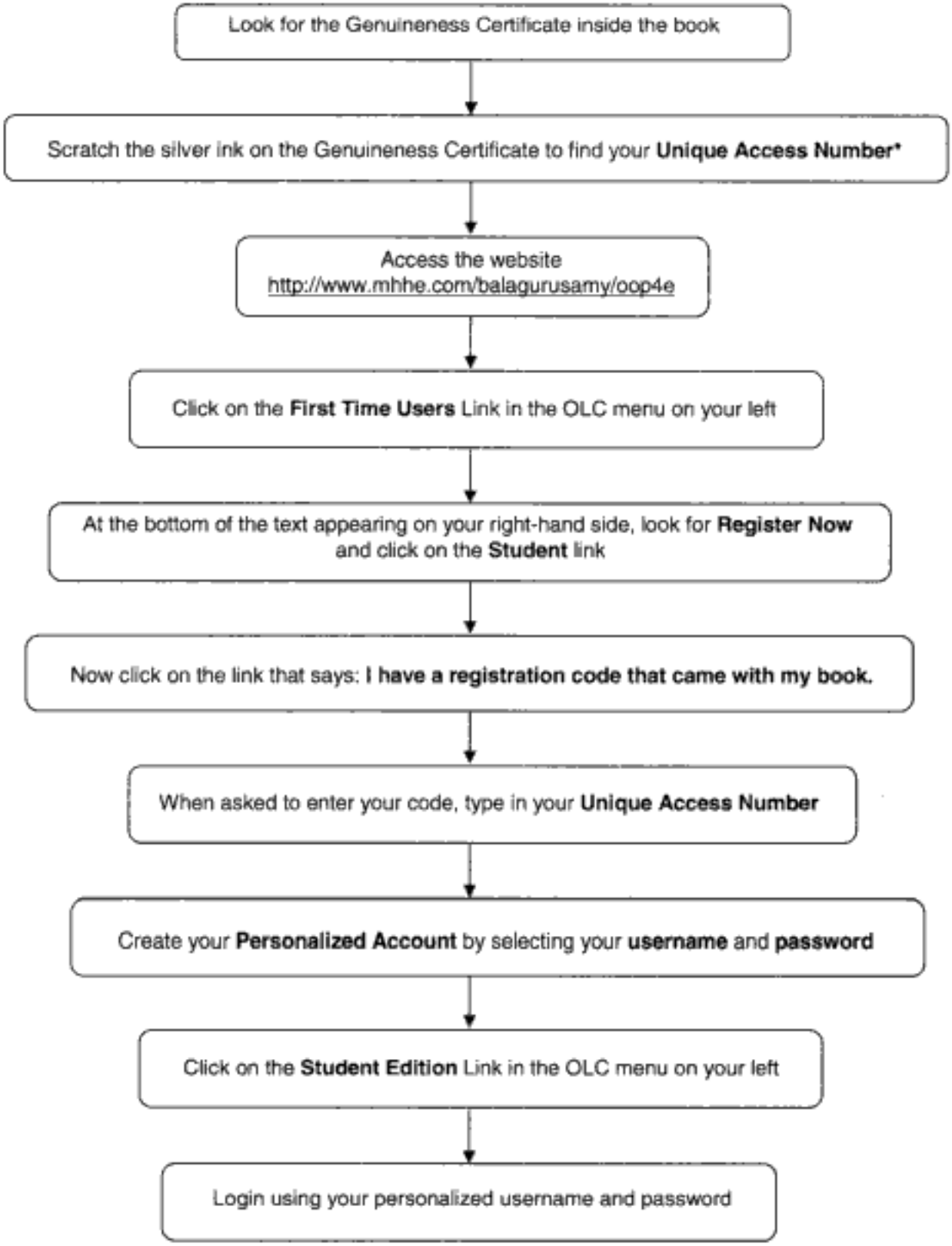
The McGraw-Hill Companies

FOURTH EDITION

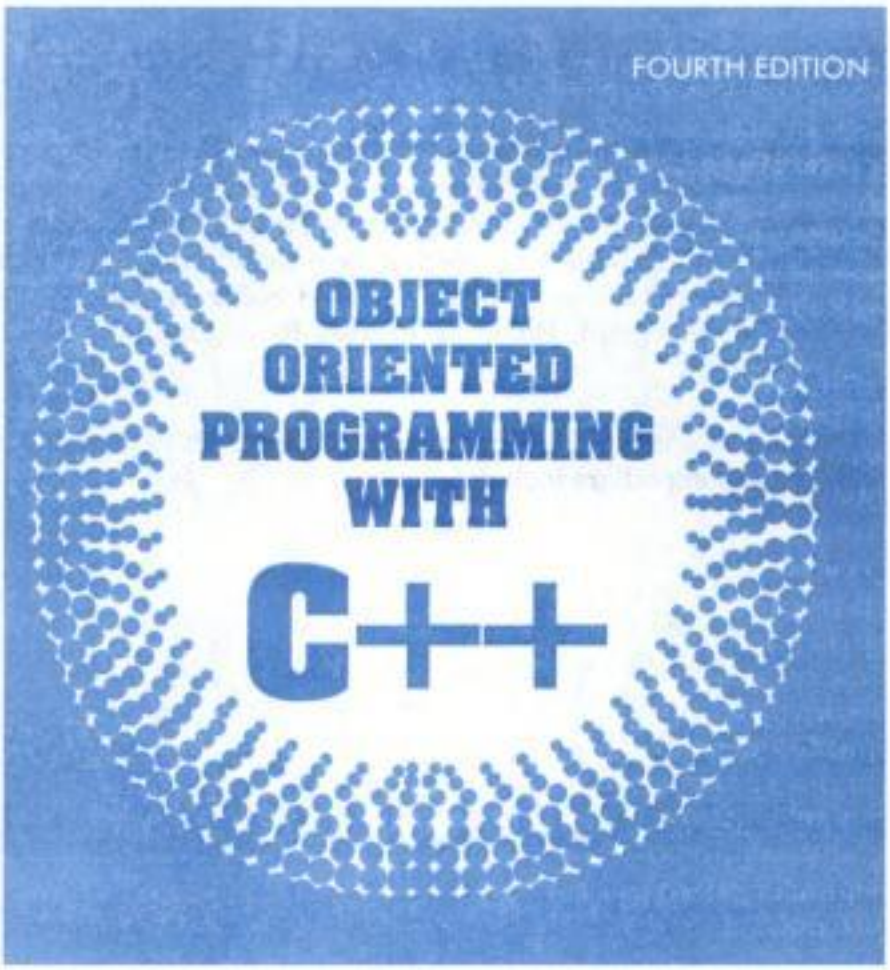
OBJECT ORIENTED PROGRAMMING WITH C++

E BALAGURUSAMY

Now a unique opportunity to access the Web Resources!



* This number is meant for *one time use* and is *self destructible*



About the Author

E Balagurusamy, former Vice Chancellor, Anna University, Chennai, is currently Member, Union Public Service Commission, New Delhi. He is a teacher, trainer, and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and a Ph. D. in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include Object-Oriented Software Engineering, Electronic Business, Technology Management, Business Process Re-engineering, and Total Quality Management.

A prolific writer, he has authored a large number of research papers and several books. His best selling books, among others include:

- Programming in C#, 2/e
- Programming in Java, 3/e
- Programming in ANSI C, 4/e
- Programming in BASIC, 3/e
- Numerical Methods, and
- Reliability Engineering

A recipient of numerous honours and awards, he has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.

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

