Object Oriented Programming with C++

Revised First Edition

REEMA THAREJA

Assistant Professor
Department of Computer Science
University of Delhi

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
Ground Floor, 2/11, Ansari Road, Daryaganj, New Delhi 110002, India

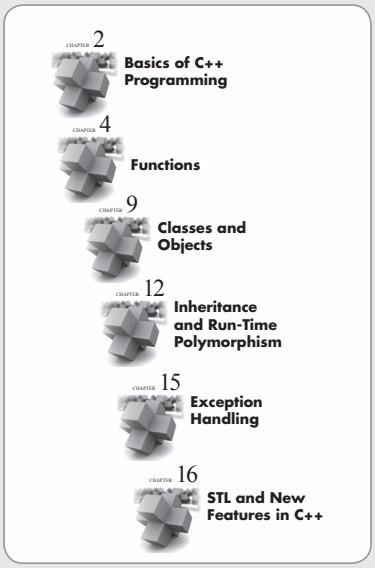
First Edition published in 2015
Revised First Edition published in 2018

ISBN-13: 978-0-19-948567-3
ISBN-10: 0-19-948567-4

Typeset in Times New Roman
by Ideal Publishing Solutions, Delhi
Printed in India by Rakmo Press, New Delhi 110020

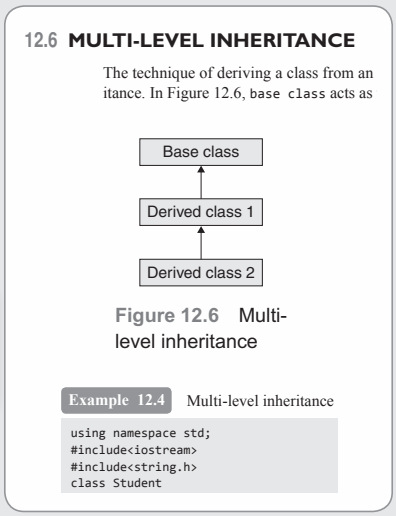
Cover image: Mmaxer / Shutterstock

Features of



Span of Coverage

The book provides wider span of coverage of topics, starting from basic concepts of object oriented programming (OOP) including discussion on Arrays, Functions, Strings, and Pointers, details on all the important concepts such as Classes, Inheritance, Operator Overloading, File Management, and Exception Handling. It also includes separate chapters on Standard Library Template (STL) and Object Oriented Systems. This will help readers have a thorough knowledge of the subject.


12.6 **MULTI-LEVEL INHERITANCE**

The technique of deriving a class from an itance. In Figure 12.6, `base class` acts as

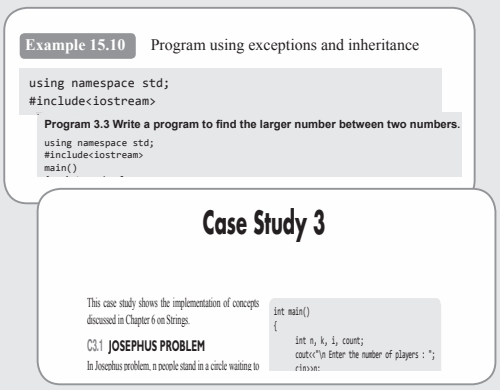
Figure 12.6 Multi-level inheritance

Example 12.4 Multi-level inheritance

```
using namespace std;
#include<iostream>
#include<string.h>
class Student
```

Approach

The book adopts bottom-up and example-based approach for explaining concepts. It first introduces the concept using simple language, examples and illustrations, and then delves into its intricacies and implementation aspects. This makes the text easier to comprehend for the readers.


Example 15.10 Program using exceptions and inheritance

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

Program 3.3 Write a program to find the larger number between two numbers.

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

Case Study 3

This case study shows the implementation of concepts discussed in Chapter 6 on Strings.

