested that C++ store the value of 18 in age after reserving storage for that variable.

Suppose that you want to declare a pointer variable, not to hold your age but to point to age, the variable that hold your age. p_age might be a good name for this pointer variable. Figure given below shows an illustration of what you want to do. This example assumes that C++ store age at address 1002, although your C++ compiler arbitrarily determines the address of age, and it could be anything.

**Figure 11.1: Page Contains the Address of Age; p_age to the Age Variable**




To declare the p_age pointer variable. You should do the following:

int*_age; // declare an integer pointer

As with the declaration for age, this line reserves a variable called p_age. It is not a normal integer variable, however, because of the C++ knows that this variable is to be a pointer variable.

Notes

The following program example demonstrates the declaration and use of pointer in C++.



*Example:*

```
# include<iostream.h >
              int main( )
              {
                     int i = 18, * p_age;
                     p_age   = &I;
                     cout<<i<< " << * p_age << endl;
              }
```



*Did u know?* **Why Member function pointers are important?**

Member function pointers are important because they provide an efficient way to cache the outcome of a decision over which member function to call.

### 11.2.1 Accessing Address of Variable

Consider the following statements:

int q, * i, n;

q = 35;

i = & q;

n = * i;

i is a pointer to an integer containing the address of q. In the fourth statement we have assigned the value at address contained in i to another variable n. Thus, indirectly we have accessed the variable q through n using pointer variable i.

### 11.2.2 Pointer Expressions

Like other variables, pointer variables can be used in expressions. Arithmetic and comparison operations can be performed on the pointers.



*Example:* If p1 and p2 are properly declared and initialized pointers, then following statements are valid.

y = * p1 * *p2;

sum = sum + * p1;

### 11.2.3 Pointer Initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:

Notes

```
int  number;

int  *tommy  =  &number;
```

The behavior of this code is equivalent to:

```
int  number;

int  *tommy;

tommy  =  &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (tommy), never the value being pointed (*tommy). You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

int number;

int *tommy;

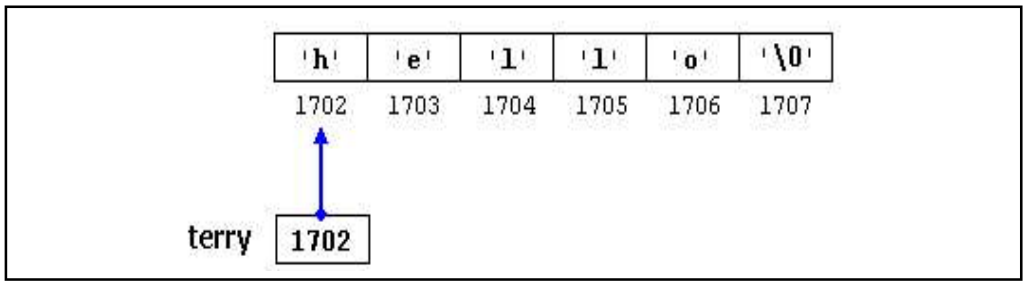
*tommy = &number;

that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

char * terry = “hello”;

In this case, memory space is reserved to contain “hello” and then a pointer to the first character of this memory block is assigned to terry. If we imagine that “hello” is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:




It is important to indicate that terry contains the value 1702, and not ‘h’ nor “hello”, although 1702 indeed is the address of both of these.

The pointer terry points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer).

*Example:*

We can access the fifth element of the array with any of these two expression:

*(terry+4)

terry[4]

Both expressions have a value of ‘o’ (the fifth element of the array).

Notes

### Self Assessment

Fill in the blanks:

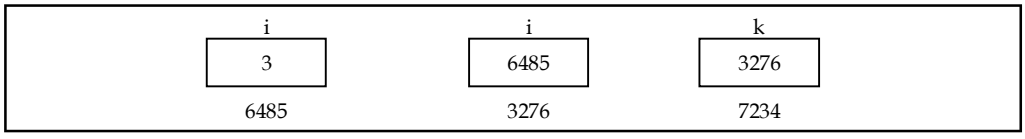
5. C++ also knows that you will store only …… in age, not floating-point or double floating point data.
6. Arithmetic and …….. operations can be performed on the pointers.
7. Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its ……. which data type a pointer is going to point to.

## 11.3 Pointer to Pointers

Pointer is a variable, which contains address of a variable. This variable itself could be another pointer. Thus, a pointer contains another pointer's address as shown in the example given below:

```
void main()
{
        int i = 3; int *j, **k;
        j = &i; k = &j;
        cout<<"Address of i = \n"<< &i;
        cout<<"Address of i = \n"<<j;
        cout<<"Address of i = \n"<<*k;
        cout<<"Address of j = \n"<<&j;
        cout<<"Address of j = \n"<<*k;
        cout<<"Address of k = \n\n"<<&k;
        cout<<"Value of j = \n"<<j;
        cout<<"Value of k = \n"<<k;
        cout<<"Value of i = \n"<<i;
        cout<<"Value of i = \n"<<*(&i);
        cout<<"Value of i = \n"<<*j;
        cout<<"Value of i = \n"<<**k;
}
```

The following figure would help you in tracing out how a program prints the output.




Output:

Address of i = 6485

Address of i = 6485

Address of i = 6485

Address of j = 3276

Notes

| | |
|---|---|
| Address of j | = 3276 |
| Address of k | = 7234 |
| Value of j | = 6485 |
| Value of k | = 3276 |
| Value of i | = 3 |
| Value of i | = 3 |
| Value of i | = 3 |
| Value of i | = 3 |

*Task* Explain the advantages of pointers to pointers.

## 11.4 Pointer and Functions

A function is a user defined operation that can be invoked by applying the call operator ("( )") to the function's name. If the function expects to receive arguments, these arguments (actual arguments) are placed inside the call operator. The arguments are separated by commas. A functions may be involved in one of the two way;

1. Call by value
2. Call by reference

### *Call by Value*

In this method, the value of each actual argument in the calling function is copied on to the corresponding formal arguments of the called function. The called function works with and manipulates its own copy of arguments and because of this, changes made to the formal arguments in the called function have no effect on the values of the actual arguments in the calling function.

### *Call by Reference*

This method can be used in two ways:

1. By passing the reference
2. By passing the pointer.

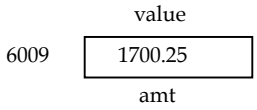
A reference is an alias name for a variable.

For instance, the following code

```
float value = 1700.25;

float &amt = value;   //amt reference of value
```

declares amt as an alias name for the variable value. No separate memory is allocated for amt rather the variable value can now be accessed by two names value and amt.



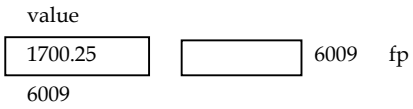
**Notes**

A pointer to a variable holds its memory address using which the memory area storing the data value of the variable can be directly accessed. For instance, the following code

float value = 1700.25

float fp = &value ; // fp stores the address of value

makes a float pointer *fp point to a float variable value. The variable value can now be accessed through fp also (using *fp)




The call by reference method is useful in situations where the values of the original variables are to be changed using a function. The following example program explains it:

*Example:* Program to swap values of two variables using pass by reference method.

```
#include<iostream.h>
#include<conio.h>          //for clrscr)
int main( )
{ clrscr( );
void swap(int &, int &); // prototype
int a = 7,  b = 4;
cout<<"Original values \n";
cout<<"a=" << a <<, "b=" << b << "\n";

swap(a,b);                 //invoke the function
cout<<"swapped values \n";
cont<<"a=" << a << "b=" << b << "\n";
return 0;
}
\\function definition
void swap(int & x, int & y)
{ int temp;
       temp = x;
       x=y;
       y=temp;
}
```

The output produced by the above program is as follows:

Original Values

a= 7, b = 4

Notes

Swapped Values

a = 4, b = 7

In the above example, the function swap ( )

creates reference 'x' for the first incoming integers and reference 'y' for the second incoming integer. Thus the original values are worked with but by using the names x and y. The function call is the simple one i.e.;

swap (a,b);

but the function declaration and definition include the reference symbol &. The function declaration and definition, both, start as

void swap(int &, int &)

Therefore, by passing the references the function works with the original values (i.e., the same memory area in which original values are stored) but it uses alias names to refer to them. Thus, the values are not duplicated.

### 11.4.1 Pointers to Functions

When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using the formal arguments (the addresses of original values) in the called function, we can make changes in the actual arguments of the calling function.



*Example:* Program to swap values of two variables by passing pointers.

```
#include<iostream.h>
#include<conio.h>                          // for clrscr( )
int main( )
{ clrscr( );
void swap(int *x, int *y);
// prototype
int a = 7, b = 4;
cout<< "Original values \n";
cout<< "Original values \n";
cout<< " a=" << a << ",b=" << b<< "\n";
Swap(&a, &b);                              // function call
cout<< "swapped values \n";
cout<< "a=" << a <<, "b=" << b << "\n";
return 0;
}
//function definition
void swap(int *x, int * y)
{      int temp;
```

Notes

```
        temp = *x;
        *x = *y;
        *y  = temp;
}
```

Output: Original values

a = 7, b = 4

Swapped values

a = 4, b= 7



*Notes* The called function does not create own copy of original values by the addresses (passed through pointers) it receives.

### 11.4.2 Function Returning Pointers

The way a function can returns an int, a float, a double, or reference or any other data type, it can even return a pointer. However, the function declaration must replace it. That is, it should be explicitly mentioned in the function's prototype. The general form of prototype of a function returning a pointer would be type function-name (argument list);

The return type of a function returning a pointer must be known because the pointer arithmetic is relative to its base type and a complier must know what type of data the pointer is pointing to in order to make it point to the next data item.



*Example:* Program to illustrate a function returning a pointer.

```
#include<iostream.h>
#include<conio.h>
int *big(int&, int&)                    //prototype
int main( )
{       clrscr( )
        int a, b, *c;
        cout<<"Enter two integers \n";
        cin>>a>>b;
        c= big(a, b);
        cout<<"The bigger value is "<< *c<< "\n";
return 0;
}
int *big(int &x, int &y)
{ if   (x>y)
        return(&x);
```

Notes

```
else
        return(&y);
}
if input = 7  13
then, output : The bigger value is 13
```

### Self Assessment

Fill in the blanks:

8. Pointer is a variable, which contains …….. of a variable.
9. A function is a user defined operation that can be …….. by applying the call operator ("( )") to the function's name.
10. A pointer to a variable holds its memory address using which the memory area storing the data value of the variable can be …… accessed.

## 11.5 Dynamic Memory Management

Dynamic memory is allocated by the **new** keyword. Memory for one variable is allocated as below:

**ptr=new DataType (initializer);**

Here,

1. ptr is a valid pointer of type DataType.
2. DataType is any valid c++ data type.
3. Initializer (optional) if given, the newly allocated variable is initialized to that value.

*Task* Dynamic memory allocation is the allocation of memory at "run time". Discuss.

*Example:*

```
//Example Program in C++
#include<iostream.h>
void main(void)
{
 int *ptr;
  ptr=new int(10);

  cout<<*ptr;

  delete ptr;
}
```

Notes

This is will allocate memory for an int(eger) having initial value 10, pointed by the **ptr** pointer.

Memory space for arrays is allocated as shown below:

**ptr=new DataType [x];**

Here,

1. ptr and DataType have the same meaning as above.
2. x is the number of elements and C is a constant.



*Example:*

```
//Example Program in C++
#include<iostream.h>

void main(void)
{
  int *ptr, size;

  cin>>size;
  ptr=new int[size];

  //arrays are freed-up like this
  delete []ptr;
}
```

### 11.5.1 New and Delete Operator

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

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

#### *Operators new and new[]*

In order to request dynamic memory we use the operator new. New is followed by a data type specifier and – if a sequence of more than one element is required – the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

pointer = new type

pointer = new type [number_of_elements]

Notes

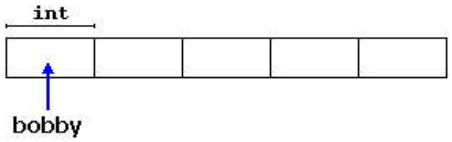
The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
int * bobby;

bobby = new int [5];
```

In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby.

*Notes* Note the now, bobby points to a valid block of memory with space for five elements of type int.




The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression *bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with bobby[1] or *(bobby+1) and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to 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 us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the default method used by new, and is the one used in a declaration like:

```
bobby = new int [5];  // if it fails an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument fornew:

```
bobby = new (nothrow) int [5];
```

Notes

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if bobby took a null pointer value:

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
   // error assigning memory. Take measures.
 };
```



*Caution* This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation.

Anyway this method can become tedious for larger projects, where the exception method is generally preferred.

### *Operators delete and delete[]*

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

delete pointer;

delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;
int main ()
{
   int i,n;
   int * p;
   cout << "How many numbers would you like to type? ";
   cin >> i;
   p= new (nothrow) int[i];
   if (p == 0)
      cout << "Error: memory could not be allocated";
   else
   {
```

Notes

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

How many numbers would you like to type? 5

Enter number : 75

Enter number : 436

Enter number : 1067

Enter number : 8

Enter number : 32

You have entered: 75, 436, 1067, 8, 32,

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

p= new (nothrow) int[i];

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated).



*Notes* Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

### 11.5.2 The this Pointer

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object on which this function was called. The pointer 'this' acts as an implicit argument to all the member function apart from the explicit arguments passed to it.

**Notes**

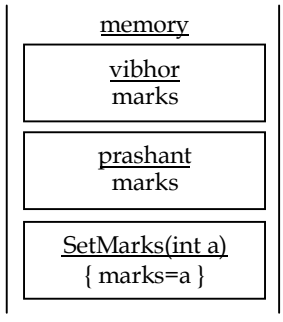
When a number of objects are created from the same class each object's data members are created as separate copies. However, only a single copy of methods is retained in the memory. That is all objects of a class share a single copy of the compiled class functions. A particular object is referenced by "this" pointer internally.

*Task* What do you think the disadvantages of this pointer?

Consider the following code snippet