1

Principles of Object-Oriented Programming

Key Concepts

- Software evolution
- Procedure-oriented programming
- Object-oriented programming
- Objects
- Classes
- Data abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing
- Object-oriented languages
- Object-based languages

1.1 Software Crisis

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession.This has forced the software engineers and industry to continuously look for new approaches to software design and development, and they are becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. These rapid advances appear to have created a situation of crisis within the industry. The following issues need to be addressed to face this crisis:

- How to represent real-life entities of problems in system design?
- How to design systems with open interfaces?

- 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.

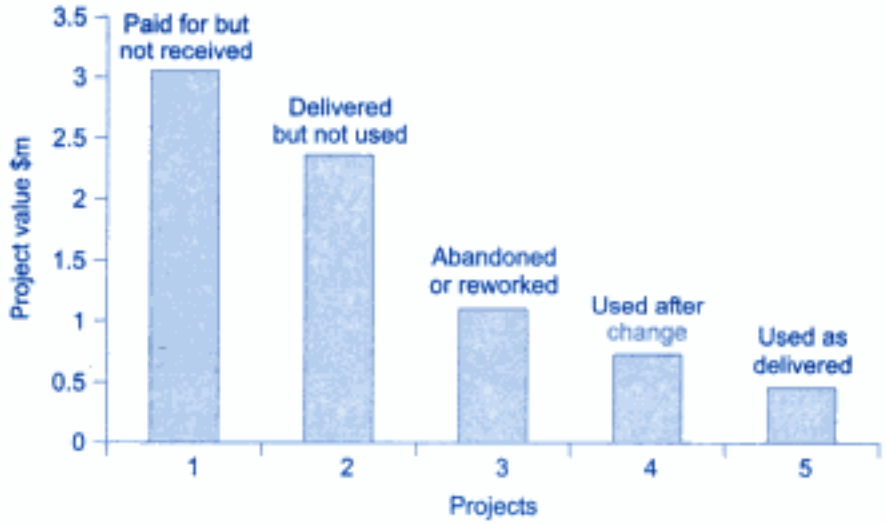


Fig. 1.1 ⇔ *The state of US defence projects (according to the US government)*

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.

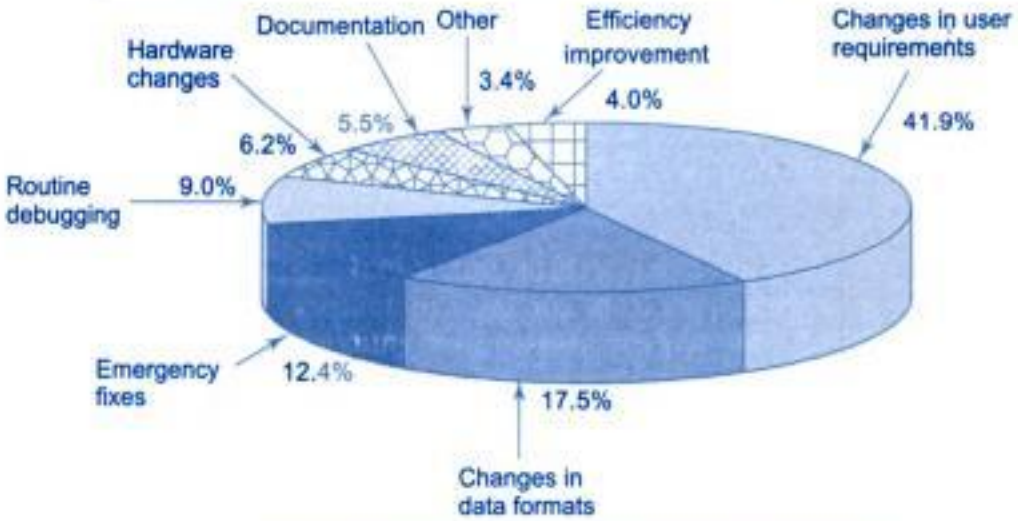


Fig. 1.2 ⇔ *Breakdown of maintenance costs*

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.