C3.1 **JOSEPHUS PROBLEM**

In Josephus problem, n people stand in a circle waiting to

```
int main()
{
    int n, k, i, count;
    cout<<"\n Enter the number of players : ";
```

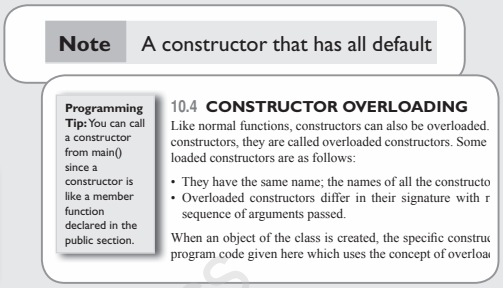
Programming Examples and Case Studies

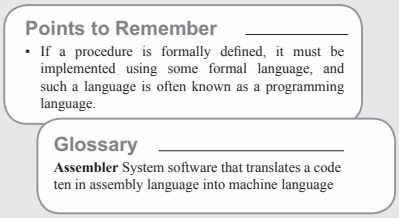
Plenty of programming examples (close to 520) along with their outputs and descriptions are provided in support of text. These examples are arranged in a simple-to-complex format. Case-studies illustrating the applications of concepts studied in different scenarios are also included in the book. These will help readers understand the concept, logic, and syntax used while developing a program, thereby transforming them into efficient programmers.

the Book

Notes and Programming Tips

Every chapter includes 'notes' mentioning the important concepts to remember and 'programming tips' stating the do's and don'ts while developing a program. This will help readers keep the critical points in mind and create error-free programs.



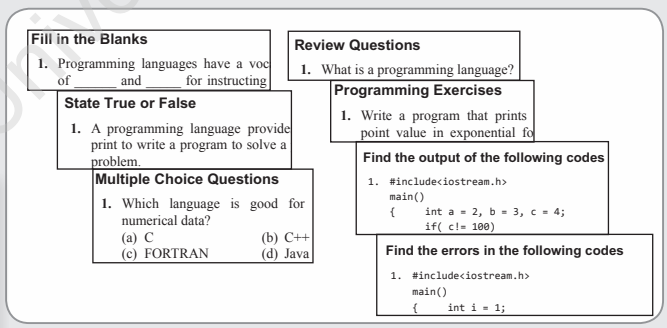


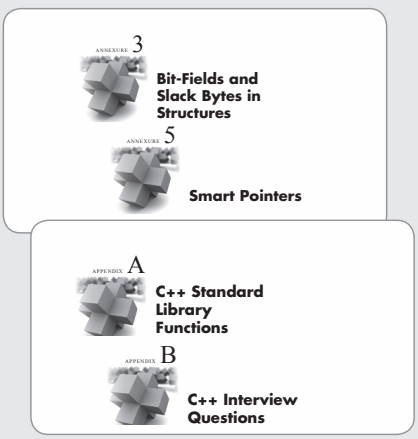
Glossary and Points to remember

A point-wise summary and glossary of terms are provided at the end of each chapter to help readers quickly recollect and memorize the important concepts explained in that chapter.

Test Your Skills

The book contains numerous (close to 1880) questions and a variety of chapter-end exercises including objective-type questions (fill in the blanks, true/false, and multiple choice questions) with answers, review questions, programming exercises, and find the output and error questions. These are provided to test readers' knowledge of the concepts, improve their programming skills as also help them practice and prepare for their examinations.





Annexures and Appendices

The annexures linked to various chapters of the book details some of the important concepts discussed in that particular chapter to further enhance the knowledge of the readers. Appendices on library functions and interview questions will help readers get better understanding and hands-on knowledge of the C++ concepts.

Preface

C++ is a general purpose, compiled programming language that stands second (only to C) in the most popular programming languages list. It is based on object-oriented concepts and can be used on several different software platforms. C++ programming language has been in use for a long time. Therefore there is a lot of information available on it and hence, it is easier to comprehend. Moreover, a sound understanding of C++ concepts facilitates in learning JAVA and similar programming languages.

In recent times, C++ is extensively used for implementing desktop applications, system programming, designing operating system kernels, developing embedded systems and web search engines, SQL servers, space probes, telephone switches, and entertainment software, to name a few. The important features that make C++ so exciting to use include its high performance, efficiency, and flexibility.

No student can learn to program just by reading a book; rather it is a skill that must be developed by practice. After learning the C++ concepts and their applications demonstrated through example programs, students will find a number of programming exercises at the end of each chapter which will help them hone their programming skills.