```
class  student
{
   int  marks;
public:
   void  setMarks(int  a)
   {
          marks  =  a;
   }
int  main()
{
   student  vibhor,  prashant;
   vibhor.setMarks(86);
   prashant.setMarks(67);
}
```

When the code is executed, two marks variables are created - one for vibhor and one for prashant, but only one copy of setMarks function is kept in the memory (see the figure below).



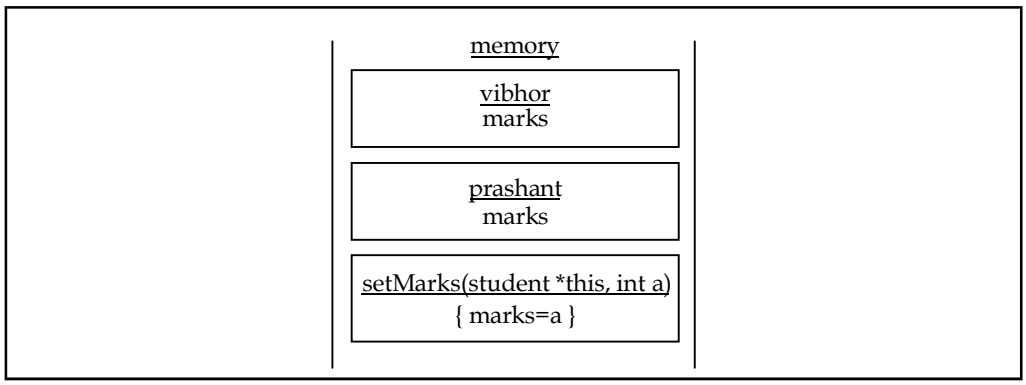

Both the objects can execute the setMarks function. That is both the function calls given below will work correctly.

vibhor.setMarks(86);

prashant.setMarks(67);

How is this possible? What happens is that at the compile time the compiler inserts the necessary code into the function setMarks so that it receives the 'this' pointer in addition to an int value as argument as shown below.

Notes




When vibhor calls setMarks, a pointer to vibhor is passed to setMarks telling it that it is object vibhor and not prashant that called setMarks.

## Self Assessment

Fill in the blanks:

11. Dynamically allocated memory is kept on the …… .
12. Declarations are used to …… allocate memory, the new operator is used to dynamically allocate memory.
13. The …… operator performs pointer arithemtic and de-references the resulting pointer.
14. A static member function does not have a ……. pointer.
15. The type of the this pointer for a member function of a class type X, is ……. const.



*Caselet*

### Almost in the Maker's Shoes

Art is lies that tell the truth, said Piccaso. Art is art; everything else is everything else, said Ad Reinhardt. Art completes what nature cannot bring to finish, is an Aristotle quote.

If art imitates life, one form of such exercise is animation. Peter Ratner's "3-D Human Modeling and Animation" has all the tools and know-how "to create digital characters that can move, express emotions and talk". The book demonstrates how you can use your artistic skills in figure drawing, painting, and sculpture to create animated human figures using the latest computer technology. "No one has been able to make computer graphics humans that have been mistaken for real ones in movies and photos when viewed at close range," states the preface. "Until technology evolves to the point that this becomes possible, creating an artistic representation of a human is still a worthwhile goal." A few frames from the book:

The closer a character becomes to an everyday human, the more ordinary it will appear. Synthetic humans most often lack personality. Computer characters that try to mimic human movement through unedited motion-capture techniques generally look like puppets

*Contd...*

Notes

or store mannequins that have come to life. As contradictory as it sounds, when animators exaggerate the movements and expressions of their characters, they appear more lifelike and realistic.

Generally, the average height of a man or a woman can be measured as seven heads tall. If your goal is to create and animate the ideal male and female, then consider modelling them eight heads tall. Stretch their proportions first and then use them as your guides. Superheroes are often portrayed as very tall with tiny heads.

Some artists prefer to start with the smallest unit possible: the point, or vortex. After placing a series of these vertices, one can connect them as a spline or create polygons from them. Splines are flexible line segments defined by edit points or vertices. Sometimes splines are referred to as curves. A series of connected splines make a wire mesh. Adjoining wire meshes are patches. Thus spline modelling lends itself to the patch modelling method.

Preparing the human model for facial expressions and dialogue is an essential part of the 3-D animation process. A base model with a neutral expression is the starting point. The base model should have at least three sets of parallel lines for each wrinkle. Approximately 56 shapes or morphs, which include the mouth shapes for dialogue, should be enough for the majority of facial expressions.

Conventional art materials such as charcoal, paint, clay, fibre, and so on, are tactile. The artist who works with these has an emotional link that is often lacking when compared to the one who relies on hardware and software. The computer artist is forced to instill the quality of emotion into a medium that is for the most part cerebral. This is one of the greatest challenges. Without emotional content, the work will appear cold and removed from the human experience. Unlike other artists, computer animators work mostly in the mental realm to put feeling into their work.

If only animators could breathe life into their creations, would life not be one big comics book to sit back and read?

Book courtesy: Wiley Dreamtech India P Ltd.

Onto the next generation of Web-based technology

WEB services is an umbrella term. It describes a collection of industry-standard protocols and services used to facilitate a 'base-line level of interoperability' between applications. Which, in other words, means all different systems can get talking over the cyberspace. "Building XML Web Services for the Microsoft .NET Platform" by Scott Short deals with the basic building blocks of XML Web services, viz. Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI) and so on. There's more:

WSDL documents can be intimidating at first glance. But the syntax of a WSDL document is not nearly as complex as that of an XML Schema document. A WSDL document is composed of a series of associations layered on top of an XML Schema document that describes a Web Service. These associations add to the size and the perceived complexity of a WSDL document. But once you look underneath the covers, WSDL documents are rather straightforward.

HTTP is by nature a stateless protocol. Even with the introduction of the connection keep-alive protocol in HTTP 1.1, you cannot assume that all requests from a given client will be sent over a single connection. ASP.NET provides a state management service that can be leveraged by Web Forms and Web services.

*Contd...*

Notes

instantiate the object. From the client's perspective, the object already exists and calls can be made to it without the need to create or initialise the object. This is the default behaviour of SOAP-based Web services.

Digest authentication does not transfer the user's password in the clear; instead a hash, or digest, of the password and data provided by the server is used to authenticate the user.

From a developer's perspective, .NET My Services eliminates many of the problems of securing data, providing encrypted transport channels, and reconciling disparate data sources. And all of this is achievable using XML Web services, so businesses are spared the drastic learning curve associated with new technologies.

Remember, they say Web services are the next big thing.

**From OOP to more**

LET us say you already understand OOP (object-oriented programming) concepts such as data abstraction, inheritance and polymorphism. And that you want to 'leverage the power of .NET Framework to build, package and deploy any kind of application'. Jeffrey Richter's "Applied Microsoft .NET Framework Programming" has the answers. Read on:

As an application runs, the common language runtime (CLR) maintains a 'snapshot' of the set of assemblies loaded by the application. When the application terminates, this information is compared with the information in the application's corresponding .ini file. If the application loaded the same set of assemblies that it loaded previously, the information in the .ini file matches the information in memory and the in memory information is discarded.

Sometimes the add and remove methods the compiler generates are not ideal. For example, if you're adding and removing delegates frequently and you know that your application is single-threaded, the overhead of synchronising access to the object that owns the delegate can really hurt your application's performance.

Compilers convert code that references an enumerated type's symbol to a numeric value at compile time. Once this occurs, no reference to the enumerated type exists in metadata and the assembly that defines the enumerated type doesn't have to be available at run time. If you have cod that references the enumerated type - rather than just having references to symbols defined by the type - the assembly that defines the enumerated type will be required at run time.

A try block doesn't have to have a finally block associated with it at all; sometimes the code in a try block just doesn't require any cleanup code. However, if you do have a finally block, it must appear after any and all catch blocks, and a try block can have no more than one finally block associated with it.

If the CLR suspends a thread and detects that the thread is executing unmanaged code, the thread's return address is hijacked and the thread is allowed to resume execution. A pinned object is one that the garbage collector isn't allowed to move in memory.

## 11.6 Summary

- A pointer is a variable that holds the memory address of the location of another variable in memory. A pointer is declared in the following form:

  type * var_name ;

  where type is a predefined C++ data type and var_name is the name of the pointer variable.

Notes

- The operator &, when placed before a variable, returns the memory address of its operand. The operator * returns the memory address of its operand.
- The operator * returns the data value stored in the area being pointed to by the pointer following it.
- The pointer variables must always point to the correct type of data. Pointers must be initialized properly because uninitialized pointers result in the system crash.
- In pointer arithmetic, all pointers increase and decrease by the length of the data type point to.
- An array name is a pointer that stores the address of its first element. If the array name is incremented, It actually points to the next element of the array.
- Array of pointers makes more efficient use of available memory. Generally, it consumes lesser bytes than an equivalent multi-dimensional array.
- Functions can be invoked by passing the values of arguments or references to arguments or pointers to arguments.
- When references or pointers are passed to a function, the function works with the original copy of the variable. A function may return a reference or a pointer also.

## 11.7 Keywords

***Alias:*** A different name for a variable of C++ data type.

***Base Address:*** Starting address of a memory location holding array elements.

***Function Pointer:*** A function may return a reference or a pointer variable also. A pointer to a function is the address where the code for the function resides. Pointer to functions can be passed to functions, returned from functions, stored in arrays and assigned to other pointers.

***Memory location:*** A container that can store a binary number.

***Pointer:*** A variable holding a memory address.

***Reference:*** An alias for a pointer that does not require de-referencing to use.

## 11.8 Review Questions

1. How does pointer variable differ from simple variable?
2. How do we create and use an array of pointer-to-member-function?
3. How can we avoid syntax errors when creating pointers to members?
4. How can we avoid syntax errors when calling a member function using a pointer-to-member-function?
5. How do we pass a pointer-to-member-function to a signal handler, X event callback, system call that starts a thread/task, etc?
6. Find the syntax error (s), if any, in the following program:

```
{
    int x [5], *y [5]
  for (i = 0; i< 5; i++)
  { x [i] = I;
```

Notes

```
    x[i] = i + 3;
    y = z;
    x = y;
}
```

7. Discuss two different ways of accessing array elements.
8. Write a program to traverse an array using pointer.
9. Write a program to compare two strings using pointer.
10. Can we convert a pointer-to-function to a void*? Explain with an example.

### Answers: Self Assessment

| | | | |
|---|---|---|---|
| 1. | memory | 2. | '&' |
| 3. | indirection | 4. | unique |
| 5. | integers | 6. | comparison |
| 7. | declaration | 8. | address |
| 9. | invoked | 10. | directly |
| 11. | memory heap | 12. | statically |
| 13. | [ ] | 14. | This |
| 15. | X* | | |

## 11.9 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

*Object Oriented Programming with C++*; Cyber tech publications



*Online links*

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101 index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr035.htm

http://www.cplusplus.com/doc/tutorial/pointers/

**Notes**

# Unit 12: Console I/O

**CONTENTS**

Objectives

Introduction

12.1 Concept of Streams

12.2 Hierarchy of Console Stream Classes

12.3 Unformatted I/O Operations

12.4 Managing Output with Manipulators

12.4.1 Manipulate Function

12.4.2 Predefined Manipulators

12.5 Summary

12.6 Keywords

12.7 Review Questions

12.8 Further Readings

## Objectives

After studying this unit, you will be able to:

- Recognize the concepts of streams
- Describe the hierarchy of console stream classes
- Explain the unformatted I/O operations
- Discuss the managing output with manipulators

## Introduction

One of the most essential features of interactive programming is its ability to interact with the users through operator console usually comprising keyboard and monitor. Accordingly, every computer language (and compiler) provides standard input/output functions and/or methods to facilitate console operations.

C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored. This way, C++ is able to treat all the input and output operations in a uniform manner. Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.

We have used the objects cin and cout (predefined in the iostream.h file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the general format for reading data from the keyboard:

cin >> variable1 >> variable2 >>.. ...>> variableN;

Notes

where variable1, variable2, . . .. are valid C++ variable names that have been declared already. This statement will cause the computer to halt the execution and look for input data from the keyboard. The input data for this statement would be:

datal data2……dataN

The input data are separated by white spaces and should match the type of variable in the cin list. Spaces, new lines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. 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. For example, consider the following code:

int code;

cin >> code;

Suppose the following data is given as input:

1267E

The operator will read the characters up to 7 and the value 1267 is assigned to code. The character E remains in the input stream and will be input to the next cin statement. The general format of outputting data:

cout << iteml <<item2 << .. ..<< itemN;



*Caution* The items item1 through itemN may be variables or constants of any basic types.

## 12.1 Concept of Streams

A stream is a source of sequence of bytes. A stream abstracts for input/output devices. It can be tied up with any I/O device and I/O can be performed in a uniform way. The C++ iostream library is an object-oriented implementation of this abstraction. It has a source (producer) of flow of bytes and a sink (consumer) of the bytes. The required classes for the stream I/O are defined in different library header files.

To use the I/O streams in a C++ program, one must include iostream.h header file in the program. This file defines the required classes and provides the buffering. Instead of functions, the library provides operators to carry out the I/O. Two of the Stream Operators are:

<< : Stream insertion for output.

>> : Stream extraction for input.

The following streams are created and opened automatically:

cin : Standard console input (keyboard).

cout : Standard console output (screen).

cprn : Standard printer (LPT1).

cerr : Standard error output (screen).

clog : Standard log (screen).

caux : Standard auxiliary (screen).

Notes

*Example:* The following program reads an integer and prints the input on the console.

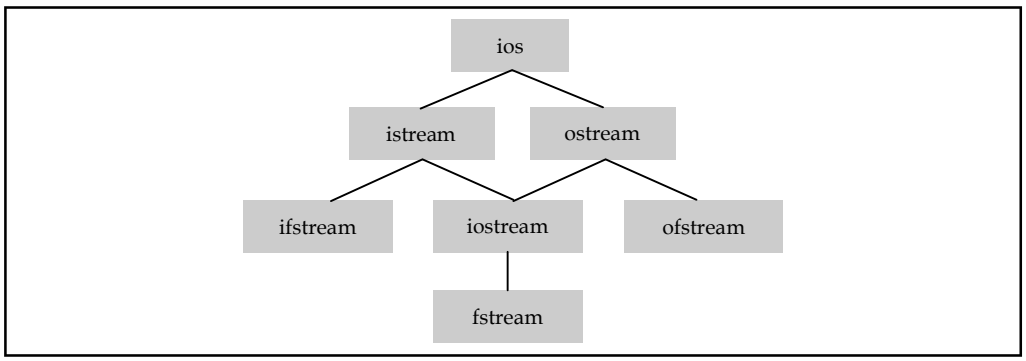
```
#include <iostream> // Header for stream I/O.
int main(void)
{
        int p;          //   variable to hold the input integer
        cout << "Enter an integer: ";
        cin >> p;
        cout << "\n You have entered " << p;
}
```

## 12.2 Hierarchy of Console Stream Classes

Streams can also be tied up with data files. The required stream classes for file I/O are defined in fstream.h and/or strstream.h.

fstream : File I/O class.

| | |
|---|---|
| ifstream | Input file class. |
| istrstream | Input string class. |
| ofstream | Output file class. |
| ostrstream | Output string class. |
| strstream | String I/O class. |




There are some special functions that can alter the state the stream. These functions are called manipulators. Stream manipulators are defined in iomanip.h.

| | |
|---|---|
| dec | Sets base 10 integers. |
| endl | Sends a new line character. |
| ends | Sends a null (end of string) character. |
| flush | Flushes an output stream. |
| fixed | Sets fixed real number notation. |
| hex | Sets base 16 integers. |

*Contd...*

Notes

| | |
|---|---|
| oct | Sets base 8 integers. |
| ws | Discard white space on input. |
| setbase(int) | Sets integer conversion base (0, 8, 10 or 16 where 0 sets base 10). |
| setfill(int) | Sets fill character. |
| setprecision(int) | Sets precision. |
| setw(int) | Sets field width. |
| resetiosflags(long) | Clears format state as specified by argument. |
| setiosflags(long) | Sets format state as specified by argument. |

The stream classes have a variety of member functions to give them their required functionalities. Thus, there is a function to open the stream, one for reading/writing, one for closing the stream and the like. The stream class member functions are listed below:

| | |
|---|---|
| void .close() | Closes the I/O object. |
| int .eof() | Returns a nonzero value (true) if the end of the stream has been reached. |
| char .fill(char fill_ch \| void) | Sets or returns the fill character. |
| int .fail() | Returns a nonzero value (true) if the last I/O operation on the stream failed. |
| istream& .get(int ch) | Gets a character as an int so EOF (-1) is a possible value. |
| istream& .getline(char* ch_string, int maxsize, char delimit) | Get a line into the ch_string buffer with maximum length of maxsize and ending with delimiter delimit. |
| istream& .ignore(int length[, int delimit]) | Reads and discards the number of characters specified by length from the stream or until the character specified by delimit (default EOF) is found. |
| iostream& .open(char* filename, int mode) | Opens the filename file in the specified mode. |
| int .peek(); | Returns the next character in the stream without removing it from the stream. |
| int .precision(int prec \| void) | Sets or returns the floating point precision. |
| ostream& .put(char ch) | Puts the specified character into the stream. |
| istream& .putback(char ch) | Puts the specified character back into the stream. |
| istream& .read(char* buf, int size) | Sends size raw bytes from the buf buffer to the stream. |
| long .setf(long flags [, long mask]) | Sets (and returns) the specified ios flag(s). |
| long .unsetf(long flags) | Clears the specified ios flag(s). |
| int .width(int width \| void) | Sets or returns the current output field width. |
| ostream& .write(const char* buf, int size) | Sends size raw bytes from buf to the stream. |

A file may be opened for a number of file operations. The corresponding stream must be set with the intended operation. The different file stream modes are indicated by File Access Flags as listed below:

**Notes**

| | |
|---|---|
| Ios::app | Open in append mode. |
| Ios::ate | Open and seek to end of file. |
| Ios::in | Open in input mode. |
| Ios::nocreate | Fail if file doesn't already exist. |
| Ios::noreplace | Fail if file already exists. |
| Ios::out | Open in output mode. |
| Ios::trunc | Open and truncate to zero length. |
| Ios::binary | Open as a binary stream. |



*Example:*

1. Basic Program File Structure and Sample Function Call with stream I/O

```
#include <iostream>    // Header for stream I/O.
#include <iomanip>     // Header for I/O manipulators.
                // Function declaration (prototype) with default values.
float sum(float required_term, float optional_term = 0.0);
                                                        //main function
int main(void)
{
  int p;               //   Output numeric precision.
  float a;             // Units and description of a.
  float b;             // Units and description of b.
  cout.setf(ios::showpoint);
  cout << "Enter the output precision (an integer): ";
  cin >> p;
  if (!cin)
  {
        cout << "Input error.\n";
        return 1;
  }
  cout << setprecision(p);
  cout << "Enter two real numbers to be summed: ";
  cin >> a >> b;
  if (!cin)
  {

        cout << "Input error\n";
        return 2;
```

**Notes**

```
  }
  cout << "Entered values: a = " << a << " and b = " << b << endl;
  cout << "sum(a, b) = " << sum(a, b) << endl;
  cout << "sum(a) = " << sum(a) << endl;
  return 0;
}
  // Define the sum function.
float sum(float x, float y)
{
  return (x + y);
}
```

2. This program displays use of different stream manipulators.