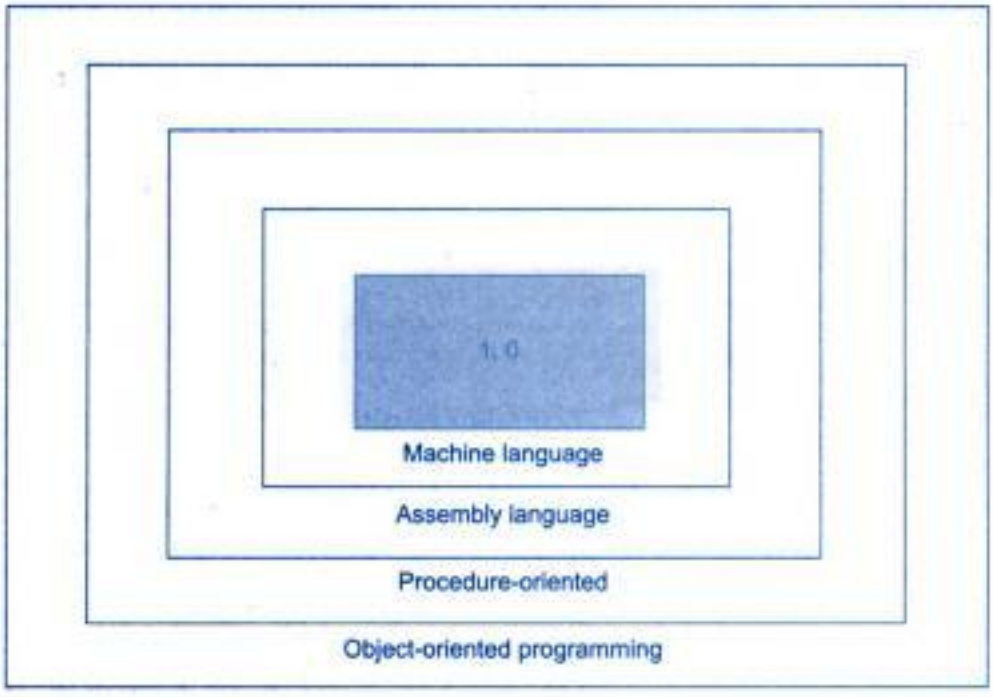


Fig. 1.3 Layers of computer software

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.

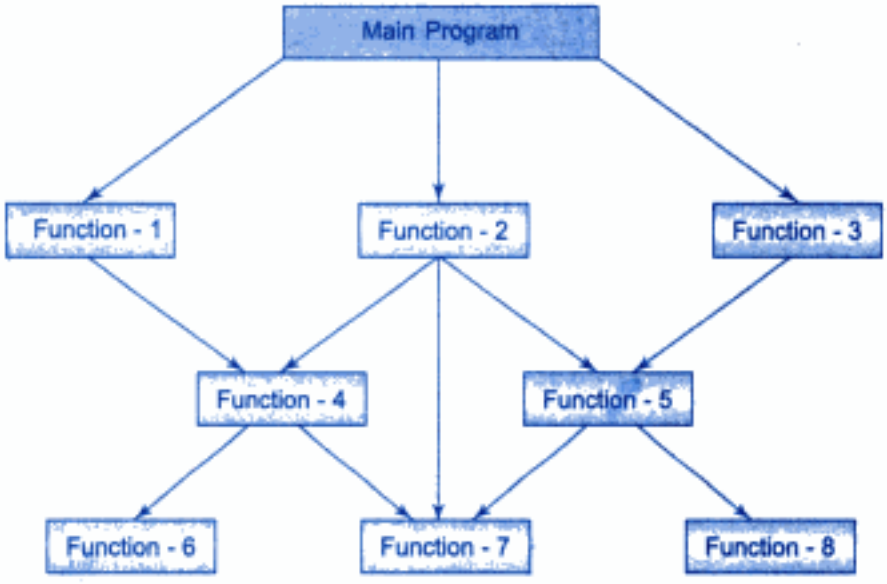


Fig. 1.4 ⇒ *Typical structure of procedure-oriented programs*

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.

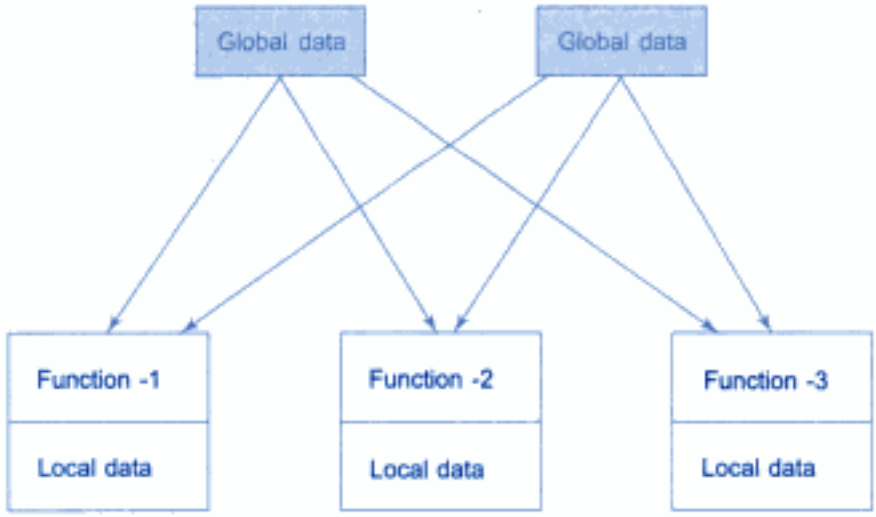


Fig. 1.5 ⇔ *Relationship of data and functions in procedural programming*

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.