About the Book

Object Oriented Programming with C++ is designed as a textbook for undergraduate degree and diploma students of computer science engineering and information technology as well as postgraduate students of computer applications. The objective of this book is to introduce the concepts of C++ programming language and apply these concepts in solving real world problems. This will help readers get acquainted with the techniques and applications of C++ and also prepare them for taking up programming-based challenging tasks. The book is also useful as a reference and resource to computer professionals working in C++ and other similar languages.

Beginning with an introduction to programming languages and object-oriented programming concepts; the book goes on to elucidate the basics of C++ programming. It explains the various control and looping statements, functions, arrays, strings, pointers, structure, union and enumeration in the subsequent chapters. Further, it discusses the important constructs of C++ programming, namely, classes and objects, constructors and destructors, operator overloading, inheritance, polymorphism, templates, generics, and exception handling in detail. Finally, standard template library and object oriented analysis, design and development are discussed in separate chapters. The book also contains useful annexures to various chapters including user-defined header files, pointer declarations, bit-fields, volatile and restrict qualifiers, and smart pointers for additional information. Case studies, important library functions, and interview questions are also provided to supplement the text.

The book first explains the fundamentals of a concept using simple language, examples and illustrations, and then delves into its complexities and implementation aspects. Every chapter contains multiple programming examples to impart practically sound knowledge of the concepts learnt. All these programs have already been implemented and tested using Dev C++ and g++ compilers. To further enhance the understanding of the subject and the application and analytical ability of the students, there are numerous objective type, subjective type, and programming exercises at the end of each chapter. Thus, this text can be easily grasped by the readers.

Salient Features

The salient features of the book include:

- *Simple and lucid explanations* for important C++ concepts using block diagrams and examples for easy understanding.
- All programs in this book compiled on Dev C++ and g++ compilers.
- Plenty of *example programs provided along with their outputs* to help readers hone their programming skills.
- *Notes* and *Programming tips* to help readers keep in mind the critical concepts and develop error-free programs.
- *Case studies* interspersed within the text include programs which demonstrate the implementation of the concepts learnt in the various chapters.
- Abundant and variety of chapter-end exercises including *objective-type questions* (fill in the blanks, true/false, and multiple choice questions) *with answers, review questions, programming exercises,* and *find the error and output type questions* for self-check and practice.
- *Glossary* of key terms and point-wise *summary* at the end of each chapter to help readers quickly revise and memorize the important concepts learnt.
- *Interview questions* at the end of the book to help readers prepare for competitive examinations.

Organization of the Book

The book is divided into 17 chapters, 6 case studies, 5 annexures, and 2 appendices.

Chapter 1 provides an introduction to programming languages, different programming paradigms, concepts of OOP along with merits and demerits of object oriented programming languages. The chapter also gives a comparative study of some OOP languages and highlights the difference between C and C++.

Chapter 2 discusses the building blocks of the C++ programming language. The chapter includes identifiers, constants, variables, operators, type conversion, and casting supported by the C++ language.

Annexure 1 given at the end of this chapter covers user-defined header files.

Chapter 3 deals with the different types of decision control statements in C++ such as conditional branching statement, iterative statement, break statement, control statement, and jump statement.

Case Study 1 including representation of roman numbers and calculation of day and date of birth using a program illustrates the implementation of concepts discussed in Chapters 2 and 3.

Chapter 4 deals with declaring, defining, and calling functions. The chapter also discusses the storage classes, variable scope in C++, inline functions, and function overloading. It ends with an important concept of recursive functions.

Chapter 5 provides a detailed explanation of arrays that includes one, two, and multi-dimensional arrays. The operations that can be performed on such arrays are also explained.

Case Study 2 shows the applications of concepts discussed in Chapters 4 and 5 with the help of merge sort and quick sort examples.

Chapter 6 unleashes the concept of strings which are better known as character arrays. The chapter not only focuses on reading and writing strings but also explains various operations that can be used to manipulate the strings.

Chapter 7 presents a detailed overview of pointers, pointer variables, pointer arithmetic, so and so forth. The chapter also relates the use of pointers with arrays, strings and functions for writing better and efficient programs. The chapter ends with the discussions on dynamic memory management.

Annexure 2 on the process of deciphering pointer declarations is given after Chapter 7.

Chapter 8 deals with the two user-defined data types—structure and union. The chapter includes the use of structures and unions with pointers, arrays and functions so that the inter-connectivity between the programming techniques can be well understood. Enumerated data types are also explained in this chapter.