```
#include <iostream.h>
#include <iomanip.h>
int main(void)
{
  int I = 100;
cout << setfile('.');
cout << setiosflags(ios::left);
cout << setw(20) << "Decimal";
cout << resetiosflags(ios::left);
cout << setw(6) << dec << I << endl;
cout << setiosflags(ios::left);
cout << setw(20) << "Hexadecimal";
cout << resetiosflags(ios::left);
cout << setw(6) << hex << I << endl;
cout << setiosflags(ios::left);
cout << setw(6) << oct << I << endl;
}
```

The output of the program:

Decimal…….100

Hexadecimal……64

Octal……..144

3. This program exchanges the values of two variable.

```
#include <iostream>    // Header for stream I/O.
#include <fstream>     // Header for file I/O.
#include <iomanip>     // Header for I/O manipulators.
```

**Notes**

```
// Function declaration use references and without default values.
void swap(float& first_indentifier, float& second_indentifier);
// Main program
int main(void)
{
int p = 3;      // Output numeric precision.
float a = 2.5f; // Units and description of a (f indicates float value).
float b = 7.5f; // Units and description of b.
ofstream fout;  // Declare an output file object.
fout.open("swap.out", ios::out);     // Open the output file.
if (!fout)             // See if the file was opened successfully.
{
  cout << "Can't open output file!\n";
  return 1;
}
fout.setf(ios::showpoint);    // Show decimal points.
fout << setprecision(p);      // Set the precision.
fout << "Before swapping...\n";
fout << "a = " << a << " and b = " << b << endl;
swap(a, b);
fout << "After swapping...\n";
fout << "a = " << a << " and b = " << b << endl;
fout.close();                 // Close the out file.
return 0;
}
// Define the swap function.  Use references so argument changes are returned.
void swap(float& x, float& y)
{

float hold;
hold = x;
x = y;
y = hold;
return;
}
```

Notes

4. This program demonstrates the reading of data files containing comments and displaying them on the screen. The lines in the input file starting with '!' will be treated as comment line. The input file name is comments.txt having the following sample data:

```
!
! These comment lines should be ignored.
!
This is the first non-comment line.
This is the second non-comment line.
Almost done.
This is the last non-comment line.
 // Included files.
#include <iostream>    // Header for console I/O
#include <fstream>     // Header for file I/O.
#include <string>      // Header for STL strings.
// Define skip_comments function.
int skip_comments(ifstream& file, char mark)
{
/*
This function skips the all the lines at the start of the specified
file that begin with the specified character.  The file must already be
open when this function is called.  Error checking not yet included.
*/
const int MAX_CHARS = 100; // This is the maximum characters per line.
while (file.peek() == mark)
{
file.ignore(MAX_CHARS, '\n');}
return 0;
}
//main function
int main()
{
// Open input file.
ifstream fin;                          // Declare an input file object.
fin.open("comments.txt", ios::in);     // Open the input file in project
                                       //folder.
if (!fin)              // See if the file was opened successfully.
{
  cout << "Can't open input file. \n";
  return 1;
```

Notes

```
}
cout << "About to skip comments.\n";
skip_comments(fin,'!');
string sometext;
getline(fin, sometext);
while (!fin.fail())
{
cout << sometext << '\n';
getline(fin, sometext);
}
if (fin.fail() && !fin.eof())
{
cout << "Error while reading file. \n";
fin.close();
return 2;
}
return 0;
}
```

### *The put() and get() Functions*

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character. . .

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get(c);  //get a character from keyboard and assign it to c
while (c!= '\n')
{
cout << c; //display the character on screen cin.get (c);
//get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator> >can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement.

cin >> c;

is used in place of

Notes

cin.get(c);

Try using both of them and compare the results. The get(void) version is used as follows:

char c;

c = cin.get(); //cin.get(c) replaced

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

cout << put('x');

displays the character x and

cout << put(ch);

displays the value of variable ch.

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

cout << put(68); .

displays the character D. This statement will convert the int value 90 to a char value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

char c;

cin.get (c); //read a character

while(c!= '\n')

{

cout << put(c); //display the character on screen cin.get (c );

}

### *The getline () and write () Functions*

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

cin.getline(line, size);

This function call invokes the 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 number of characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character. For example; consider the following code:

char name[20];

cin.getline(name, 20); .

Assume that we have given the following input through the keyboard:

Neeraj good

Notes

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

*Object Oriented Programming*

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string, cin automatically adds the terminating null character to the character array.

Remember, the two blank spaces contained in the string are also taken into account, i.e. between Objects and Oriented and Pro.

We can also read strings using the operator >> as follows:

cin >> name;

But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as "Neeraj good".



*Notes* Characters are extracted until either (n – 1) characters have been extracted or the delimiting character is found (which is delim if this parameter is specified, or '\n' otherwise). The extraction also stops if the end of file is reached in the input sequence or if an error occurs during the input operation.

### Self Assessment

Fill in the blanks:

1. ……. extracts characters from the input sequence and stores them as a c-string into the array beginning at s.
2. The base of the …….. library is the hierarchy of class templates.
3. A stream is an abstraction that represents a device on which input and …….. operations are performed.
4. As part of the …… library, the header file <iostream> declares certain objects that are used to perform input and output operations on the standard input and output.
5. Streams are generally associated to a …… source or destination of characters.
6. Once a file stream is used to open a file, any input or output operation performed on that stream is physically reflected in the…….. .
7. The class templates in this class hierarchy have the same name as their …….. instantiations but with the prefix basic_.

## 12.3 Unformatted I/O Operations

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file. Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file. Formatted input reads characters from the input file and

converts them to internal form. Formatted I/O can be either "Free" format or "Explicit" format, as described below.

Notes

### *Advantages and Disadvantages of Unformatted I/O*

Unformatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data. Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation. It should be noted that XDR (eXternal Data Representation) files, described in Portable Unformatted Input/Output, can be used to produce portable binary data.



*Did u know?* **Is unformatted input/output is not directly human readable?**

Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

### *Advantages and Disadvantages of Formatted I/O*

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.) Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

However, formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text. Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

### *Formatted Console I/O Operations*

C++ supports a number of features that could be used for formatting the output. These features include:

1. ios class functions and flags.
2. Manipulators.
3. User-defined output functions.

The ios class contains a large number of member functions that could be used to format the output in a number of ways. The most important ones among them are listed below.

| Function | Task |
|---|---|
| width() | To specify the required field size for displaying an output value |
| precision() | To specify the number of digits to be displayed after the decimal point of a float value |
| fill() | To specify a character that is used to fill the unused portion of a field. |
| setf() | To specify format flags that can control the form of output display (such as Left-justification and right-justification). |
| unsetf() | To clear the flags specified. |

Notes

Manipulators are special functions that can be included in these statements to alter the format parameters of a stream. The table given below shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip.h should be included in the program.

| Manipulators | Equivalent ios function |
|---|---|
| setw() | width() |
| setprecision() | precision() |
| Setfill() | fill() |
| setiosflags() | setf() |
| resetiosflags() | unsetf() |

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats.



*Task* Analyze the difference between the advantages of unformatted and formatted I/O.

### Self Assessment

Fill in the blanks:

8. …….. input/output transfers the internal binary representation of the data directly between memory and the file.
9. Unformatted data files can only be moved easily to and from computers that share the same internal……...
10. Formatted input/output is more computationally ……. than unformatted input/output.
11. The …… class contains a large number of member functions that could be used to format the output in a number of ways.
12. ……. are special functions that can be included in these statements to alter the format parameters of a stream.

## 12.4 Managing Output with Manipulators

These operators are used to format the data display. The commonly used manipulators are endl and setw.

1. endl manipulator when used causes a linefeed to be inserted. For example,

```
cout     <<"m=" <<m <<endl
  <<"n=" <<n <<endl
  <<"p=" <<p <<endl;
```

would cause the output as:

Assuming the values of variables as 2597, 14 and 175.

m = 2597

Notes

n = 14

p = 175

2. The setw manipulator does the job as follows:

cout <<setw(5) <<sum <<endl;

The manipulator setw(5) specifies a field width 5 for printing. The value of the variable sum is printed as 5 digits. This value is right justified within the field.

| | | | | |
|---|---|---|---|---|
| | | 3 | 4 | 5 |

We can also write our own manipulators as:

```
#include <iostream.h>

ostream & Symbol (ostream & output)

{

        return  output  <<"\+₹";

}
```

The symbol is the new manipulator, which represents ₹ The identifier symbol can be used

Wherever we need to display the string "₹".

### 12.4.1 Manipulate Function

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. The most commonly used manipulators functions are given below. To access these function, we must include the file iomanip.h in the program.

1. Setw (int w): As discussed earlier, this manipulator function is used to changes or set the field width for output to w.
2. Set fill (char C): It is used to change or set the fill character to n (default is a space).
3. Set precision (int p): It is used to change or set the floating point precision to p.
4. Set base (base n): It is used to changes base to n, where n is 8, 10, or 16. If n is zero, output is base 10, but input uses the C convention: 10 is 10, 010 is 8 and O x c is 12.
5. Set Pos flags (print flags f): This manipulator function changes or sets the format flag f. Setting remain in effect until next change.
6. Reset i os flags (fmt flags f): This manipulators clears only the flags specified by f. Setting remains in effect until next change.

### 12.4.2 Predefined Manipulators

Manipulators comes in tow flavors: Those take an argument as discussed in previous topic and those that don't. Manipulators with no arguments are known as predefined manipulators. These are provided in iostream.h. The most commonly used predefined manipulators are given below:

1. end l; This manipulator sends a newline to the stream and flushes it.
2. skipws: It is used to skip white space on input.

Notes

3. no skipws: Its purpose is not to skip white space on input.
4. dec: it is used for the decimal conversion.
5. oct; : It is used for the octal conversion.
6. hex: It is used for hexadecimal conversion.
7. left; This manipulator is used to left align and pad on right in the field.
8. right: This manipulator is used to right align and pad on left in the field.
9. internal: It is used for padding between sign or base indicator and value.
10. showpos: Shows plus sign for positive values.
11. noshowpos: Do not show plus sign for positive value.
12. uppercase: It displays uppercase A-F for hex values and E for Scientific values.
13. no upper case: It do not display hex values in upper case.
14. show point: It shows decimal point and trailing zeros for float values.
15. scientific: It uses scientific notation for printing float value.
16. fixed: It uses fixed notation for printing float values.
17. ends; it inserts null character to terminate an output string.
18. flush; It flushes the output stream.
19. lock: It unlocks the file handling.



*Example:* Let us consider an example to understand the usage of above said manipulators:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
        int I = 2b;
        float b = 123.500032;
        char str[ ] = "I am a c++ programmer";
        clrscr( );
cout<<setios flags (ios: :unitbuf | ios: :stdio | ios:: showpos);
        cout<<i<<end l;
        cout<< set iosflogs(ios :: showbase | ios :: uppercase);
        cout << hex << i << end;
        cout < < oct << i << endl;
        cout < < set fill ('o');
        cout << set w (40) <<str << endl;
```

## Self Assessment

Notes

Fill in the blanks:

13. …… manipulator function changes or sets the format flag f. Setting remain in effect until next change.

14. …….. manipulators clears only the flags specified by f. Setting remains in effect until next change.

15. Manipulators with no arguments are known as …….. manipulators.



*Caselet*

### Finance with the 'e' edge

Bank failures, non-performing loans and unattractive deposit rates are all enough to demotivate when one thinks of our financial system. But V.C. Joshi happily logs in to the future with his book e-Finance, published by Response Books (www.indiasage.com) . There he talks about the potential that banks and financial institutions have "to improve the quality and scope of the financial services and products that they offer."

Joshi describes the 'online value chain' where customers access through various devices and banking services are delivered through a network. He points out that banks often host Web sites but find them not to be of much use. "A financial institution must make its presence felt," says the author.

A chapter is devoted to e-finance products and services such as Cyber Gold, E-charge, Ipin and Millicent. That e-finance lowers costs and increases availability is something for the CFO to factor in when considering investments in IT. Joshi points out that not much attention has been bestowed upon the development of Internet platforms to trade and pledge electronic warehouse receipts; "this may reduce the need for government to purchase commodities for stock piling."

The book anticipates that more investors would make use of e-trading, though "unfortunately online trading coincided with the market meltdown." The e-thing has the potential to make markets more transparent, Joshi would add. "It is not restricted to information about price alone, but the user has the full log of the transaction behaviour."

HDFC Bank is a model that can be emulated, says the book, because the bank's policies are not only technically superior but also highly profitable. "Its treasury, corporate and even retail activities are mostly automated with a strong focus on online connectivity and e-commerce."

The book discusses topics such as risk, crime, law, security and so on. You need to engage in not just system development but survivable system development. To survive in e-finance, it would be advisable to log in to Joshi first.

**Byte is not Dracula's favourite pastimemisspelled**

AFTER C comes D. Wrong, it's C++. Mystified? Clear up the cloud with Jeff Kent's C++ Demystified, a self-teaching guide from Tata McGraw-Hill (www.tatamcgrawhill.com) . "C++ was my first programming language," writes Kent in his intro. He's learned many

*Contd...*

**Notes**

other languages but he thinks C++ is the best. Why? "Perhaps because of the power it gives the programmer." But, remember, that this power is a double-edged sword. Also, "knowing C++ makes learning other programming languages easier." What's the best way to learn programming? "Write programs."

Let's say, you ask, what's a programming language? The author begins with an analogy: "When you enter a darkened room and want to see what is inside, you turn on a light switch. When you leave the room, you turn the light switch off. The first computers were not too different than that light switch." From there you move on to `Hello World,' and before you dump C++ and say `Cruel World', the author would lead you to the innards of the language, all the while keeping you in good humour.

For instance, to explain bits and bytes, he writes: "While people live at street addresses, what is stored at each memory address is a byte. Don't worry, I have not misspelled Dracula's favourite pastime." Similarly, the chapter on variables begins thus: "Recently, while in a crowded room, someone yelled `Hey, you!' I and a number of other people looked up, because none of us could tell to whom the speaker was referring." So? "We use names to refer to each other. Similarly, when you need to refer in code to a particular item of information" call it by name.

"Variables, like people, have a lifetime," Kent would write in a different chapter. "A person's lifetime begins at birth. A variable's lifetime begins when it is declared. A person's lifetime ends with death.

A variable's lifetime ends when it goes out of scope." That's some philosophy demystified, shall we say?

**Test at all costs**

WANT to get into software testing as a career? Go for Dr K.V.K.K. Prasad's Software Testing Tools, published by Dreamtech Press (www.wileydreamtech.com) .

"Many software engineers have a wrong notion that software testing is a second-rate job," notes the preface. For them, what's first rate is development. "These engineers tend to forget that testing is a part of development." How?

Because only through testing can you deliver a quality product. Prasad lists the four criteria for a software project's success: "Meet all quality requirements; be developed within the time frame; be developed within the budget; and maintain a cordial relationship among the team members."

Testing is detested because it is tough. Testing process is iterative. So, test the software in the lab, and also in actual working environment (called beta testing). Any test creates stress for those who are tested; and `stress testing' is to test the software "at the limits of its performance specifications". You accept that testing is important, but `acceptance testing' is the most important testing, the author would emphasise. It decides whether the client approves the product or not.