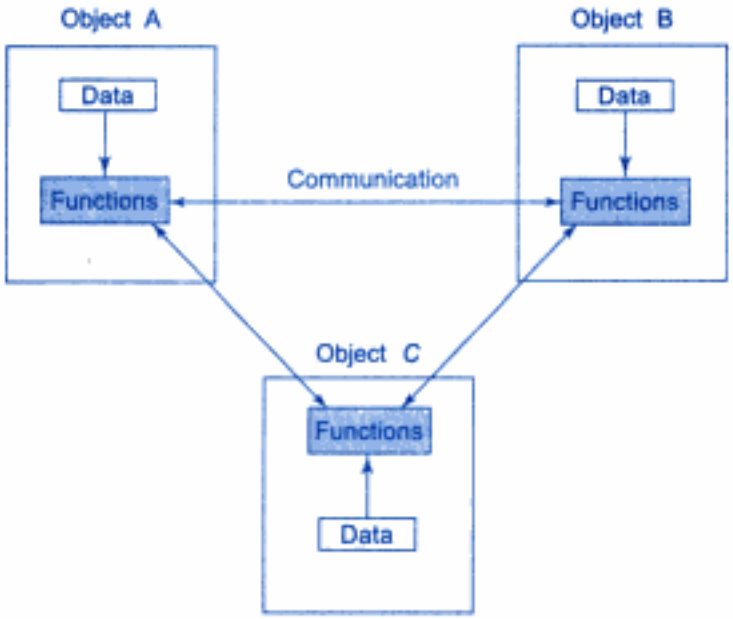


Fig. 1.6 ⇔ *Organization of data and functions in OOP*

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.

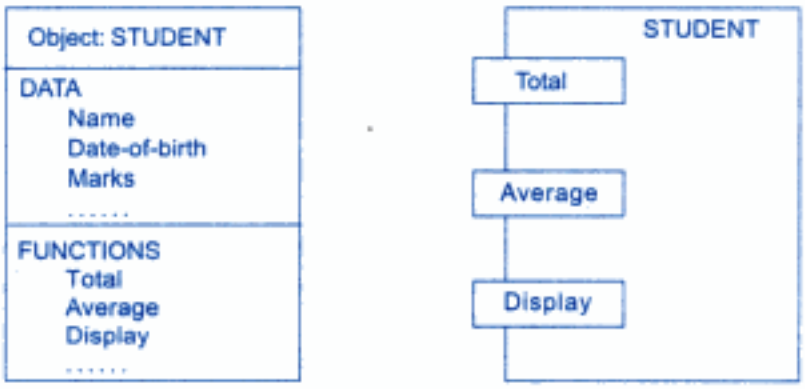


Fig. 1.7 ⇔ *Two ways of representing an object*

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.

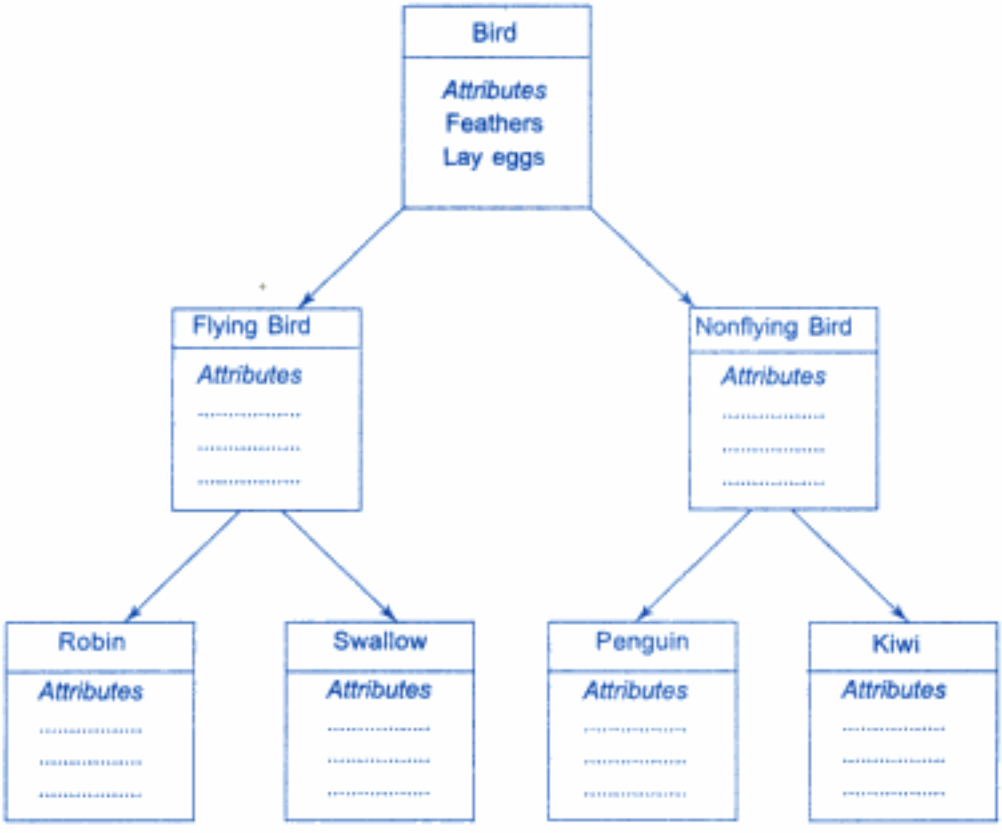


Fig. 1.8 ⇔ *Property inheritance*

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.