Annexure 3 discuss bit-fields and slack bytes in structures followed by ***Case Study 3*** on Josephus Problem shows the implementation of string operations.

Chapter 9 introduces the concept of classes and objects. It explains dynamic allocation of objects, static data members, nested, inline and friend functions, constant members, this pointer, empty and local classes.

Chapter 10 goes a step beyond declaration and definition of user defined classes. It unleashes the use of constructors and destructors. It also explains overloading of constructors and creating anonymous objects.

Case Study 4 covering the examples of Interpolation Search and Selection Sort presents the implementation of concepts covered in Chapters 9 and 10.

Chapter 11 is all about overloading unary and binary operators. It also discusses the conversion of a variable from one class type to another and from basic type to class type and vice versa.

Case Study 5 illustrates the concept of operator overloading discussed in Chapter 11.

Chapter 12 introduces inheritance in its various forms—single, hierarchical, hybrid, multi-level, and multiple. It provides a detailed explanation for different access specifiers, virtual functions, abstract classes, and object slicing. It also throws light on polymorphism and run time polymorphism through virtual functions. Abstract base classes, pure virtual functions, and virtual constructors and destructors are also covered in this chapter.

Chapter 13 discusses how data can be stored in files. The chapter explains the different types of files, and opening, processing and closing of files through a C++ program. These files are handled in text mode as well as binary mode for better clarity of the concepts. It also discusses sequential I/O functions and error handling during file operations.

Case Study 6 demonstrates the implementation of concepts discussed in Chapters 12 and 13.

Chapter 14 is about generic programming through templates. In this chapter, function and class templates have been discussed. It also gives a combo effect of templates and overloading, templates and inheritance, templates and static functions, templates and friends to name a few.

Chapter 15 presents the concepts of exception handling that can be used to make robust programs. The chapter demonstrates the application of exception handling in overloaded classes and functions, inherited classes, constructors or destructors. It also discusses various advantages and disadvantages of exception handling.

Chapter 16 elucidates the features of the standard template library. It also introduces some new features that have been added to the C++ language. These concepts include containers, iterators, algorithms, the string class, data type Boolean, keywords—mutable and explicit, namespaces, and finally the run time type identifiers.

Annexures 4 and 5 following this chapter briefs about the restrict and volatile qualifiers and smart pointers, respectively.

Chapter 17, the last chapter of the book, co-relates the use of objects and classes with the software development methodology. The chapter also gives an introduction of unified modelling language (UML).

The two appendices, namely, *Appendix A* discusses some of the important library functions in C++ and *Appendix B* provides frequently asked interview questions with answers and certain programming tips.

Online Resources

To aid teachers and students, the book is accompanied with online resources that are available at http://oupinheonline.com/book/thareja-object-oriented-programming-with-C/9780199459636. The content for the online resources are as follows.

For Faculty

- Chapter-wise PPTs
- Solutions Manual

For Students

- Projects
- MCQ test generator
- Model Question Papers (with answers)
- Introduction to graphics programming
- Solution to find the output and error questions
- Algorithms for basic programs
- Source codes of the programs given in the book
- Additional case studies on data structures and sorting
- Extra reading material on algorithms and flowcharts, slack bytes, bit level programming, macros, and ASCII chart

Acknowledgments

The writing of this textbook was a mammoth task for which a lot of help was required from many people. Fortunately, I have had wholehearted support of my family, friends, and fellow members of the teaching staff at Shyama Prasad Mukherji College, New Delhi.

My special thanks would always go to my parents, Mr Janak Raj Thareja and Mrs Usha Thareja, and my siblings, Pallav, Kimi, and Rashi, who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my son, Goransh, who has been very patient and cooperative in letting me realize my dreams. My sincere thanks goes to my uncle, Mr B.L. Theraja, for his inspiration and guidance in writing this book. I would like to acknowledge Er Udit Chopra for his technical assistance in designing and testing the programs.

I would like to express my gratitude to the following reviewers of this revised edition as well as the previous edition for their valuable suggestions and constructive feedback that helped in improving the book.