Oracle is a popular name in software, but test oracles are people or machines used for checking the correctness of the program for the given test cases, explains the book. "Human test oracles are used extensively if the program does not work." In ancient times, when people had a problem, it's said they'd go to a priest or priestess, called an oracle. He or she would act as a medium for divine advice or prophecy. The word is derived from Latin oraculum, from ovare, speak.

*Contd...*

Notes

The book covers Mercury Interactive's WinRunner, Segue Software's SilkTest, and IBM Rational SQA Robot. There are case studies that illustrate the use of LoadRunner, JMeter, TestDirector and so forth.

As in hospitals where the severity of the problem determines whether the patient will be in a ward or ICU, software defects are classified depending on their impact on the functionality of the software.

"Critical defects result in system-crash, while major ones may result in some portions of the application difficult to use." There are also minor defects; these "can be tolerated" as in the case of "lack of help for some functionality, spelling mistake in an error message and so on."

For IT managers, it would be a critical defect to be ignorant of how software is tested; that would be a flaw with a potential to cause a career to crash.

## 12.5 Summary

- C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored.
- This way, C++ is able to treat all the input and output operations in a uniform manner.
- Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.
- A stream is a source of sequence of bytes.
- A stream abstracts for input/output devices. It can be tied up with any I/O device and I/O can be performed in a uniform way.
- The C++ iostream library is an object-oriented implementation of this abstraction.
- It has a source (producer) of flow of bytes and a sink (consumer) of the bytes.
- The required classes for the stream I/O are defined in different library header files.
- Streams can also be tied up with data files.
- Unformatted input/output is the simplest and most efficient form of input/output.
- It is usually the most compact way to store data. Unformatted input/output is the least portable form of input/output.

## 12.6 Keywords

***C++ iostream Library:*** The C++ iostream library is an object-oriented implementation of this abstraction.

***Stream:*** A stream is a series of bytes whose value depends on the variable in which it is stored.

***Unformatted Input/Output:*** Unformatted input/output is the simplest and most efficient form of input/output.

## 12.7 Review Questions

1. Why should we use <iostream> instead of the traditional <cstdio>?
2. How can we get std::cin to skip invalid input characters?

Notes

3. How does that funky while (std::cin >> foo) syntax work?
4. Why does my input seem to process past the end of file?
5. Why we should end the output lines with std::endl or '\n'?
6. Why shouldn't we always use a printOn() method rather than a friend function?
7. How can we provide printing for an entire hierarchy of classes?
8. How can we open a stream in binary mode?
9. How can we "reopen" std::cin and std::cout in binary mode?
10. How can we write/read objects of my class to/from a data file?

### Answers: Self Assessment

1. Getline() function
2. iostream
3. ouput
4. iostream
5. physical
6. file
7. char-type
8. Unformatted
9. data representation
10. expensive
11. ios
12. Manipulators
13. Set Pos flags (print flags f)
14. Reset i os flags (fmt flags f)
15. predefined

## 12.8 Further Readings



*Books*

E Balagurusamy; *Object-oriented Programming with C++*; Tata Mc Graw-Hill.

Herbert Schildt; *The Complete Reference C++*; Tata Mc Graw Hill.

*Object Oriented Programming with C++*; Cyber tech publications.



*Online links*

http://www.cplusplus.com/reference/iostream/

http://www.cplusplus.com/reference/iostream/istream/getline/

# Unit 13: Working with Files

Notes

CONTENTS

Objectives
Introduction
13.1 Data Streams
13.2 Opening a File
13.3 Reading/Writing a Character from/into a File
13.4 Appending
13.5 Processing and Closing a File
13.6 Different Types of Files
13.6.1 Types of File Systems
13.6.2 File Systems and Operating Systems
13.6.3 Binary Files
13.7 Command Line Arguments
13.8 Summary
13.9 Keywords
13.10 Review Questions
13.11 Further Readings

## Objectives

After studying this unit, you will be able to:

- Demonstrate the opening a file
- Recognize the reading and writing in a file
- Describe the appending in a file
- Explain the processing and closing a file
- Discuss the different types of files
- Discuss the command line arguments

## Introduction

Programs often input data from reading from a data file and output the result into another (or same) data file. This unit will focus on issues related to accessing data from a data file through a C++ program.

## 13.1 Data Streams

Programs would not be very useful if they cannot input and/or output data from/to users. Some programs that require little or no input for their execution are designed to be interactive through

Notes

user console - keyboard for input and monitor for output. However, when data volume is large it is generally not convenient to enter the data through console. In such cases data can be stored in a file and then the program can read the data from the data file rather than from the console.

The data may be organized in fixed size record or may be in free form text. The various operations possible on a data file using C++ programs are:

1. Opening a data file
2. Reading data stored in the data file into various variables and objects in the program
3. Writing data from a program into a data file
   (a) Removing the previously stored data in the file
   (b) Without removing the previously stored data in the file
      (i) At the end of the data file
      (ii) At any other location in the file
4. Saving the data file onto some secondary storage device
5. Closing the data file once the ensuing operations are over
6. Checking status of file operation



*Did u know?* **Where did data file exist?**

The data file itself can exist in many forms. It may contain textual data in ASCII format or binary data.

C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and from the program (Figure 13.1).

**Figure 13.1 (a): Input Stream Currently Attached to a Data File**

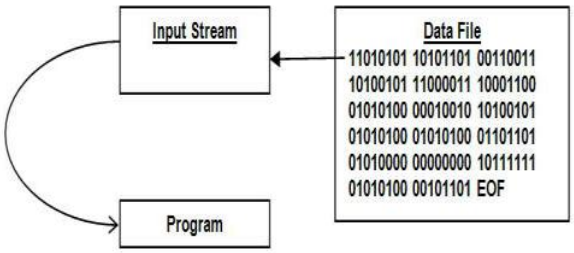
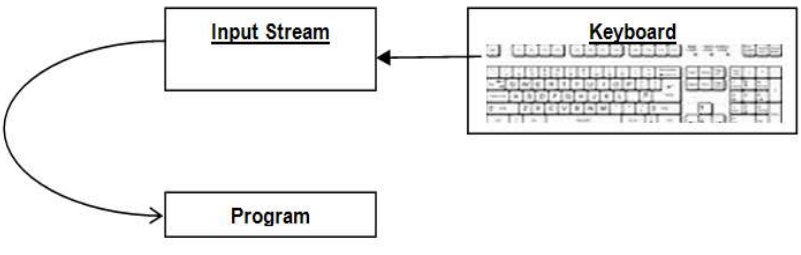



**Figure 13.1 (b): Input Stream Currently attached to Keyboard**



Notes

Figure 13.1 (c): Output Stream Currently Attached to Monitor

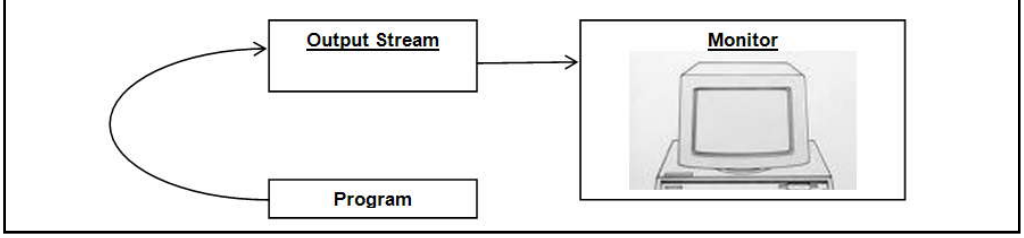



Figure 13.1 (d): Output Stream Currently Attached to a Data File

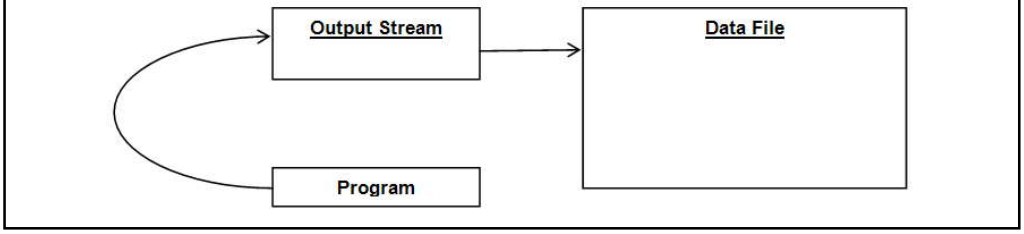
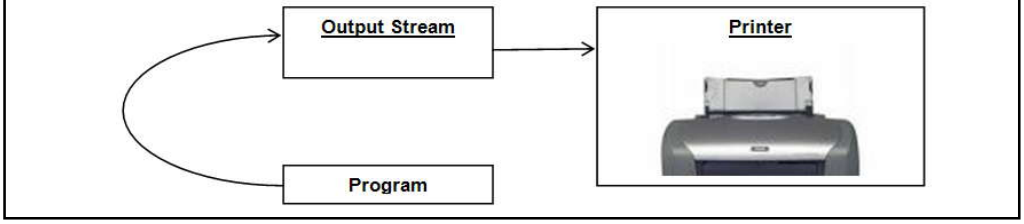



Figure 13.1 (e): Output Stream Currently Attached to a Printer




C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes. Often the array representing a data file is indexed from 0 to len-1, where len is the total number of bytes in the entire file.

A file normally remains in a closed state on a secondary storage device until explicitly opened by a program. A file in open state file has two pointers associated with it:

1. ***Read pointer:*** This pointer always points to the next character in the file which will be read if a read command is issued next. After execution of each read command the read pointer moves to point to the next character in the file being read.
2. ***Write pointer:*** The write pointer indicates the position in the opened file where next character being written will go. After execution of each write command the write pointer moves to the next location in the file being written.

These two file positions are independent, and either one can point anywhere at all in the file. To elucidate I/O streaming in C++ let us write a small program.

```
#include <fstream.h>

int main()

{

       ofstream filename("c:\cppio.dat");
```

Notes

```
        filename << "This text will be saved in the file named cppio.dat in
C:\";

        filename.close();

        return 0;

}
```

When you run this program the text This text will be saved in the file named cppio.dat in C:\ will be saved in a file named cppio.dat in the root directory of the C: drive. Let us run through each line of this program.

#include <fstream.h>

C++ file stream classes and functions have been defined in this file. Therefore, you must include this header file in the beginning of your C++ program so that these classes may be accessed.

ofstream filename ("c:\cppio.dat");

This statement creates an object of class ofstream (acronym for output file stream) named filename. This object acts as the output stream to write data in the specified file. Cppio.dat is the name of the data file in which the program will write its output. If this file does not exists in the specified directory, it is created there.



*Notes* Note that ofstream is a class. So, ofstream filename("c:\cppio.dat"); creates an object from this class.

Here the file name c:\cppio.dat is being passed to the constructor of this class. In short we create an object from class ofstream, and we pass the name of the file we want to create, as an argument to the class' constructor. There are other things, too, that can be passed to the constructor.

filename << "This text will be saved in the file named cppio.dat in C:\";

The << operator is a predefined operator. This line puts the text in the file. As mentioned before, filename is a handle to the opened file stream. So, we write the handle name, << and after it we write the text in inverted commas. If we want to pass variables instead of text in inverted commas, just pass it as a regular use of the cout << as,

filename << variablename;

filename.close();

Having finished with writing into the file the stream must be closed. Filename is an object of class ofstream, and this class has a function close() that closes the stream. Just write the name of the stream object, dot and close(), in order to close the file stream. Note that once you close the file, you can't access it anymore, until you reopen it.

Before we take up the subject any further let us quickly go through a program that reads from a data file and presents the same on the monitor. Here is the program listing.

```
#include <fstream.h>

void main() //the program starts here

{

        ifstream filename("c:\cppio.dat");

        char ch;
```

Notes

```
        while(!filename.eof())
        {
                filename.get(ch);
                cout  <<  ch;
        }
        filename.close();
}
```

Let us run through this program line by line quickly to catch some salient features.

***ifstream filename("c:\cppio.dat")***

Similar to ofstream, ifstream is the input stream for a file. The term input and output have been used with respect to the program. What goes to the program from outside is the input while processed data coming out of the program is the output. We here create an input file stream by the name - filename - to handle the input file stream. The parameter passed to the constructor is the file we wish to read into the program.

***char ch;***

This statement declares a variable of type char to hold one character at a time while reading from the file. In this program we intend to read one character at a time.

***while(!filename.eof())***

The class ifstream has a member function eof() that returns a nonzero value if the end of the file has been reached. This value indicates that there are no more characters in the file to be read further. This function is therefore used in the while loop for stopping condition. The file is read a character at a time till the last character has been successfully read into the program when the while loop terminates.

***filename.get(ch);***

Another member function of the class ifstream is get() which returns the next character to be read from the stream followed by moving the character pointer to the next character in the stream.

***cout << ch;***

The character read in the variable ch is streamed to the standard output device designated by cout (for console output, i.e., monitor) in this statement.

***filename.close();***

Since we reach at this statement when all the characters have been read and processed in the while loop, we are done with the file and hence it is duly closed.

## Self Assessment

Fill in the blanks:

1. The data may be organized in …….. record or may be in free form text.

**Notes**

2. The term …….. have been used with respect to the program.
3. The class ifstream has a member function …… that returns a nonzero value if the end of the file has been reached.
4. The character read in the variable …… is streamed to the standard output device designated by cout.
5. Member function of the …….. is get() which returns the next character to be read from the stream followed by moving the character pointer to the next character in the stream.

## 13.2 Opening a File

The example programs listed above are indeed very simple. A data file can be opened in a program in many ways. These methods are described below.

ifstream filename("filename <with path>"); Or ofstream filename("filename <with path>");

This is the form of opening an input file stream and attaching the file "filename with path" in one single step. This statement accomplishes a number of actions in one go:

1. Creates an input or output file stream
2. Looks for the specified file in the file system
3. Attaches the file to the stream if the specified file is found otherwise returns a NULL value

In case the file has been successfully attached to the stream the pointer is placed at the first position The file stream created is accessible using the object's name. The specified file is searched in the specified directory if a path is included otherwise the file is searched only in the current directory. If the file is not found it is created in case it is being opened for output. While opening a file for output if the specified file is found then it is truncated before opening thereby loosing all there was in the file before. Care should be taken to ensure that the program does not overwrite a file unintentionally. In any case if the file is not found then a NULL value is returned which we can check to ensure that we are not reading a file which was not found. This will cause an error in the program if we attempt to read a file which was not found.

ifstream filename;

filename.open("file name <with path>");

In this approach the input stream - filename - is created but no specific file is attached to the stream just created. Once the stream has been created a file can be attached to the stream using open() member function of the class ifstream or ofstream as is exemplified by the following program snippet which defines a function to read an input file.

```
#include <fstream.h>
void read(ifstream &ifstr)          // file streams can be passed to functions
{
        char ch;
        while(!ifstr.eof())
        {
                ifstr.get(ch);
                cout << ch;
        }
```

Notes

```
        cout << endl << "———" << endl;
}
void main()
{
        ifstream filename("data1.dat");
        read(filename);
        filename.close();
        filename.open("data2.dat");
        read(filename);
        filename.close();
}
```



*Notes* Note how the same input file stream can be attached to different files at different times.

ifstream filename(char *fname, int open_mode);

In this form the ifstream constructor takes two parameters - a filename and another the mode in which the input file would be read. C++ offers a host of different opening modes for the input file each offering different types of reading control over the opened file. The file opening modes have been implemented in C++ as enumerated type called ios. The various file opening modes are listed below.

| Opening Mode | Description |
|---|---|
| ios::in | Open file in input mode for reading |
| ios::out | Open file in output mode for writing |
| ios::app | Open file in output mode for writing the new content at the end of the file without removing the previous contents of the file. |
| ios::ate | Open file in output mode for writing the new content at the current position of the pointer without removing the previous contents of the file. |
| ios::trunc | Open file in output mode for writing the new content at the beginning of the file removing the previous contents of the file. |
| ios::nocreate | The file is not created. The operation takes place on existing file. If the file is not found an error occurs. |
| ios::noreplace | The existing file is not overwritten. The operation takes place on existing file. If the file is not found an error occurs. |
| ios::binary | Opens the file in binary mode reading not a character but reading/writing whatever binary value is stored in the file. |

Note that all these values are int constants from an enumerated type ios. The following program demonstrates the usages of a file opening mode.

```
#include <fstream.h>
void main()
{
```

**Notes**

```
        ofstream myfile("data1.dat", ios::ate);
        myfile << "Save this text to the file";
        myfile.close();
}
```

This program will write Save this text to the file at the end of the file data1.dat without removing the previous content of the file.