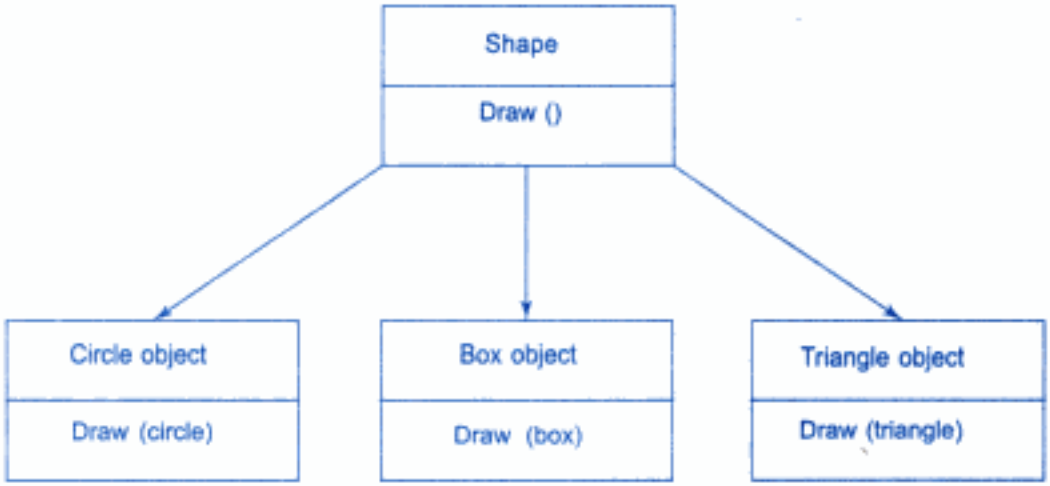


Fig. 1.9 ⇔ *Polymorphism*

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).

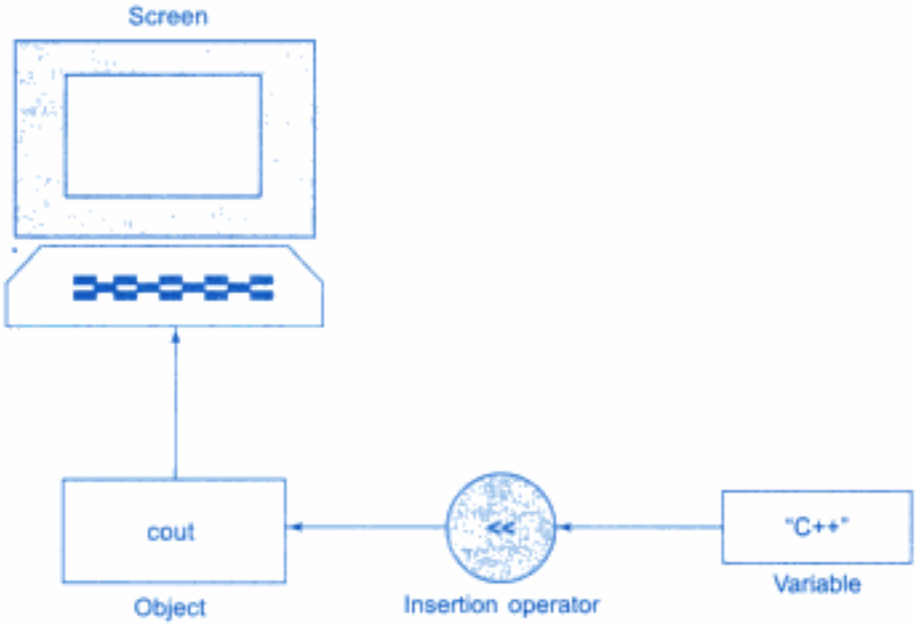


Fig. 2.1 ⇔ *Output using insertion operator*

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.

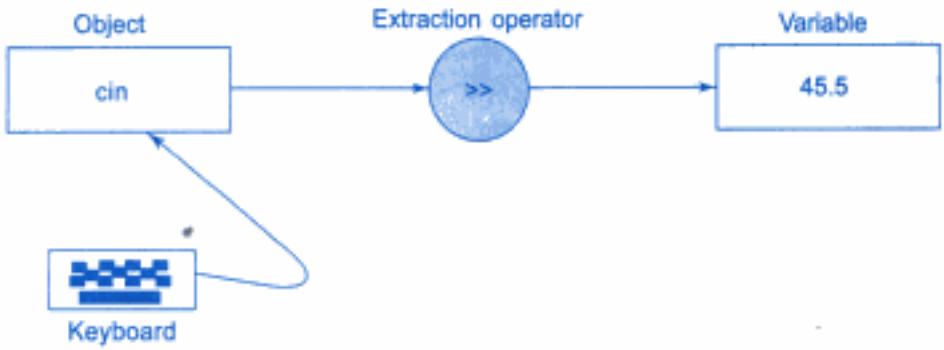


Fig. 2.2 ⇔ *Input using extraction operator*

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.