Ms Preeti Rai Jain
Miranda House, University of Delhi

Ms Seema Rani
Shyama Prasad Mukherji College, University of Delhi

Ms Pratibha Yadav
Shyama Prasad Mukherji College, University of Delhi

Mr Soumen Swarnakar
Netaji Subhash Engineering College, Garia, Kolkata

CH.V.K.N.S.N. Moorthy
Institute of Aeronautical Engineering (Autonomous), Hyderabad

Dr Anthonisan Arockiasamy
Saveetha School of Engineering, Saveetha University, Chennai

Nithya E.
Dr Ambedkar Institute of Technology, Bangalore

Last but not the least I would like to thank the editorial team at Oxford University Press, India for their help and support.

Comments and suggestions for the improvement of the book are welcome. Please send to me at reemathareja@gmail.com.

Reema Thareja

Detailed Contents

Features of the Book *iv*
Preface *vi*

1. Introduction to Object Oriented Programming (OOP) 1

Introduction *1*
1.1 Generation of Programming Languages *2*
1.1.1 First Generation: Machine Language *2*
1.1.2 Second Generation: Assembly Language *3*
1.1.3 Third Generation: High-Level Language *5*
1.1.4 Fourth Generation: Very High-Level Languages *8*
1.1.5 Fifth Generation Programming Language *9*
1.2 Programming Paradigms *10*
1.2.1 Monolithic Programming *10*
1.2.2 Procedural Programming *10*
1.2.3 Structured Programming *11*
1.2.4 Object Oriented Programming *12*
1.3 Features of Object Oriented Programming *14*
1.3.1 Classes *14*
1.3.2 Objects *15*
1.3.3 Method and Message Passing *15*
1.3.4 Inheritance *16*
1.3.5 Polymorphism: Static Binding and Dynamic Binding *17*
1.3.6 Containership *17*
1.3.7 Genericity *18*
1.3.8 Delegation *18*
1.3.9 Data Abstraction and Encapsulation *18*
1.4 Merits and Demerits of Object Oriented Programming Language *19*
1.5 Applications of Object Oriented Programming *20*
1.6 Differences Between Programming Languages *20*
1.7 C++ Compilers *22*

2. Basics of C++ Programming 27

Introduction to C++ *27*
2.1 History of C++ *27*
2.2 Structure of C++ Program *28*
2.3 Writing the First C++ Program *28*
2.4 Files Used in C++ Program *31*
2.4.1 Source Code File *31*
2.4.2 Header Files *31*
2.4.3 Object Files *33*
2.4.4 Binary Executable File *33*
2.5 Compiling and Executing C++ Programs *33*
2.6 Using Comments *34*
2.7 Tokens in C++ *35*
2.8 Character Set *35*
2.9 Keywords *36*
2.10 Identifier *36*
2.11 Data Types in C++ *37*
2.12 Variables *39*
2.12.1 Declaring Variables *39*
2.12.2 Initializing Variables *40*
2.12.3 Reference Variables *40*
2.13 Constants *41*
2.13.1 Integer Constant *41*
2.13.2 Floating Point Constant *42*
2.13.3 Character Constant *42*
2.13.4 String Constant *42*
2.13.5 Declaring Constants *43*
2.14 Input and Output Statements in C++ *43*
2.14.1 Streams *43*
2.14.2 Reading and Writing Characters and Strings *45*
2.14.3 Formatted Input and Output Operations *46*
2.15 Operators in C++ *52*
2.15.1 Arithmetic Operators *52*
2.15.2 Relational Operators *55*
2.15.3 Equality Operators *56*

2.15.4 Logical Operators *56*
2.15.5 Unary Operators *58*
2.15.6 Conditional Operators *60*
2.15.7 Bitwise Operators *61*
2.15.8 Assignment Operators *63*
2.15.9 Comma Operator *64*
2.15.10 Sizeof Operator *64*
2.15.11 Operator Precedence and Associativity *65*
2.16 Type Conversion and Type Casting *72*
2.16.1 Type Conversion *72*
2.16.2 Type Casting *74*
Annexure 1 – User-Defined Header Files 83