A file can also be opened in mixed mode using OR (|) operator to combine the modes. For example,

ios::ate | ios::binary

opens the file in binary mode for output at the current pointer position without removing the previous contents of the file. Another example of mixed mode is given below.

fstream myfile("data.dat",ios::in | ios::out);

Input file stream (ifstream) allows only reading from the file and output file stream (ofstream) only for writing into the file. If you wish to open a file both for writing and reading at the same time you should use file stream (fstream) with specific opening mode as the following program demonstrates.

```
#include <fstream.h>
void main()
{
        fstream myfile ("data.dat",ios::in | ios::out);
        myfile << "Write this text";       //Writing into the file
        static char TextRead[50];          //Create an array to hold what is
                                           read from the file
        myfile.seekg(ios::beg);            //Get back to the beginning of
                                           the file explained later
        myfile >> TextRead;
        cout << TeaxtRaed << endl;
        myfile.close();
}
```

In this program the statement fstream myfile ("data.dat",ios::in | ios::out); creates an object from class fstream. At the time of execution, the program opens the file data.dat in read/write mode (ios::in | ios::out) which allows to read from the file, and put data into it, at the same time. This is done by allowing the program to move the character pointer in the file to a desired location before reading or writing.

The read and write operation takes place on the current position of the pointer in the file. Therefore you might need to shift this position to a desired location. The member function seekg() of the class fstream allows you to do just that as shown below.

myfile.seekg(ios::beg) //Take the pointer to the beginning of the file

myfile.seekg(ios::end) //Take the pointer to the end of the file

**Notes**

myfile.seekg(10) //Take the pointer to 10 characters after the current location

myfile.seekg(-10) //Take the pointer to 10 characters before the current location

Moreover, the function seekg()is overloaded to take two parameters. Thus, another form of the function is,

myfile.seekg(-4, ios::end) //Take the pointer 4 characters before the end of file

Most of the time it is necessary to check whether a specified file exists or not while opening it. Opening of file may also fail due to other reasons. Remember that if the file opening fails the open() function return a non-zero value. By checking the return type of the function one can ensure whether the file was opened successfully or not as demonstrated in the following program snippet.

```
fstream myfile("data.dat");
if (!myfile)
{
cout << "Error : File could not be opened\n";
exit(1);
}
```

The ofstream class also provides a fail() function which return true if the opening of file operation failed as shown in the following code snippet

```
ofstream myfile("data.dat", ios::nocreate);
if(myfile.fail())
{
cout << "Error : File could not be opened\n";
exit(1);
}
```

binary files are unformatted and uninterpreted file of binary digits. It simply contains binary numbers whose meaning is provided by the program that manipulates the file contents. The functions that give you the possibility to write/read unformatted files are get() and put(). To read a byte, you can use get() and to write a byte, use put(). Both get() and put() functions take one parameter - a char variable or character.

If you want to read/write whole blocks of data, then you can use the read() and write() functions. Their prototypes are:

istream &read(char *buf, streamsize num);

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

For the read() function, but should be an array of chars, where the read block of data will be put. For the write() function, buf is an array of chars, where is the data you want to save in the file. For the both functions, num is a number, that defines the amount of data (in symbols) to be read/written.

Another function that provides the number of symbols read so far - gcount(). It simply function returns the number of read symbols for the last unformatted input operation. You can specify that the file is going to be operated on in binary mode in the open() function. The required mode is ios::binary.

**Notes**

Let us write a program that uses get() and put() functions for binary files.

```
#include <fstream.h>
void main()
{
        fstream myfile("data.dat",ios::out | ios::in | ios::binary);
                                        //open file in binary mode
        char ch;
        ch='A';
        myfile.put(ch);                 //put the content of ch to the file
        myfile.seekg(ios::beg);         //go to the beginning of the file
        myfile.get(ch);                 //read one character
        cout << ch << endl;             //display it on console
        myfile.close();
}
```

The following program uses read() and write() functions for binary file manipulations.

```
#include <fstream.h>
#include <string.h>
void main()
{
        fstream myfile("data.dat",ios::out | ios::in | ios::binary);
        char Carray[13];
        strcpy(Carray,"C++ Programming"); //put the string into the
                                          character array

        myfile.write(Carray,5);         //put the first 5 characters "C++ P"
                                        into the file
        myfile.seekg(ios::beg);         //go to the beginning of the file
        static char Rarray[10];         //To store read data
        myfile.read(Rarray,3);          //read the first 3 characters- "C++"
        cout << Rarray << endl;         //display them on console
        myfile.close();                 //close file
}
```

*Task*

1. A file in open state file has two pointers associated with it. What are they?
2. Please quote two methods of opening a data file in a program.

### Self Assessment

Notes

Fill in the blanks:

6. In case the file has been successfully attached to the stream the pointer is placed at the …… position.
7. Care should be taken to ensure that the program does not …….. a file unintentionally.
8. …… files are unformatted and uninterpreted file of binary digits.
9. Function that provides the number of symbols read - …….. .

## 13.3 Reading/Writing a Character from/into a File

Reading and writing a character in a data file has been dealt with in the previous sections in detail. The general procedure of reading a file one character at a time is listed below:

1. Create an input file stream from <fstream.h> header file:

   ifstream name_of_input_stream;

2. Open the data file by passing the file name (optionally full name) to this input stream:

   name_of_input_stream.open("data.dat");

   Both the above statements can be combined in the following:

   ifstream name_of_input_stream("data.dat");

3. Set up a character type variable to hold the read character.

   char ch;

4. Read a character from the opened file using get() function:

   name_of_input_stream.get(ch);

   This way you can read the entire file in a loop stopping condition of the loop being the end of file:

```
while(!filename.eof())
{
        name_of_input_stream.get(ch);
        //process the read character
}
```

5. When finished close the file using close() function:

   name_of_input_stream.close();

   The if stream class is defined in fstream.h header file. Therefore you must include this file in your program. The complete program is listed below.

```
//reading a file one character at a time
#include <fstream.h>
void main() //the program starts here
{
```

Notes

```
        ifstream filename("c:\cppio.dat");
        char ch;
        while(!filename.eof())
        {
                filename.get(ch);
                cout << ch;
        }
        filename.close();
}
```



*Did u know?* **What do you need to declare to store the date read?**

You need to declare a character array to store the data read when reading from a file. The character array can be any size as long as it is big enough to store what you are reading in.

The general procedure of writing one character at atime in a file is listed below:

1. Create an output file stream from <fstream.h> header file:

   ofstream name_of_output_stream;

2. Open the data file by passing the file name (optionally full name) to this output stream:

   name_of_output_stream.open("data.dat");

   Both the above statements can be combined in the following:

   ofstream name_of_output_stream("data.dat");

3. Write a character in the opened file using << operator:

   name_of_output_stream << 'A';

4. When finished close the file using close() function:

   name_of_output_stream.close();

The ofstream class is defined in fstream.h header file. Therefore you must include this file in your program. The complete program is listed below.

```
//Writing a character in a data file
#include <fstream.h>
int main()
{
        ofstream filename("data.dat");
        filename << 'A';
        filename.close();
        return 0;
}
```

Notes

### Some useful File Operation Functions

In addition to the function discussed thus far, there are many more functions which come handy in writing practical programs on data file processing. An assorted list of them is given below:

#### *tellg()*

This function returns an int type value representing the current position of the pointer inside the data file opened in input mode as demonstrated in the following program.

```
//Program demonstrating tellg() function
#include <fstream.h>
void main()
{
        //Assume that the text stored in data.dat file is "Welcome to C++"
        ifstream myfile("data.dat");
        char ch;
        for(int j=0; j<4;j++)
        myfile.get(ch);
        cout <<myfile.tellg() << endl;
        //this should return 5, as four characters have been read from the
file so the current pointer points at next
        //character, i.e., 5th characterHello is 5 characters long
        myfile.close();
}
```

#### *tellp()*

This function does to output files what tellg() does to input files. It returns an int type value representing the current position of the pointer inside the data file opened in output mode as shown in the following code snippet.

```
        ofstream myfile("data.dat");
        myfile<<"India";
        cout << myfile.tellp();
        //this should return 6, as five characters have been written in the
file so the current pointer points at next
        //position, i.e., 6th
        myfile.close();
```

#### *seekg()*

While reading data from a file this function is used to shift the pointer to a specified location as shown in the following code snippet.

```
        //Assume that the text stored in data.dat file is "Welcome to C++"
        ifstream myfile("data.dat");
        char ch;
```

Notes

```
        for(int  j=0;  j<4;j++)
        myfile.get(ch);
        myfile.seekg(-2);
        //this  will  take  the  pointer  to  2  characters  before  the  current
location.  It  will  now  point  to  (l)
```

### *seekp()*

This function does to output files what seekp() does to input files. It shifts the pointer to the specified location in the output file. If you want to overwrite the last 5 characters, you will have to go back 5 characters from the current pointer position. This you can do by the following statement.

myfile.seekp(-5);

### *ignore()*

This function is used when reading a file to ignore certain number of characters. You can use seekg() as well for this purpose just to move the pointer up in the file. However, ignore() function has one advantage over seekg() function. The prototype is of ignore() function is given below.

fstream& ignore( int, char);

Caution

You can specify a delimiter character in ignore() function whence it ignores all the characters up to the first occurrence of the specified delimiter.

Where int is the count of characters to be ignored and delimiter is the character up to which you would like to ignore as demonstrated in the following program.

```
//demonstration  of  ignore()  function
#include  <fstream.h>
void  main()
{
              //Assume  that  the  text  contained  in  data.dat  file  is  "Welcome
to  C++"
              ifstream  myfile("data.dat");
              static  char  Carray[10];
              //go  on  ignoring  all  the  characters  in  the  input  up  to  10th
character  unless  an                'm'  is  found
              myfile.ignore(10,'m');
              myfile.read(Carray,10);
              cout  <<  Carray  <<  endl;     //it  should  display  "me  to  C++"
              myfile.close();
}
```

Notes

### *getline()*

This function is used to read one line at a time from an input stream until some specified criterion is met. The prototype is as follows:

getline(Array,Array_size,delimiter);

The stopping criterion can be either specified number of characters (Array_size) or the first occurrence of a delimiter (delimiter) else the entire line (up to newline character '\n') is read. If you wish to stop reading until one of the following happens:

1. You have read 10 characters
2. You met the letter 'm'
3. There is new line

Then the function will be called as follows:

getline(Carray,10,'m');

The use of getline() function is demonstrated in the following example.

```
//Demonstration of getline() function to read a file line-wise
#include <fstream.h>
void main()
{
       //Assume that the text contained in data.dat file is "Welcome to
C++"
       ifstream myfile("data.dat");
       static char Carray[10];
       myfile.getline(Carray,10,'m');
       cout << Carray << endl;     //the output should be "Welco"
       myfile.close();
}
```

### *peek()*

This function returns the ASCII code of the current character from an input file stream very much like get() function, however, without moving the pointer to the next character. Therefore, any number of successive call to peek() function will return the ASCII code of same character each time. To convert the ASCII code (as returned by peek() function use char type cast) as demonstrated in the following code program.

```
//Demonstration of peek() function
#include <fstream.h>
void main()
{
       // Assume that the text contained in data.dat file is "Welcome to
C++"
       ifstream myfile("data.dat");
```

Notes

```
        char ch;
        myfile.get(ch);
        cout << ch << endl;          //the output should be 'W' and the pointer
                                     will move to point 'e'
        cout <<       char(myfile.peek()) << endl; //should display "e"
        cout <<       char(myfile.peek()) << endl; //should display "e" again
        cout <<       myfile.peek() << endl;      //should display 101
        myfile.get(ch);
        cout << ch << endl;          //will display "e" again and move the
                                     pointer to 'l'
        myfile.close();

}
```

### *putback()*

This function returns the last read character, and moves the pointer back. In other words, if you use get() to read a char and move the pointer to next character, then use putback(), it will show you the same character, but it will set the pointer to previous character, so the next time you call get() again, it will again show you the same character as shown in the following program.

```
//Program demonstrating use of putback() function
#include <fstream.h>
void main()
{
       // Assume that the text contained in data.dat file is "Welcome to
C++"
       ifstream myfile("data.dat");
       char ch;
       myfile.get(ch);
       cout << ch << endl;          //output will be 'W'
       myfile.putback(ch);
       cout << ch << endl;          //output will again be 'W'
       myfile.get(ch);
       cout << ch << endl;          // output will again be 'W'
       myfile.close();
}
```

### *flush()*

I/O streams are created and maintained in the RAM. Therefore, when dealing with the output file stream, the data is not saved in the file as the program enters them. A buffer in the memory holds the data until the time you close the file or the buffer is full. When you close the file the

Notes

data is actually saved in the designated file on the disk. Once the data has been written to the disk the buffer becomes empty again.

In case you want to force the data be saved even though the buffer is not full without closing the file you can use the flush() function. A call to flush() function forces the data held in the buffer to be saved in the file on the disk and get the buffer empty.

### Self Assessment

Fill in the blanks:

10. ……. returns the ASCII code of the current character from an input file stream.
11. …….. are created and maintained in the RAM.
12. …….. is used to read one line at a time from an input stream until some specified criterion is met.

## 13.4 Appending

Inserting data somewhere in a sequential file would require that the entire file be rewritten. It is possible, however, to add data to the end of a file without rewriting the file. Adding data to the end of an existing file is called appending.



*Did u know?* **What happens when you open a file?**

Files are cleared by default when you open them.

```
fout.open("filename.dat", ios::app)   //open file for appending


//add names and ages to an existing file
#include <iostream.h>
#include <fstream.h>
```

If the file you open for appending does not exist, the operating system creates one just as if you had opened it using ios::out mode.

```
int main(void)
{
       apstring name, dummy;
       int number, i, age;
       ofstream fout;

       cout<<"How many names do you want to add?";
       cin>>number;
       getline(cin,dummy);
```

Notes

```
        fout.open("name_age.dat",ios::app);      // open file for appending
        assert (!fout.fail( ));

        for(i=1; i<=number; i++)
        {
            cout<<"Enter the name:  ";
            getline(cin,name);
            cout<<"Enter age:  ";
            cin>>age;
            getline(cin,age);

            fout<<name<<endl;    //send to file
            fout<<age<<endl;
        }
        fout.close( );         //close file
        assert(!fout.fail( ));
        return 0;
}
```

## 13.5 Processing and Closing a File

File processing in C++ is performed using the fstream class. Unlike the FILE structure, fstream is a complete C++ class with constructors, a destructor and overloaded operators.

To perform file processing, you can declare an instance of an fstream object. If you do not yet know the name of the file you want to process, you can use the default constructor.



*Notes* Unlike the FILE structure, the fstream class provides two distinct classes for file processing. One is used to write to a file and the other is used to read from a file.

This is an example of performing file processing in C++. The following technique can be used to save complete but separate lines of text using the "ws" feature on Microsoft Windows. This (undocumented) feature is built-in in the operating system and works on both Microsoft C++ and Borland C++ compilers. I didn't test this program on Linux (my Linux computer was not available) but it is not likely to work on that operating system because, as stated already, cin >> ws which is used to "freely" eat a keyboard stroke is part of MS Windows.