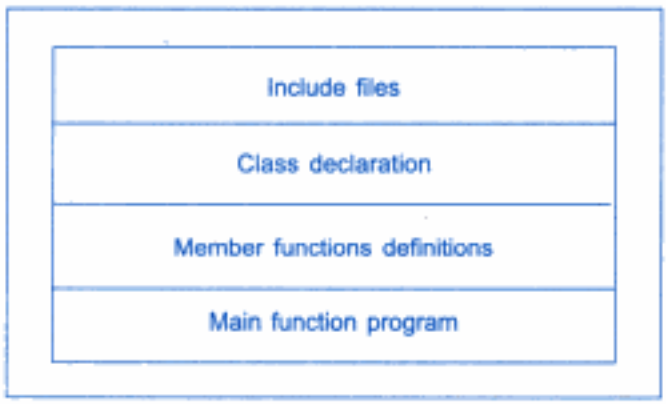


Fig. 2.3 ⇔ *Structure of a C++ program*

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.

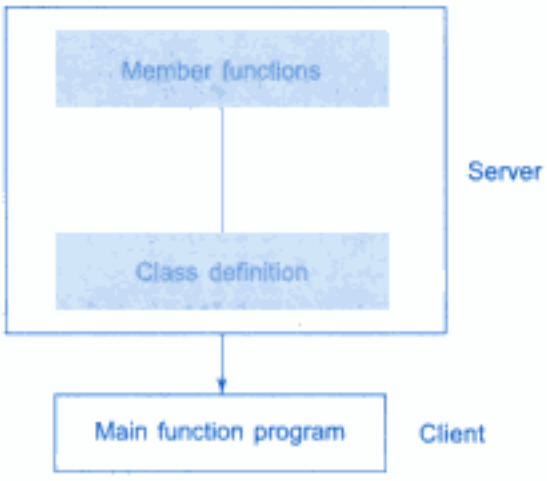


Fig. 2.4 ⇔ *The client-server model*

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 << \h "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.