3. Decision Control and Looping Statements 85

Introduction to Decision Control Statements *85*
3.1 Conditional Branching Statements *85*
3.1.1 If Statement *86*
3.1.2 If-Else Statement *87*
3.1.3 If-Else-If Statement *90*
3.1.4 Switch Case Statement *96*
3.2 Iterative Statements *101*
3.2.1 While Loop *101*
3.2.2 Do-While Loop *105*
3.2.3 For Loop *108*
3.2.4 Selecting an Appropriate Loop *112*
3.3 Nested Loops *113*
3.4 Break Statement *125*
3.5 Continue Statement *126*
3.6 Goto Statement *128*
3.6.1 Key Points About Goto Statement *129*
3.7 Avoiding Usage of Break, Continue, and Goto Statements *129*
Case Study 1 141

4. Functions 145

Introduction *145*
4.1 Need For Functions *146*
4.2 Using Functions *147*
4.3 Function Declaration or Function Prototype *147*
4.4 Function Definition *149*
4.5 Function Call *150*
4.6 Return Statement *151*
4.7 Passing Parameters to the Function *153*
4.7.1 Call-By-Value *153*
4.7.2 Call-By-Address *155*
4.7.3 Call-By-Reference *156*
4.8 Default Arguments *161*
4.9 Return by Reference *163*
4.10 Passing Constants as Arguments *164*
4.11 Variables Scope *165*
4.11.1 Block Scope *165*
4.11.2 Function Scope *165*
4.11.3 Scope of the Program *166*
4.11.4 File Scope *167*
4.12 Storage Classes *167*
4.12.1 Auto Storage Class *167*
4.12.2 `Register` Storage Class *168*
4.12.3 `Extern` Storage Class *169*
4.12.4 Static Storage Class *170*
4.12.5 Comparison of Storage Classes *171*
4.13 Inline Functions *172*
4.13.1 Advantages and Disadvantages of Inline Functions *173*
4.13.2 Comparison of Inline Functions with Macros *173*
4.14 Function Overloading *174*
4.14.1 Matching Function Calls with Overloaded Functions *175*
4.14.2 Key Points About Function Overloading *176*
4.14.3 Functions that Cannot be Overloaded *178*
4.15 Recursive Functions *179*
4.15.1 Greatest Common Divisor *180*
4.15.2 Finding Exponents *181*
4.15.3 Fibonacci Series *182*
4.16 Recursion Versus Iteration *183*
4.17 Functions with Variable Number of Arguments *184*

5. Arrays 192

Introduction *192*
5.1 Declaration of Arrays *193*
5.2 Accessing Elements of the Array *194*
5.2.1 Calculating the Address of Array Elements *195*
5.3 Storing Values in Arrays *196*
5.3.1 Initialization of Arrays *196*
5.3.2 Inputting Values *197*
5.3.3 Assigning Values *197*

5.4 Calculating the Length of Array *198*
5.5 Operations that can be Performed on Arrays *198*
5.5.1 Traversal *199*
5.5.2 Insertion *205*
5.5.3 Deletion *206*
5.5.4 Merging *207*
5.5.5 Searching the Array Elements *209*
5.6 One-Dimensional Arrays for Inter Function Communication *214*
5.6.1 Passing Individual Elements *214*
5.6.2 Passing The Entire Array *215*
5.7 Two-Dimensional Arrays *218*
5.7.1 Declaration of Two-Dimensional Arrays *219*
5.7.2 Initialization of Two-Dimensional Arrays *221*
5.7.3 Accessing the Elements *221*
5.8 Operations on Two-Dimensional Arrays *225*
5.9 Two-Dimensional Arrays for Inter-Function Communication *228*
5.9.1 Passing a Row *228*
5.9.2 Passing the Entire Two-Dimensional Array *228*
5.10 Multi-Dimensional Arrays *232*
Case Study 2 239

6. Strings 244

Introduction *244*
6.1 Representation and Declaration of Strings *244*
6.1.1 Reading Strings *246*
6.1.2 Writing Strings *247*
6.2 String Taxonomy *248*
6.3 Strings Operations *249*
6.4 Character Manipulation Functions *258*
6.5 String Functions Defined In `String.h` Header File *259*
6.6 Array of Strings *267*