Microsoft Visual C++ Version

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

**Notes**

```
using namespace std;
int main()
{
      char FileName[20];
       char EmployeeName[40], Address[50], City[20], State[32], ZIPCode[10];

      // 1. Uncomment the following section to create a new file
/*
       cout << "Enter the Following pieces of information\n";
      cout << "Empl Name: "; cin >> ws;
       cin.getline(EmployeeName, 40);
      cout << "Address:    "; cin >> ws;
       cin.getline(Address, 50);
      cout << "City:       "; cin >> ws;
       cin.getline(City, 20);
      cout << "State:      "; cin >> ws;
       cin.getline(State, 32);
      cout << "ZIP Code:   "; cin >> ws;
       cin.getline(ZIPCode, 10);
      cout << "\nEnter the name of the file you want to create: ";
      cin >> FileName;
       ofstream EmplRecords(FileName, ios::out);
      EmplRecords << EmployeeName << "\n" << Address << "\n" << City << "\n"
<< State << "\n" << ZIPCode;
*/
      // 2. Uncomment the following section to open an existing file
/*
      cout << "Enter the name of the file you want to open: ";
      cin >> FileName;
       ifstream EmplRecords(FileName);
       EmplRecords.getline(EmployeeName, 40, '\n');
       EmplRecords.getline(Address, 50);
       EmplRecords.getline(City, 20);
       EmplRecords.getline(State, 32);
       EmplRecords.getline(ZIPCode, 10);

      cout << "\n -=- Employee Information -=-";
```

Notes

```
        cout << "\nEmpl Name: " << EmployeeName;
        cout << "\nAddress:    " << Address;
        cout << "\nCity:       " << City;
        cout << "\nState:      " << State;
        cout << "\nZIP Code:   " << ZIPCode;
*/
        cout << "\n\n";
        return 0;
}
```

Closing an opened data file is the simplest of all the data file operations. All you have to do is to call the member function close() on the file stream hooked to the opened file. Once the file is closed it cannot be read/written unless reopened. The syntax for the same action is,

File_stream_name.close();

### Self Assessment

Fill in the blanks:

13. To perform file processing, you can declare an instance of an …….. object.
14. If you want to add things to an existing file then you must open it for appending by using …… when opening the file.

## 13.6 Different Types of Files

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file. Whether you copied a file from elsewhere or created your own, you will need to return to it later in order to edit its contents.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sector, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called "clusters" or "blocks" which contain a certain number of disk sectors (usually 1-64). This is the smallest logical amount of disk space that can be allocated to hold a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g, from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

Notes

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's meta-data was changed. (Note that many early PC operating systems did not keep track of file times.) Other information can include the file's device type (e.g., block, character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts. For example, interprocess pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.

Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data.

More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.



*Task* File systems share much in common with database technology, but it is debatable whether a file system can be classified as a special-purpose database (DBMS). Analyze.

### 13.6.1 Types of File Systems

File system types can be classified into disk file systems, network file systems and special purpose file systems.

1. ***Disk file systems:*** A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.

Notes

2. ***Flash file systems:*** A flash file system is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives.

   While a block device layer can emulate a disk drive so that a disk file system can be used on a flash device, this is suboptimal for several reasons:

   (a) *Erasing blocks:* Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.

   (b) *Random access:* Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.

   (c) *Wear leveling:* Flash memory devices tend to wear out when a single block is repeatedly overwritten; flash file systems are designed to spread out writes evenly.

   Log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

3. ***Database file systems:*** A new concept for file management is the concept of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Example: dbfs.

4. ***Transactional file systems:*** Each disk operation may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

   Transaction processing introduces the guarantee that at any point while it is running, a transaction can either be finished completely or reverted completely (though not necessarily both at any given point). This means that if there is a crash or power failure, after recovery, the stored state will be consistent. (Either the money will be transferred or it will not be transferred, but it won't ever go missing "in transit".)

   This type of file system is designed to be fault tolerant, but may incur additional overhead to do so.

   Journaling file systems are one technique used to introduce transaction-level consistency to file system structures.

5. ***Network file systems:*** A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Examples of network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

6. ***Special purpose file systems:*** A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space.

   Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the procfs (/proc) file system used by some Unix variants, which grants access to information about processes and other operating system features.

Notes

Deep space science exploration craft, like Voyager I & II used digital tape based special file systems. Most modern space exploration craft like Cassini-Huygens used Real-time operating system file systems or RTOS influenced file systems. The Mars Rovers are one such example of an RTOS file system, important in this case because they are implemented in flash memory.

7. ***Flat file systems:*** In a flat file system, there are no subdirectories-everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc. While simple, this system rapidly becomes inefficient as the number of files grows, and makes it difficult for users to organise data into related groups. Like many small systems before it, the original Apple Macintosh featured a flat file system, called Macintosh File System. Its version of Mac OS was unusual in that the file management software (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of MFS. This structure meant that every file on a disk had to have a unique name, even if it appeared to be in a separate folder. MFS was quickly replaced with Hierarchical File System, which supported real directories.

### 13.6.2 File Systems and Operating Systems

Most operating systems provide a file system, as a file system is an integral part of any modern operating system. Early microcomputer operating systems' only real task was file management - a fact reflected in their names. Some early operating systems had a separate component for handling file systems which was called a disk operating system. On some microcomputers, the disk operating system was loaded separately from the rest of the operating system. On early operating systems, there was usually support for only one, native, unnamed file system; for example, CP/M supports only its own file system, which might be called "CP/M file system" if needed, but which didn't bear any official name at all.

Because of this, there needs to be an interface provided by the operating system software between the user and the file system. This interface can be textual (such as provided by a command line interface, such as the Unix shell, or OpenVMS DCL) or graphical (such as provided by a graphical user interface, such as file browsers).



*Caution* If graphical, the metaphor of the folder, containing documents, other files, and nested folders is often used.

### 13.6.3 Binary Files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions likegetline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

write ( memory_block, size );

read ( memory_block, size );

Where memory_block is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written

Notes

are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.



*Example:*

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
   ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
   if (file.is_open())
  {
      size = file.tellg();
      memblock = new char [size];
      file.seekg (0, ios::beg);
      file.read (memblock, size);
      file.close();

      cout << "the complete file content is in memory";

      delete[] memblock;
  }
   else cout << "Unable to open file";
   return 0;
}      the complete file content is in memory
```

In this example the entire file is read and stored in a memory block. Let's examine how this is done:

First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file. Notice the type we have used to declare variable size:

ifstream::pos_type size;

ifstream::pos_type is a specific type used for buffer and file positioning and is the type returned by file.tellg(). This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another

Notes

integer type large enough to contain the size of the file. For a file with a size under 2GB we could use int:

```
int size;
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

### Self Assessment

Fill in the blanks:

15. If you want to enter your own text or data, you will start by …….. a file.
16. File streams include two member functions specifically designed to input and output binary data sequentially: ……... .

## 13.7 Command Line Arguments

The main function may be defined not to have any parameter. In some cases, though, the program is provided with some input values at the time of execution. These values are known as command line parameters. If the main function must process the command line parameters, it should be defined as having two parameters – argc of int type and argv of pointer to character type array.

argc is initialized with a number of parameters provided in command line while argv points to the list of parameters itself, as is illustrated through the code snippet listed below:

```
#include<iostream.h>
void main(int argc, char *argv[])
{
        for (int i=0; i<argc;i++)
                cout<<'\n'<<argv[i];
}
```

Let us assume that the name of the program is abc.exe. If you execute this program from the command line as shown below:

C:\>abc delhi agra kolkata

Notes

Then the output of the program would be as follows:

C:\>abc

delhi

agra

kolkata

As you must have noticed the parameter argc is initialized with the number of command line arguments plus one. These command lines are stored in argv[1], argv[2]… The name of the program is itself stored in argv[0]. It is up to the programmer to posses these command line arguments the way they want.



*Notes* Please note that the command line arguments are always read as string of characters even if a numeric value is passed as argument.

Another important fact to be noted is the argument names in the main function need not be argc and argv alone. They can have any valid name as long as the type are int and char * respectively as illustrated in the following code snippet.

```
#include <iostream.h>
void main(int a, char *ar[])
{
        for (int i=0; i<a;i++)
                cout<<'\n'<<ar[i];
}
```

## Self Assessment

Fill in the blanks:

17. …….. is initialized with a number of parameters provided in command line.



*Caselet*

### Hardware and Software of Innovation

There is much that is dysfunctional in the US, but there is also a great deal to be optimistic about, writes Adam Segal in Advantage: How American innovation can overcome the Asian challenge (www.wwnorton.com). Noting that across the US, numerous regions, companies, and universities are experimenting with new ways to promote and structure innovation, launching bottom-up efforts to create collaborative communities, he adds that these efforts are grounded in the country's comparative advantage — 'an open and flexible culture and a web of institutions, attitudes, and relations that move ideas from the lab to the marketplace.'

*Contd...*

Notes

The author concedes that more science and scientific discovery will occur outside the US, in new government and university labs in China and India, and in the corporate labs of Japanese and South Korean companies. "While we have grown accustomed to science flowing west across the Pacific, our true shift in consciousness will not be realised until we internalise how much we can learn and gain from collaboration with Asia by meshing our software advantage with Asia's emerging hardware strengths."

**Source Code**

A section on 'the hardware and software of innovation' avers that Asia is underdeveloped in the software aspect, which includes both the specific organisations and relationships that structure innovation and the underlying cultural framework, the 'source code.'

A dismal story narrated in the section is of Chen Jin, the dean of the School of Microelectronics at Shanghai Jiaotong University, who became a national hero in 2004 for his work on the Hanxin chip, China's first digital signal processing computer chip. "The chip, which can be used in modems, cellular phones, high-capacity hard disks, digital cameras, and digital TVs, is critical to China's drive to become the preeminent player in information technology markets… The Chinese press praised Chen as a patriot, particularly since he had left a good job at Motorola to return to China."

The Ministry of Education made Chen a 'Yangtze River Scholar,' the highest academic award given by the Government of China, and the government support for his research totalled more than 100 million yuan, one learns.

**Pressures on Scientists**

Alas, the chip was a fake; for, when Chen left Motorola, he had taken a chip with him, and then scratched the name off it to stamp Hanxin thereon, the book recounts. "Until an assistant exposed him, Chen used connections of various universities and bribed government officials to receive fake certifications of design and testing. After the fraud was revealed, the university removed Chen, and he was required to return the investment funds."

The author observes that it is not easy to be a scientific star in China. He notes that celebrity professors like Chen face extraordinary pressure to produce tangible outcomes, after having been lured home with promises of cutting-edge equipment and brand-new labs staffed by eager graduate students, showered with attention by the media, and feted by a government that desperately wants its own technology to compete with Western standards. "A scientist who is unable to come up with the goods might be tempted to plagiarise or falsify research results… Fraud and plagiarism are prevalent because of a lack of accountability and effective oversight in Chinese society."

**Culture of Collaboration**

Instructs Segal, therefore, that regardless of how fervently China races to build the hardware of innovation, we should not mistake the inputs to the innovation process for actual innovation. An insightful quote of Cheng Jing, CEO of Beijing biotech company Capital Biochip, reads thus: "To construct a research building takes a year. To fill it with something really meaningful easily takes ten to twenty years."

The author explains that a country can build labs, invest money, enrol students, and recruit prominent professors; yet, these steps will not produce the intended results when there is no respect for the rule of law and intellectual property rights, as well as a culture of individual initiative and openness.

*Contd...*

Notes

In contrast to the approach adopted by many countries, the US model, as Segal describes, has the private sector as the main engine of technological growth, funding more than two-thirds of research and development, while the federal government funds most basic research. He extols the culture of working closely together generally among academia, industry, and government, despite 'stove-piping (the failure to share information and ideas across organisational boundaries), and turf battles.'

**Multidisciplinary Research**

An example of such collaborative research project mentioned in the book is Bio-X, a massive multidisciplinary research programme in Stanford University working at the intersection of medicine, science, and engineering. "People and ideas circulate freely, through informal gatherings and the planned meetings that Bio-X hosts – cocktail and coffee hours where bright graduate students can make pitches to the venture capital firms clustered on Sand Hill road in Menlo Park."

Elaborates Segal that what is critical beyond the free flow of ideas is the existence of strong incentives to move inventions from the lab to the market. In the US, ideas can make one rich, because intellectual property is protected and individual scientists are able to exploit their breakthroughs for commercial gain, he informs. "The young entrepreneur has many role models to emulate: the Sergey Brin, Steve Jobs, and others who demonstrate the massive rewards that come to those who execute good ideas well."

Underlining the risk-embracing culture among scientists and entrepreneurs, Segal speaks of how failure is seen as a badge of honour, an entrepreneurial rite of passage; and about how invention and innovation are locally driven. "Yes, the federal government was the driving force for large-scale projects such as ARPANET (Advanced Research Projects Agency Network), the predecessor to the Internet, but the tradition of the individual tinker and the culture of making things in the backyard with a group of like-minded friends remain strong."

**Brain Circulation**

It should be heartening to read the portrayal of India as not being Delhi-driven but seeing action in many public-private partnerships, such as in the form of IT giants training thousands of computer science graduates annually, and their working with local colleges to develop relevant courses for engineers.

Segal also finds that technology entrepreneurs and returnees to be especially important in building a culture of innovation. He cites the Nasscom's statistics that between 2001 and 2007, 35,000 IT professionals returned to India; and the findings of a survey of Indian executives living in the US that 68 per cent were actively looking for an opportunity to return home, and 12 per cent had already decided to do so.

This is 'brain circulation,' the way the Berkeley scholar AnnaLee Saxenian calls the flow or returnees from Silicon Valley to China and India, writes Segal. These individuals, he says, no longer represent 'brain drain' and a loss to their home countries, but neither are they a clear-cut 'brain gain' since they often retain business and personal connections to the US.

Examples of the 'new argonauts' (Saxenian's phrase for those travelling between two worlds) that Segal lists are Rosen Sharma, with degrees from IIT Delhi and Cornell University who lives in California and travels to New Delhi and Pune to oversee local employees of Solidcore, a developer of security software; and Rajiv Mody, who founded Sasken Communication Technologies in Silicon Valley and moved the company to Bangalore, and now travels back to the US and pays US taxes.

*Contd...*

Notes

After studying the Indian IT sector and also the prevailing politics here, the author grimly predicts that the sector is likely to remain divorced from the rest of Indian society and be focused on export markets, thus reinforcing much of the inequality in Indian society, which in turn feeds back into Indian domestic politics and maintains the status quo. He, however, feels that a virtuous cycle can possibly see a strong reform programme creating the conditions for a broad-based innovation system.

Recommended read.

dmurali@thehindu.co.in

Tailpiece

"After the management decided to ban all social networking within the office, we found a dramatic increase in…"

"Productivity?"

"No, absenteeism"

## 13.8 Summary

- Fields in C++ are interpreted as a sequence of or stream of bytes stored on some storage media'. Member functions of these or base classes are used to perform I/O operations.
- The read( ) and write( ) functions work in binary mode. The ifstream class is used for input, ofstream for output and itstream for both input and output.
- A data of a file is stored in the form of readable and printable characters then the file is known as text file. A file contains non-readable characters in binary code then the file is called binary file.
- The function get( ) read and write data respectively. The read( ) and write( ) function read and write block of binary data. The close( ) function close the stream.
- The eof( ) functions determines end-of-file by returning true otherwise false.
- C++ treats each source of input and output uniformly. The abstraction of a data source and data sink is what is termed as stream. A stream is a data abstraction for input/output of data to and fro the program.
- C++ library provides prefabricated classes for data streaming activities. In C++, the file stream classes are designed with the idea that a file should simply be viewed as a stream or array or sequence of bytes.
- A file normally remains in a closed state on a secondary storage device until explicitly opened by a program.
- The << operator is a predefined operator. This line puts the text in the file or an output stream.
- C++ offers a host of different opening modes for the input file each offering different types of reading control over the opened file. The file opening modes have been implemented in C++ as enumerated type called ios.
- During a program execution structures are not created on a disk rather they are created in the memo.
- Binary files provide a better way of storing structures into a data file on the disk using read() and write() functions. Classes are also created in the memory just like structures and hence are lost at the termination of the program that created them.

Notes

## 13.9 Keywords

***Command Line Parameters:*** The main functions may be defined not to have any parameter. In some cases, though, the program is provided with some input values at the line of execution. These values are known as command line parameter.

***EOF:*** End of file.

***EOL:*** End of line.

***File:*** A storage unit that contains data. A file can be stored either on tape or disk.

***Input Stream:*** The stream that suppliers date to the program is known as input stream.

***Output Stream:*** The stream that receives data from the program is known as output stream.

***Stream:*** A stream is a general name given to a flow of data.

## 13.10 Review Questions

1. How is C++ able to treat all the input and output operation uniformly?
2. Develop a simple C++ class that provides rudimentary functionalities of a Database management system including:
   (a) Creating a table
   (b) Record insertion
   (c) Record updation
   (d) Record deletion
   (e) Report generation
3. Programs would not be very useful if they cannot input and/or output data from/to users. Explain.
4. Write a note on classes for file stream operations.
5. How will you open and close a file in C++?
6. Write a note on read() and write() functions.
7. Describe various file operations applicable on array of class objects.
8. Inserting data somewhere in a sequential file would require that the entire file be rewritten. Analyze.
9. A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Justify your answers with an example.
10. How can you achieve random access in C++?
11. Write an interactive menu driven program to create and read a text file.
12. Write an interactive menu driven program to create a data file, read a data file, append records, insert records and modify records. The data file has following fields: name of student, year of admission, class and section.
13. Write a program in C++ that creates a text file, which is an exact copy of a given file.