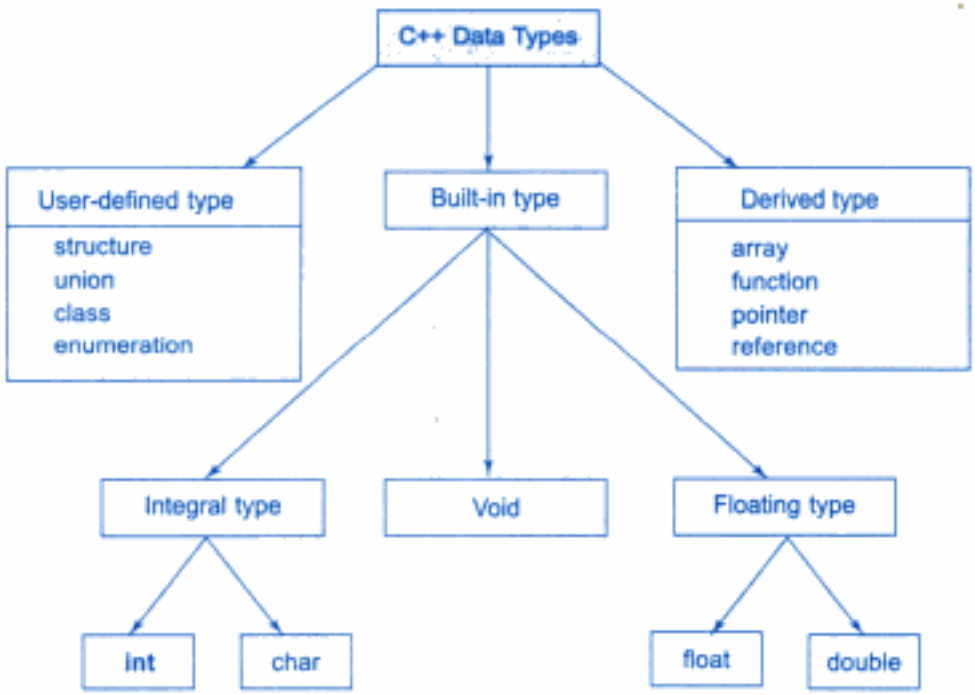


Fig. 3.1 ⇔ *Hierarchy of C++ data types*

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
```

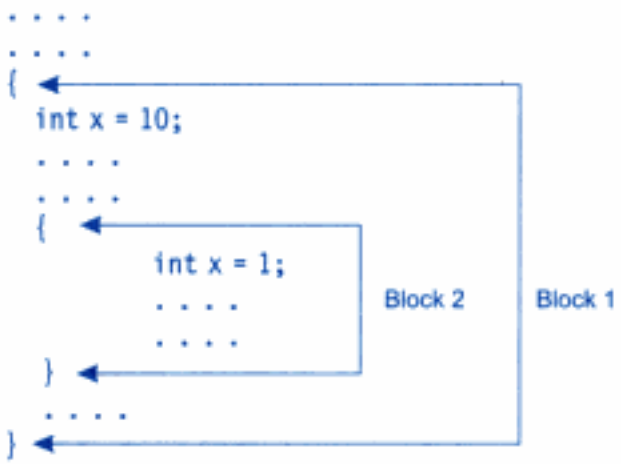
In addition, C++ also allows us to provide new definitions to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

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.

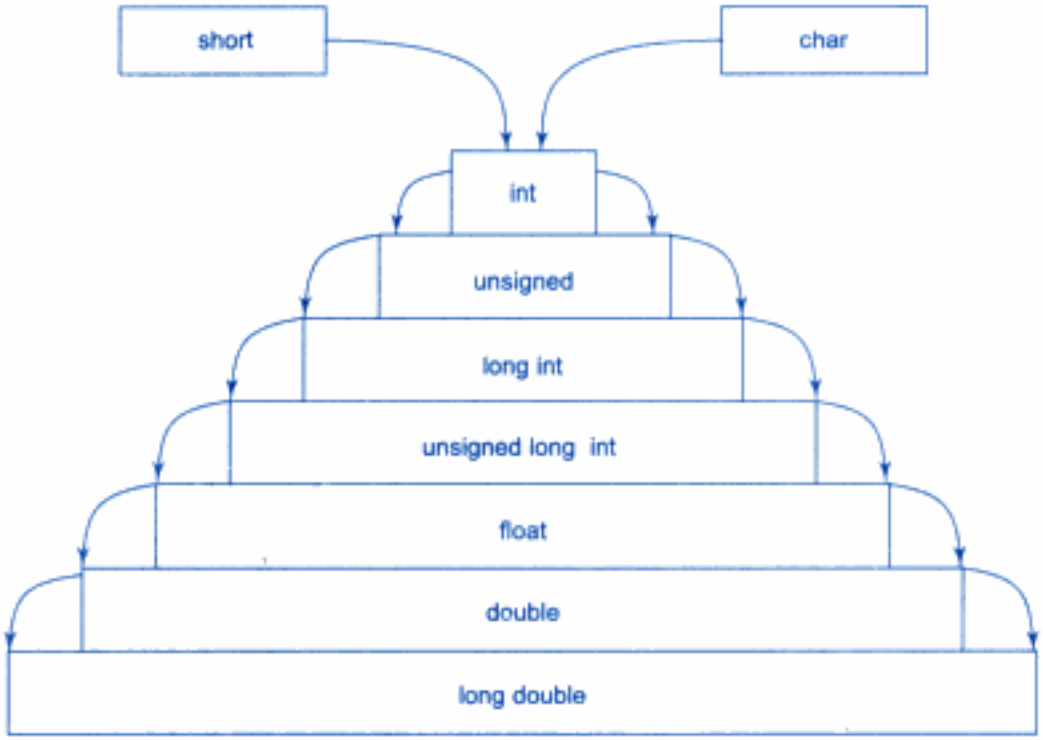


Fig. 3.3 ⇔ *Water-fall model of type conversion*