7. Pointers 282

Understanding Computer's Memory *282*
7.1 Defining Pointers *283*
7.2 Declaring Pointer Variables *284*
7.3 Pointer Expressions and Pointer Arithmetic *288*
7.4 Null Pointers *290*
7.5 Generic Pointers *291*
7.6 Passing Arguments to Function Using Pointers *291*
7.7 Pointers And Arrays *292*
7.8 Passing Array To Functions *297*
7.9 Differences Between Array Name and Pointers *299*
7.10 Pointers and Strings *300*
7.11 Array of Pointers *302*
7.12 Pointers and 2D Arrays *304*
7.13 Pointers and 3D Arrays *308*
7.14 Pointers to Functions *309*
7.14.1 Initializing Function Pointer *309*
7.14.2 Calling a Function using a Function Pointer *310*
7.14.3 Comparing Function Pointers *311*
7.14.4 Passing a Function Pointer as an Argument to a Function *311*
7.15 Array of Function Pointers *312*
7.16 Pointers to Pointers *313*
7.17 Constant Pointer *314*
7.18 Pointer to Constants *314*
7.19 Constant Pointer to a Constant *314*
7.20 Memory Allocation in C++ Programs *315*
7.21 Memory Usage *315*
7.22 Dynamic Memory Allocation *316*
7.22.1 Memory Allocation Process *316*
7.22.2 Allocating Memory using the New Operator *316*
7.22.3 Releasing the Used Space using the `Delete` Operator *317*
7.22.4 Alter the Size of Allocated Memory *318*
7.22.5 Advantages of `New/Delete` Operators Over `Malloc()/Free()` 319
7.22.6 Dynamically Allocating 2D Arrays *321*
Annexure 2 – Deciphering Pointer Declarations 329

8. Structure, Union, and Enumerated Data Types 333

Introduction *333*
8.1 Structure Declaration *333*

8.2 Typedef Declarations *334*
8.3 Initialization of Structures *335*
8.4 Accessing the Members of a Structure *336*
8.5 Copying and Comparing Structures *337*
8.6 Nested Structures *339*
8.7 Arrays of Structures *341*
8.8 Structures and Functions *344*
8.8.1 Passing Individual Members *344*
8.8.2 Passing The Entire Structure *345*
8.8.3 Returning Structures *346*
8.8.4 Passing Structures Through Pointers *348*
8.9 Self-Referential Structures *353*
8.10 C++ Extension to Structures *354*
8.11 Union *355*
8.11.1 Declaring a Union *355*
8.11.2 Accessing a Member of a Union *356*
8.11.3 Initializing Unions *356*
8.12 Unions Inside Structures *356*
8.13 Enumerated Data Types *357*
8.13.1 Enum Variables *359*
8.13.2 Assigning Values To Enumerated Variables *359*
8.13.3 Enumeration Type Conversion *359*
8.13.4 Comparing Enumerated Types *360*
8.13.5 Input or Output Operations on Enumerated Types *360*
Annexure 3 – Bit-Fields and Slack Bytes in Structures 367
Case Study 3 369

9. Classes and Objects 370

Introduction *370*
9.1 Specifying a Class *370*
9.1.1 Class Declaration *371*
9.1.2 Function Definition *372*
9.2 Creating Objects *373*
9.3 Accessing Object Members *374*
9.4 Nested Member Functions *374*
9.5 Making a Member Function Inline *375*
9.6 Memory Allocation for Class and Objects *380*
9.6.1 Memory Allocation for Static Data Members *381*
9.6.2 Static Member Functions *383*
9.6.3 Static Object *384*
9.7 Array of Objects *385*
9.8 Dynamic Memory Allocation for Array of Objects *385*
9.9 Objects as Function Arguments *388*
9.10 Returning Objects *389*
9.11 `This` Pointer *390*
9.12 Constant Parameters and Members *397*
9.13 Pointers within a Class *399*
9.14 Local Classes *401*
9.15 Nested Classes in C++ *402*
9.16 Complex Objects (Object Composition) *403*
9.17 Empty Classes *405*
9.18 Friend Function *411*
9.19 Friend Class *413*
9.20 Bit-Fields in Classes *418*
9.21 Pointers and Class Members *420*
9.21.1 Declaring and Assigning Pointer to Data Members of a Class *420*
9.21.2 Pointer to Member Functions *422*