### Answers: Self Assessment

Notes

1. fixed size
2. input and output
3. eof()
4. ch
5. class ifstream
6. first
7. overwrite
8. Binary
9. gcount()
10. peek ()
11. I/O streams
12. getline ()
13. Fstream
14. ios::app
15. creating
16. write and read
17. argc

## 13.11 Further Readings



*Books*

E. Balagurusamy, *Object-oriented Programming through C++*, Tata McGraw Hill.

Herbert Schildt, *The complete Reference-C++*, Tata Mc Graw Hill.

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.



*Online links*

http://www.learn-programming.za.net/programming_cpp_learn09.html

http://www.cplusplus.com/reference/clibrary/cstdio/FILE/

**Notes**

# Unit 14: Advanced Concept in C++

**CONTENTS**

Objectives

Introduction

14.1 Function of Templates

14.2 Classes of Template

14.3 The Typename Keyword

14.4 Template Specialization

14.5 Point of Instantiation

14.6 Error Handling

14.7 Error Isolation

14.8 Concurrent Object-oriented Systems

14.9 Summary

14.10 Keywords

14.11 Review Questions

14.12 Further Readings

## Objectives

After studying this unit, you will be able to:

- Describe the function of templates
- Describe the classes of templates
- Describe the templates specialization
- Explain the Error handling and error isolation
- Describe watch values, break point and stepping

## Introduction

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class with several different types of data without having to explicity recode specific versions for each data type.

## 14.1 Function of Templates

Notes

In C ++ when a function is overloaded, many copies of it have to be created, one for each data type it acts on. In the example of the max ( ) function, which returns the greater of the two values passed to it this function would have to be coded for every data type being used. Thus, you will end up coding the same function for each of the types, like int, float, char, and double. A few versions of max ( ) are:

```
Int max ( int x, int y)
{
        return x  > y ?   x : y
}
        char max ( char x, char y )
{
        return x > y  ? > x : y
}
        double max (double x , double y)
{
        return x > y ? x : y
}
        float max ( float x, float y)
{
        return x > y ? x : y
}
```

Here you can see, the body of each version of the function is identical. The same code has to be repeated to carry out the same function on different data types. This is a waste of time and effort, which can be avoided using the template utility provided by C ++.

"A template function may be defined as an unbounded functions " all the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary. Template functions are using the keyword, template. Templates are blueprints of a function that can be applied to different data types.



*Notes* The definition of the template begins with the template keyword. This is followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.

Syntax of Template

```
Template < class type 1, type 2 ... >
Void function - name ( type 2 parameter 1, type 1 parameter 2 ) {...}
```

**Notes**



*Example:*

```
Template < class   X >
X min ( X a , X b )
{
return (a < b ) ? a : b ;
}
```

This list of parameter types is called the formal parameter list of the template, and it cannot be empty. Each formal parameter consists of the keyword, type name, followed by an identifier. The identifier can be built-in or user-defined data type, or the identifier type. When the function is invoked with actual parameters, the identifier type is substituted with the actual type of the parameter. This allows the use of any data type. The template declaration immediately precedes the definition of the function for which the template is being defined. The template declaration, followed by the function definition, constitutes the template definition. The template of the max ( ) function is coded below:

```
# include   < iostream.h >
template < class type >
type max (  type x , type y)
{
            return x > y ? x : y ;
}
int main ( )
{
cout   << " max ( 'A', 'a') : " << max ( 'A', 'a')   << endl ;
cout   <<   " max ( 30, 40 ) : " << max ( 30 , 40) <<   endl;
cout   << " max ( 45 . 67F, 12 . 32 F)    : " << max (45. 67F, 12 .
32 F)  <,
endl;
return    0 ;
}
```

Output

```
max ( 'A' , 'a')          : a
max ( 30 , 40 )           : 40
max ( 45.67F, 12 . 32F )  : 45 . 67
```

In the example, the list of parameters is composed of only one parameter. The next line specifies that the function takes two arguments and returns a value, all of the defined in the formal parameter list. See what happens if the following command is issued, keeping in mind the template definition for max ( ) :

```
max ( a , b ) ;
```

**Notes**

in which a and b are integer type variables. When the user invokes max ( ) , using two int values, the identifier, 'type', is substituted with int, wherever it is present. Now max ( ) works just like the function int max (int, int) defined earlier, to compare two int values. Similarly, if max ( ) is invoked using two double values, 'type' is replaced with 'double'. This process of substitution, depending on the parameter type passed to the function, is called template instantiation. The template specifies how individual functions will be constructed, given a set of actual types. The template facility allows the creation of a blueprint for a function like max ( ).



*Did u know?* **Is the template or blueprint can then be instantiated for all data types?**

Yes, the template or blueprint can then be instantiated for all data types, eliminating duplication of the source code. The identifier, 'type', can be used within the function body of a function that, otherwise, remains unchanged.



*Example:* The template for a function called square 9, which calculates the square of any number (int, float, or double type) passed to it can be given as:

```
# include < iostream.h >
template < class type >
type square ( type a)
{
        type b;
        b= a*a ;
        return b.;
}
        int main ( )
{
        cout << " square (25 )             : " << square ( 25) << endl;
        cout << " square 40                 : " << square 40 << endl;
        return 0 ;
}
```

Output

Square ( 25 . 45F) : 1125

Square 90 : 1600

Here is another example of the use of template function. Consider the following classes:

```
class IRON
{
private   :
float density ;
public :
IRON ( ) {density = 8.9 }
Float Density ( )   { return density ;}
} ;
```

Notes

### Self Assessment

Fill in the blanks:

1. A template function may be defined as an ..... .
2. Template functions and classes are usable for any ..... .
3. ..... would consist of instantiating the template definition for specific data type.
4. ..... errors are relatively easy to find and correct, even if the resulting error messages are unclear.
5. The identifier can be built-in or user-defined data type, or the ..... type.

## 14.2 Classes of Template

Template classes may be defined as the layout and operations for an unbounded set of related classes. Built in data types can be given as template arguments in the list for template classes.

Syntax

```
Template < class type 1 , ... >
Class   class - name
{
public
type 1 var,
...
};
```

Suppose you want two different classes for the same data type this can be achieved through the following code:

```
# include < iostream.h >
template < class T, int z >
class W
{
        public :
        T a ;
        W ( T q )
        {
                a = z + q ;
                cout << " a = " << a << endl ;
        }
} ;
int main ( )
{
        W < int,10 > one ( 100) ;              // Displays a = 110
```

Notes

```
        W  < int , 10 > + twoptr = & one ;     // No object is created
        W  < int, 10 + 3 > three = 200;           // Displays a = 230
        W  < float, 40 > four = 100.45 ;       // Displays  a = 140.45
        Return 0;
}
```

### *Rules for Using Templates*

Listed below are the rules for using templates:

1. The name of a parameter can appear only once within a formal parameter list. For example:

```
Template   <class type>
Type max ( type a, type b)
{
        return a > b ? a : b ;
}
```

2. The keyword, class must be specified before the identifier.
3. Each of the formal parameters should form a part of the signature of the function. For example:

```
Template < class T >
Void func ( T a )
{
......
}
```

4. Templates cannot be used for all overloaded functions. They can be used only for those functions whose function body is the same and argument types are different.
5. A built-in data type is not to be given as a template argument in the list for a template function but the function may be called with it. For example:

```
Template < class  ZZ  , int >    //  wrong
Void fn ( ZZ , int) ;
Template < class Z >              //  right
Void fn ( ZZ , int ) ;
```

6. Template arguments can take default values. The value of these arguments become constant for that particular instantiation of the template.



*Example:*

```
Template  < class X , int var = 69 >
Class Aclass
{
        public :
        void func  ( )
```

Notes

```
        {
                X   array   [ var ] ;
        }
};
```

## 14.3 The Typename Keyword

The keywords typename and class can be freely interchanged. For example

```
Template  <  class  T  >
Int  maxindex  (  T  arr  [  ]  ,  int  size  )
{
 .  .  .
}
```

## 14.4 Template Specialization

While template functions and classes are usable for any data type, that would hold true only, as long as the body of the functions or the class is identical throughout. So, if you have,

```
Template  <  typename  T  .
Void  swap  (  T  +  Ihs  ,  T  +  rhs  )
{
T  tmp  (  +  Ihs  )  ;
+  Ihs  =  +  rhs  ;
+  rhs  =  tmp  ;
}
```

the previous template will be instantiated correctly for any of the pointer types passed to it - except for the data types char + , unsigned char + , and signed char + . Since, the body of the function.



*Caution* Differs for these types, you would use template specialize the template function.

Specialization would consist of instantiating the template definition for a specific data type. So, when the compiler needs to instantiate the template, it finds a predefined version already existing and uses it. For example, just after the previous template declaration, you could create a template specialization for char + like this:

```
Template  e  >
Void  swap  <  char  >  (  char  +  Ihs  ,  char  +  rhs  )
{
char  +  tmp  =  new  char  [  strlen   (Ihs)   +  1  ]  ;
strcpy  (  tmp  ,  Ihs)  .;
strcpy  (  tmp  ,  Ihs);
```

Notes

```
Strcpy ( Ihs , rhs) ;
Strcpy ( rhs , tmp ) ;
}
```

## 14.5 Point of Instantiation

A template function gets instantiated under the following circumstances:

1. Implicitly instantiated because it is referenced from a function call that depends on a template argument.
2. Implicitly instantiated because it is referenced within a default argument in a declaration.
3. The point of instantiation of a function template specialization immediately follows the declaration or definition that refers to the specialization.

### Self Assessment

Fill in the blanks:

6. Error isolation is useful for locating an error resulting in a ..... message.
7. The goal of object-oriented software development is to construct a model of ..... .
8. ..... activities are those activities that take place simultaneously.

## 14.6 Error Handling

We now know the syntactic errors and execution errors usually produce error messages when compiling or executing a program. Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more troublesome. When an execution error occurs, we must first determine its location (where it occurs) within the program. Once the location of the execution error has been identified, the source of the error (why it occurs) must be determined. Location of the execution error occurred often assists, however, in recognizing and correcting the error.

Closely related to execution errors are logical errors. Here the program executes correctly, carrying out the programmer's wishes, but the programmer has supplied the computer with instructions that are logically incorrect. Logical errors can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error-free. Moreover, logical errors are often hard to locate even when they are known to exist (as, for example).

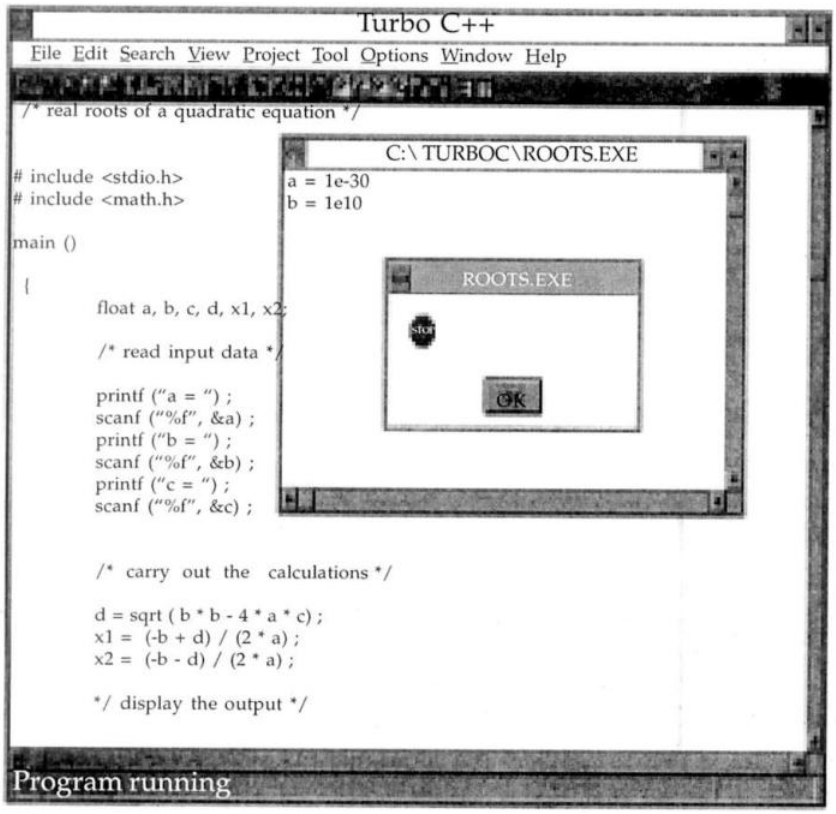
Methods are available for finding the location of execution errors and logical errors within a program. Such methods are generally referred to as debugging techniques. Some of the more commonly used debugging techniques are described below.

## 14.7 Error Isolation

Error isolation is useful for locating an error resulting in a diagnostic message. If the general location of the error is not known, it can frequently be found by temporarily deleting a portion of the program and then rerunning the program to see if the error disappears. The temporary deletion is accomplished by surrounding the instructions with comment markets (/* and */), causing the enclosed instructions to become comments. If the error message then disappears, the deleted portion of the program contains the source of the error.

Notes

Figure 14.1




A closely related technique is that of inserting several unique printf statements, such as

Printf("debugging - line 1/n");

Printf("Debugging - line 2/n");

etc. at various places within the program. When the program is executed, the debug messages will indicate the approximate location of the error.

*Notes* The source of the error will lie somewhere between the last printf statement whose message did appear, and the first printf statement whose message did not appear.

### *Watch Values*

A watch value is the value of a variable or an expression which is displayed continuously as the program executes. Thus, you can see the changes in a watch value as they occur, in response to the program logic. By monitoring a few carefully selected watch values, you can often determine where the program begins to generate incorrect or unexpected values.

### *Breakpoints*

A breakpoint is a temporary stopping point within a program. Each breakpoint is associated with a particular instruction within the program. When the program is executed, the program

Notes

execution will temporarily stop at the breakpoint, before the instruction is executed. The execution may then be resumed, until the next breakpoint is encountered. Breakpoints are often used in conjunction with watch values, by observing the current watch value at each breakpoint as the program executes.

### *Stepping*

Stepping refers to the execution of one instruction at a time, typically by pressing a function key to execute each instruction. In Turbo C++, for example, stepping can be carried out by pressing either function keys F7 or F8. ( F8 steps over subordinate functions, whereas F7 steps through the functions.) By stepping through an entire program, you can determine which instructions produce erroneous results or generate error messages.

## Self Assessment

Fill in the blanks:

9. ..... objects are objects that are capable of controlling and scheduling the received before they are served by the object.
10. There are many alternative approaches for introducing concurrency and ..... is an object-oriented language.
11. For concurrent object-oriented system ..... .

## 14.8 Concurrent Object-oriented Systems

Concurrent objects are objects that exist and operate in the same environment simultaneously.

Our most important goals of object-oriented software development is to construct a model of the real world. We do this because we want to build a conceptual model, or a description, of the real world. A closer look at the real world reveals that concurrent activities appear everywhere. Concurrent activities are those activities that take place simultaneously.

The entities in the real world are often nested, and so are the objects in software systems. Each of these nested objects may encapsulate nested activities again, and multiple interactions may occur between these objects in parallel. For example, consider a bank with a number of clerks serving customers. Provided it is open, customers may enter the bank and wait in line until they can interact with one of the clerks at the counter. The clerks may in turn interact with other employees or departments of the bank in order to fulfill the requests of the customers. All these activities are happening at the same time and therefore are concurrent.

According to the object-oriented paradigm, each object is an autonomous agent, capable of handling received requests. Active objects are objects that are capable of controlling and scheduling the received requests before they are served by the object.

This observation leads to the following conclusions: to properly model real-world situations, concurrent activities must be modeled. Each object must be capable of dealing with multiple, concurrent requests. Requests may be serialized or trigger at the same time.