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*

RHO / LHO	*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.

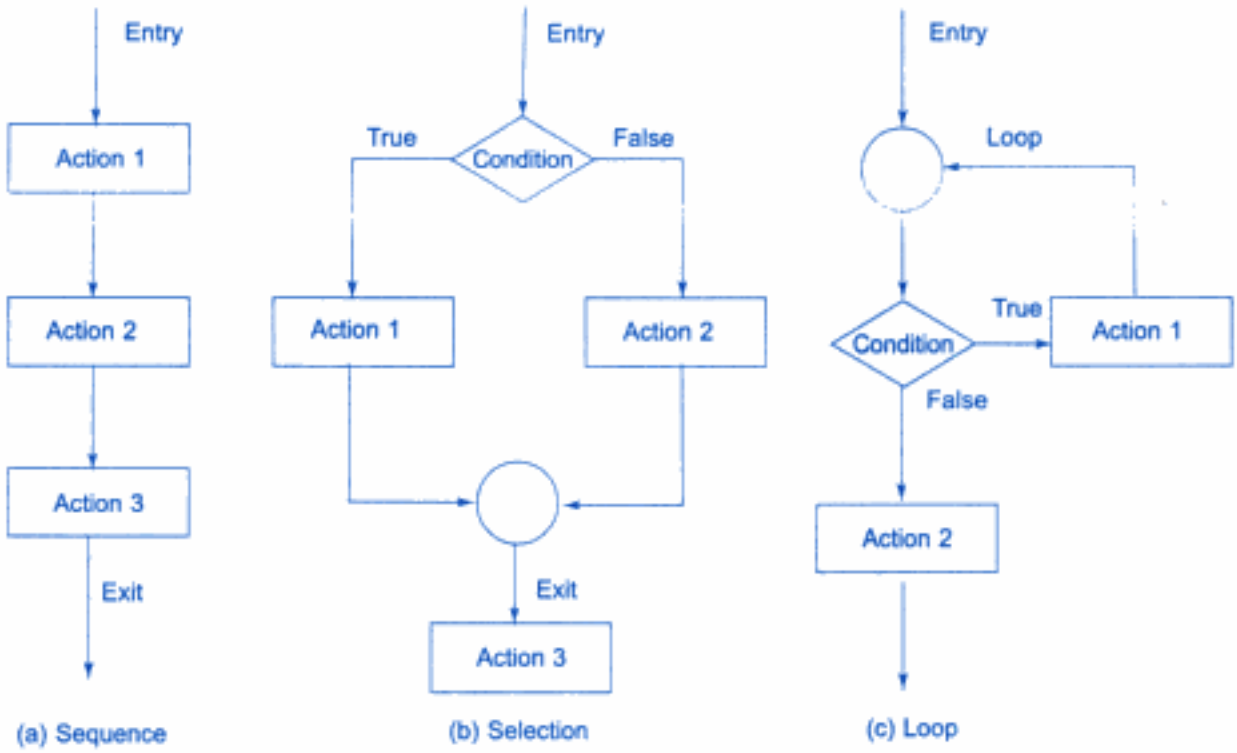


Fig. 3.4 ⇔ *Basic control structures*

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

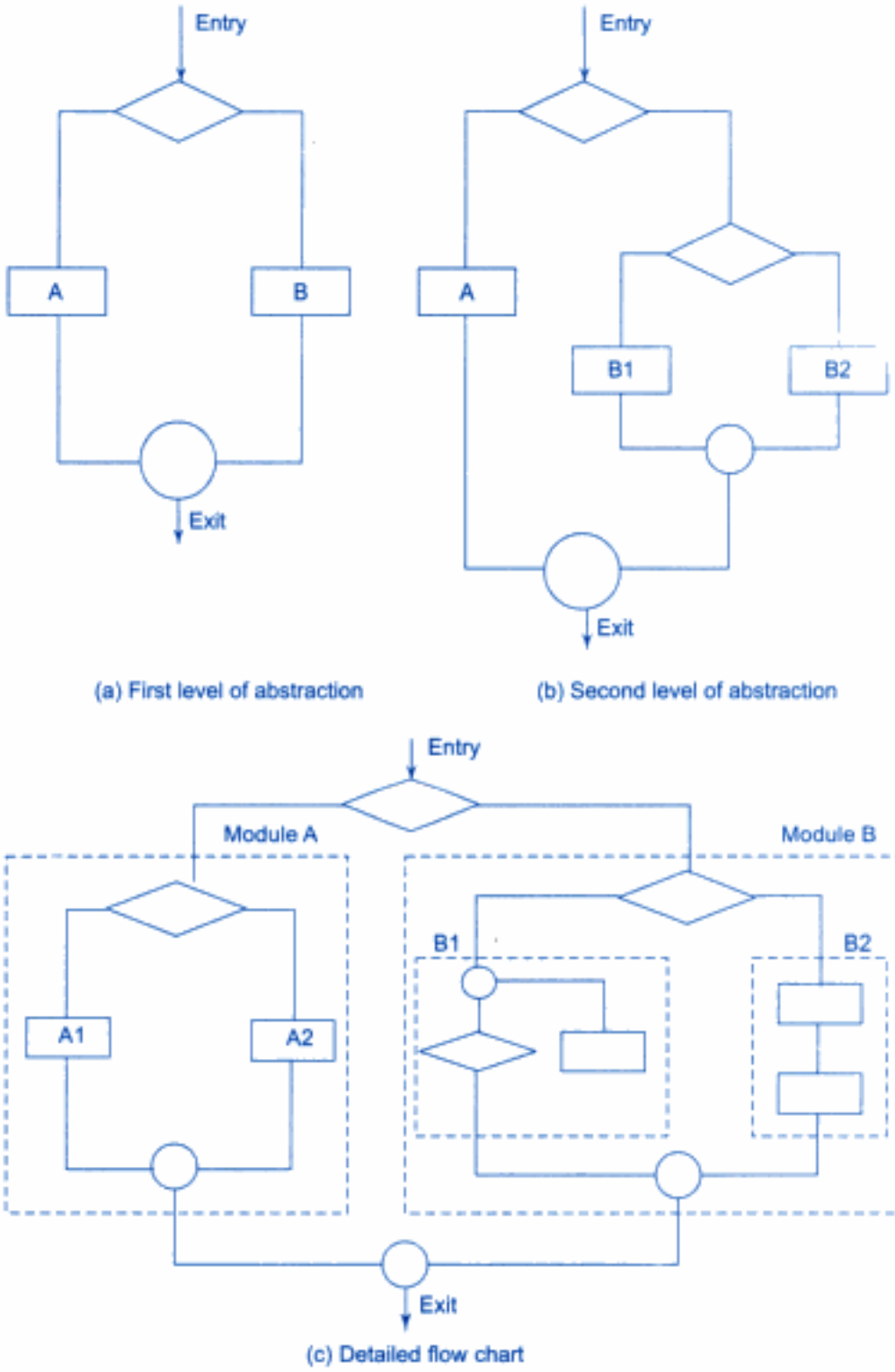


Fig. 3.5 ⇔ *Different levels of abstraction*

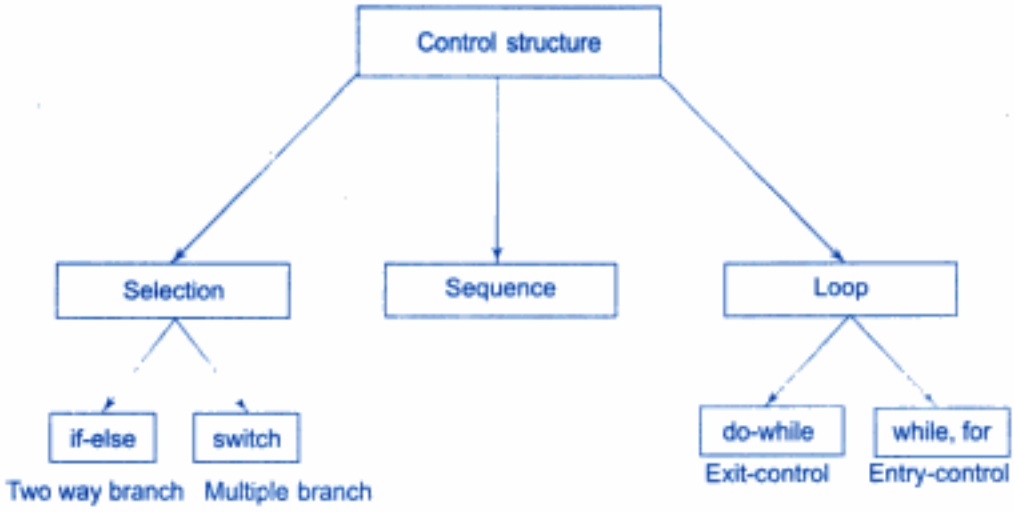


Fig. 3.6 ⇔ *C++ statements to implement in two forms*

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.*