As stated before, our primary motivation for adopting concurrency lies with the modeling issues that were just discussed. It should be remarked, however, that an object-oriented model lends itself much more for a realisation on a distributed or parallel architecture, because objects are concurrent by nature. It is thus relatively easy to identify tasks that can be executed in parallel.

**Notes**

There are many alternative approaches for introducing concurrency and synchronisation in an object-oriented language. The synchronization can be handled in two different ways:

***code level:*** In this scheme, messages are always accepted for execution. Concurrency is controlled through conventional mechanisms such as semaphores and monitors, which appear as statements that are embedded within the implementations of method bodies.

This category is referred to as passive objects and is exemplified by Smalltalk-80. The internal state of objects is only protected against inconsistencies when the methods that affect the state contain synchronisation statements.

***Object level:*** In this case we speak about active objects: upon reception, messages can be delayed until apt for execution. Thus, synchronisation occurs at the object boundary, thereby protecting the internal consistency of objects. Various types of synchronisation mechanisms can be applied at this level. Examples of languages that synchronise at the object level are POOL, Procol, etc.

For concurrent object-oriented systems a systematic development framework is provided by Rumbaugh method explained below:

Rumbaugh method (also known as Object Modeling Technique) of developing concurrent object-oriented systems is applicable in all the phases of development. It enumerates the steps that must be taken to accomplish the outcome(s) of that phase. It roughly consists of the following steps:

1. Identify objects, classes, hierarchy and relationships.
2. Identify the synchronization constraints.
3. Specify the object and system behavior, i.e. dynamic behaviour.
4. Specify the functional model using DFD or otherwise.
5. Iterate through the steps to refine the model to the acceptable levels.

The performance and applicability of an object-oriented system can be greatly enhanced if several instances of objects are allowed to co-exist and co-execute in a single environment. Such systems facilitate execution of multiple objects in a single interactive environment.

As a matter of fact concurrency can exists at different levels of abstractions. Concurrency of programs (multiprogramming), for example, has been implemented as multiprocessing or multitasking. Concurrency within an object can be achieved by multithreading.

No matter which level concurrency is in question, a concurrent system must have the ability to control three aspects of execution:

1. Concurrent execution of methods
2. Inter-method synchronization and communication
3. Security



*Task* Traditionally, concurrency within objects has been implemented on the tunes of multi-tasking systems. As a process provides abstraction in the later case, threads abstract the later case. Explain.

### Thread

Notes

A thread is a sequence of instructions to be executed within a program. Each thread has its own set of resources such as instruction pointer, set of registers and stack memory. The virtual address space is common to all threads within a process. This enables all the threads to access data on the heap.

Normal processes consist of a single thread of execution that starts in a single method (main() in UNIX). Each line of the code is executed exactly one line at a time. Earlier the normal way to achieve concurrency in a program was to use the fork() and exec() system calls (or equivalent system calls in other operating systems) to create several processes. Each of these processes used to e executed as a single thread of execution.

Since there are a lot of similarities between a process and a thread, a thread is often referred to as lightweight process.

Threads can be created as instances of the Thread class. It has attributes and methods that create and control a thread, set its priority, and get its status. The namespace of the Thread class is System.Threading assembled in mscorlib.dll. Given below is its syntax.

[ComVisibleAttribute(true)]

[ClassInterfaceAttribute(ClassInterfaceType::None)]

public ref class Thread sealed : public CriticalFinalizerObject, _Thread

A single process can create as many threads as required to execute different portions of the code. The program code to be executed by a thread is specified using ThreadStart delegate or the ParameterizedThreadStart delegate. The ParameterizedThreadStart delegate allows one to pass data to the thread procedure.

The following program demonstrates simple threading functionality offered by C++. The code must be compiled using /clr option.

```
using namespace System;

using namespace System::Threading;

public ref class ExampleThread

{

public:

   // The ThreadProc method is called when the thread starts. It loops five
times, writing to

// the console and yielding the rest of its time slice each time, and then
ends.

       static void ThreadProc()

       {

              for ( int i = 0; i < 5; i++ )

              {

                     Console::Write( "Thread Procedure: " );

                     Console::WriteLine( i );

                     Thread::Sleep(0);

              }
```

**Notes**

```
        }
};
int main()
{
        Console::WriteLine( "From Main Thread: Start a second thread." );
// Create the thread, passing a ThreadStart delegate that represents the
// ThreadExample::ThreadProc method. For a delegate representing a
// static method, no object is required.
Thread^oThread=gcnew Thread(gcnew ThreadStart(&ExampleThread::ThreadProc) );
// Start ThreadProc. Note that on a uniprocessor, the new thread does not get any
// processor time until the main thread is preempted or yields. Uncomment the
// Thread.Sleep that follows Start() to see the difference.
    Thread->Start();
        //Thread::Sleep(0);
        for ( int i = 0; i < 4; i++ )
        {
        Console::WriteLine( "From Main Thread: Work Work Work." );
            Thread::Sleep( 0 );
        }
Console::WriteLine( "From Main Thread: Call Join(), to wait until ThreadProc ends." );
        oThread->Join();
        Console::WriteLine( "From Main thread: Press Enter to end program." );
        Console::ReadLine();
        return 0;
}
```

Here is the result of a run of this simple example.

From Main Thread: Start a second thread.

From Main Thread: Work Work Work.

Thread Procedure: 0

From Main Thread: Work Work Work.

Thread Procedure: 1

From Main Thread: Work Work Work.

Thread Procedure: 2

Notes

From Main Thread: Work Work Work.

Thread Procedure: 3

From Main Thread: Call Join(), to wait until ThreadProc ends.

Thread Procedure: 4

From Main Thread: Press Enter to end program.

### *Multithreading*

A single process can be further broken into several threads so as to enhance the overall performance of the process. This is called multithreading. Hence, multithreading is a technique for achieving concurrency within a process.

### *Multithreading vs. Multiprocessing*

Multithreading allows an application with multiple threads running within a process. On the other hand, multiprocessing refers to an application organized across multiple processes.

Context switching and synchronization costs are comparatively lower in threads. Since the address space is shared among threads no extra work is required to access them. However, because of this the failure of one thread in a process can ring down all the other threads of that process. In contrast, since processes are insulated from each other by the operating system, an error in one process cannot bring down another process.



*Notes* Processes may run on behalf of different users and therefore may have different permissions unlike threads.

### *Limitations of Threads*

Despite having so much promises threads do carry a lot of pitfalls along the way. Programmers should take proper caution while programming with threads. Some of the caveats of thread programming is discussed below.

### *Race Conditions*

A race condition is said to exist where the behavior of code depends on the interleaving of multiple threads. Single-threaded codes runs as a single sequence of statements from which we can assume that data does not change between statements. However, we cannot make the same assumption for a single program statement which may compile into more than one lower level statements, i.e., we cannot guarantee the outcome of a statement such as total++; if total is shared between multiple threads. This is perhaps the most fundamental problem with multi-threaded programming. Consider the following code that depicts a logical race condition.

```
int SharedVar = 20;
void* ThreadOne(void*)
{
        while(SharedVar > 0)
        {
```

Notes

```
                CallSomeMethod();
                —SharedVar;
        }
}
```

When this code is executed as a single thread, CallSomeMethod will execute 20 times. However, imagine that we start a number of threads, all executing ThreadOne(). However, with more than one thread existing simultaneously the method CallSomeMethod() will most likely be executed too many times. Exactly how many times depends on the number of threads spawned, computer architecture, operating system scheduling and lick. The problem arises because we do not test and update SharedVar as an atomic operation, so there is a period where the value of SharedVar is incorrect. During this time other threads can pass the test when they really shouldn't have.



*Did u know?* **What is the purpose of SharedVar?**

The value of SharedVar on exit tells us how many extra times CallSomeMethod() is called. The value of SharedVar on exit will be 0 if there is a single thread running CallSomeMethod().

Race conditions can be avoided using an object called mutex.

A mutex is synchronization primitive provided by the operating system that can be used to ensure that a section of code executes by one thread at a time. A mutex has two states - locked and unlocked. In locked state any further attempt to lock it will suspend the thread. The waiting threads can acquire lock on mutex and resume execution only when the mutex becomes unlocked. The thread that locks the mutex can only unlocked it. Here is the solution for the above code using mutex.

```
int SharedVar = 20;
mutex MutexVar;
void ThreadTwo()
{
        while(SharedVar > 0)
        {
                bool flag = false;
                {
                mutex::lock(MutexVar);
                if(SharedVar > 0)
                {
                        —SharedVar;
                        flag = true;
                }
                }
                if(flag) CallSomeMethod();
        }
}
```

Notes

As is evident, the shared variable is checked and updated as an atomic operation so that the race condition does not occur.

### *Deadlock*

The concurrency may lead to a situation where one or more threads wait for resources that can never become available. Such a condition is called a deadlock. Consider the following code that illustrates a deadlock.

```
mutex X;
mutex Y;
void ThreadThree()
{
        int ct = 0;
        while(true)
        {
        mutex::lock(X);
        mutex::lock(Y);
        cout << "ThreadThree in action" <<++ct<< "\n";
        }
}
void ThreadFour()
{
        int ct = 0;
        while(true)
        {
        mutex::lock(Y);
        mutex::lock(X);
        cout << "ThreadFour in action" << ++ct<< "\n";
        }
}
```

When this code is run a deadlock is possible. The two threads may enter a situation where ThreadThree is waiting for ThreadFour to release the locks while ThreadFour waits for ThreadThree to release the locks.

To resolve this simple deadlock all we need to do is to ensure that we lock resources in a consistent order. Thus, changing ThreadFour to lock X before Y ensures there will be no deadlock.

## Self Assessment

Fill in the blanks:

12. A …….. is a sequence of instructions to be executed within a program.

Notes

13. A single process can be further broken into several threads so as to enhance the overall performance of the…….....

14. …….. method (also known as Object Modeling Technique) of developing concurrent object-oriented systems is applicable in all the phases of development.

15. …….. allows an application with multiple threads running within a process.



*Caselet*

## Software Acquisitions — Discretion to Dictate Deals

"Going global" through overseas acquisitions is the flavour for India Inc. over the past few months. This week, it was the turn of the software sector to jump on to the bandwagon, breaking the spell of inactivity on the acquisitions front by frontline companies.

The previous acquisition of note by a frontline company was Wipro's buy of US-based Nervewire Inc in April. Two companies from across the software spectrum — Infosys Technologies from the software services space and i-flex solutions from the products space — unveiled their maiden acquisitions — of Australia-based Expert Information Services and US-based SuperSolutions respectively. The media buzz surrounding the Infosys deal was particularly high, given the fact that it was its first acquisition made after listing its ADR on Nasdaq in 1999 to help explore inorganic growth options. But the stock market appeared lukewarm to these deals, with the stocks hardly budging on the day of announcement. The uptrend in the Infosys stock on Friday was largely a reflection of the bullish sentiment across Sensex heavyweights, rather than any stock-specific activity. In the backdrop of these deals, two key trends can be collated for the future.

**Presaging a trend?**

The Infosys acquisition may not be suggest a trend in the software services arena. This becomes evident from the fact that Infosys sifted 135 candidates over the past four-and-half years to find the right match this week. Though the company has been sitting on $470 million in cash and equivalents (including investments in liquid mutual units) in its balance-sheet, less than 5 per cent of it is to be used for making this acquisition. Finding the "right fit", proper due diligence of the target company and financial prudence will continue to dictate Infosys' future acquisition strategy.

Second, it cannot be said that competitive pressures were at play driving Infosys to make this deal. Wipro, its closest competitor, had announced four acquisitions — two each at home and abroad over the past 12-15 months. And the last deal by Wipro involving Nervewire Inc was put through in April. Moreover, the size of the Infosys deal at $22.9 million (relative to $18.7-million Wipro-Nervewire deal or $11.5million of i-flex solutions) is fairly small. This may hardly change the competitive dynamics for frontline companies such as Satyam Computers and HCL Technologies to press the panic button on the strategic front. The only signal that Infosys has sent out is that it will not be averse to acquisitions when the company offering the right match at the right price comes along.

**Integration Challenges**

Going by the merger and acquisition (M&A) experience in the software arena, Indian software companies, be it frontline or medium sized, are acutely conscious of the challenges of integration and employee retention issues.

*Contd...*

Notes

The record of the medium-sized Indian software companies has been mixed. For every successful acquisition, there have been instances of prolonged integration problems, while others have led to financial problems. Take for instance, Silverline Technologies' acquisition of e-business consulting firm, Seranova Inc. US in 2000/01. Practically, all the financial problems that have bedevilled Silverline since then could be linked in some way to this ambitious acquisition.

Leave alone overseas acquisitions, medium sized companies have had a tough time putting together M&As between two domestic entities. Take the case of Polaris Software. Its merger with Orbitech Solutions, the technology subsidiary of Citigroup, first announced in May 2002, ran into a multitude of problems relating to integration of the two work cultures. By October 2002, the stock swap ratio had to be changed and Polaris faced sustained challenges on the employee front. Though some signs of stability in the financial performance have been seen in the recent quarters, sustained benefits from the merger will take time to get fully reflected in the financials. Ultimately, the success of any M&A exercise itself takes at least a year or two to pan out. As Cisco's CEO, Mr John Chambers, said, "In the average acquisition, 40 to 80 per cent of the top management and key engineers are gone in two years. I would measure the success of my acquisitions through retention of people and revenue that you generate two or three years later."

It is hardly surprising that Infosys has structured its latest deal with milestone payments every year subject to the certain financial and key employee retention targets being met.

And since no foolproof formula has been evolved for managing the integration, a cautious approach to M&A will remain the norm.

## 14.9 Summary

- In C++ when a function is overloaded, many copies of it have to be created, one for each data type it act on. A template function may be defined as an unbounded functions.
- The definition of the templates begins with the template keyword followed by a comma-separated list of parameter types enclosed within the less than (<) and greater than (>) signs.
- Templates classes may be defined as the layout and operations for an unbounded set of related classes.
- While template functions and classes are usable for any data type, that would hold true only, as long as the body of functions or the class is identical throughout.
- Specialization would consist of instantiating the templates definition for a specific data type if the templates instance does not exist, and the definition is not visible, generate an error.
- Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors, on the other hand, can be much more trouble some.
- Logical errors can be very difficult to detect, since that output resulting from a logically incorrect program may appear to be error-free. Errors isolation is useful for locating an error resulting in a diagnostic message.

## 14.10 Keywords

***Break Point:*** A breakpoint is a temporary stopping point within a program.

***Logical Errors:*** Logical error can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error free.

Notes

***Stepping:*** Stepping refers to the execution of one instruction at a time, typically by pressing a function key to execute each instruction.

***Syntactic Errors:*** Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear.

***Template:*** A template function may be defined as an unbounded functions. All the possible parameters to the function are not known in advance and a copy of the function has to be created as and when necessary.

***Template Classes:*** Template classes may be defined as the layout and operations for an unbounded set of related classes.

***Watch Value:*** Watch value is the value of a variable or an expression which is displayed continuously as the program executes.

## 14.11 Review Questions

1. What are Templates?
2. Define template instantiation.
3. List down the rules for using templates.
4. How can you get a template function instantiated?
5. Explain the use of:
   (a) Watchvalues
   (b) Breakpoints
   (c) Stepping
6. Securitize the meaning of templates.
7. Give the syntax to declare a template. Explain with an example.
8. Define the template class with the explaining program.
9. Explain the rules for using templates briefly.
10. What do you mean by template specialization?
11. When can a template function be instantiated?
12. What happen if the template instance does not exist?

### Answers: Self Assessment

1. Real word
2. Concurrent
3. Active
4. Synchronization
5. identifier
6. Rumbaugh
7. unbounded functions
8. data type
9. specialization
10. syntactic
11. diagnostic
12. thread
13. process
14. Rumbaugh
15. Multithreading

## 14.12 Further Readings

Notes



*Books*

E. Balagurusamy, *Object-oriented Programming through C++*, Tata McGraw Hill.

Herbert Schildt, *The Complete Reference-C++*, Tata Mc Graw Hill.

Robert Lafore, *Object-oriented Programming in Turbo C++*, Galgotia Publications.



*Online links*

http://www.cplusplus.com/doc/tutorial/templates/

http://cutebugs.net/files/cpp/index.html

**LOVELY PROFESSIONAL UNIVERSITY**
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