10. Constructors and Destructors 430

Introduction *430*
10.1 Constructor *431*
10.2 Types of Constructors *432*
10.2.1 Dummy Constructor (Do Nothing Constructor) *432*
10.2.2 Default Constructor *432*
10.2.3 Parameterized Constructor *433*
10.2.4 Copy Constructor *433*
10.2.5 Dynamic Constructor *435*
10.3 Constructor with Default Arguments *435*
10.4 Constructor Overloading *437*
10.5 Destructors *451*
10.5.1 Important Features *452*
10.5.2 Interesting Points About Constructors and Destructors *457*
10.6 Object Copy *460*
10.7 Constant Objects *461*
10.7.1 Key Features of Constant Object *462*
10.8 Anonymous Objects *462*
10.8.1 Scope of Anonymous Objects *463*
10.8.2 Advantages of Anonymous Objects *463*
10.9 Anonymous Classes *464*
Case Study 4 471

11. Operator Overloading and Type Conversions 474

Introduction *474*
11.1 Scope of Operator Overloading *474*
11.2 Syntax for Operator Overloading *475*
11.3 Operators that can and cannot be Overloaded *476*
11.4 Implementing Operator Overloading *477*
11.5 Overloading Unary Operators *477*
11.5.1 Using a Member Function to Overload a Unary Operator *477*
11.5.2 Returning Object *478*
11.5.3 Returning a Nameless Object *479*
11.5.4 Using a Friend Function to Overload a Unary Operator *479*
11.5.5 Overloading the Prefix Increment and Decrement Operators *480*
11.5.6 Overloading the Post-Fix Increment and Post-Fix Decrement Operators *481*
11.6 Overloading Binary Operators *491*
11.7 Overloading Special Operators *501*
11.7.1 Overloading New and Delete Operators *502*
11.7.2 Overloading Subscript Operators `[]` and `()` `504`
11.7.3 Overloading Class Member Access Operator (`->`) *507*
11.7.4 Overloading Input and Output Operators *508*
11.8 Type Conversions *509*
11.8.1 Conversion from Basic to Class Type *509*
11.8.2 Conversion from Class to Basic Data Type *509*
11.8.3 Conversion from Class to Class Type *515*
Case Study 5 527

12. Inheritance and Run-Time Polymorphism 529

Introduction *529*
12.1 Defining Derived Classes *530*
12.2 Access Specifiers *530*
12.2.1 Inheriting Protected Members *531*
12.2.2 Inheriting the Class in Protected Mode *532*
12.3 Types of Inheritance *532*
12.4 Single Inheritance *532*
12.5 Constructors and Destructors in Derived Classes *536*
12.5.1 Invoking Constructors With Arguments *538*
12.6 Multi-Level Inheritance *540*
12.7 Constructor In Multi-Level Inheritance *541*
12.8 Multiple Inheritance *546*
12.9 Constructors and Destructors in Multiple Inheritance *547*
12.10 Ambiguity in Multiple Inheritance *549*
12.10.1 Solution for the Ambiguity Problem *549*
12.10.2 No Ambiguity In Single Inheritance *550*
12.11 Hierarchical Inheritance *550*
12.12 Constructors and Destructors in Hierarchical Inheritance *552*
12.13 Hybrid Inheritance *554*
12.14 Multi-Path Inheritance *564*
12.14.1 Problem in Multi-Path Inheritance or Diamond Problem *564*
12.15 Virtual Base Classes *564*
12.16 Object Slicing *566*
12.17 Pointers to Derived Class *567*
12.17.1 Upcasting, Downcasting, and Cross-Casting *569*
12.18 Run-Time Polymorphism *569*
12.19 Virtual Functions *570*
12.19.1 Run-Time Polymorphism Through Virtual Functions *570*
12.19.2 Rules For Virtual Functions *572*
12.20 Pure Virtual Functions *573*
12.21 Abstract Base Classes *574*
12.22 Concept of Vtables *577*
12.23 Virtual Constructor And Destructors *579*
12.23.1 Virtual Destructor *579*
12.24 Pros and Cons of Inheritance *580*

13. File Handling 592

Introduction *592*
13.1 Streams in C++ *592*
13.2 Classes for File Stream Operations *593*
13.3 Opening and Closing of Files *594*
13.3.1 Opening Files using Constructors *595*

13.3.2 Opening Files using Member Function *597*
13.3.3 Test For Errors *597*
13.4 Detecting the End-of-File *598*
13.5 File Modes *600*
13.6 File Pointers and their Manipulators *601*
13.6.1 Manipulating File Pointers *602*
13.6.2 Specifying The Offset *603*
13.7 Types of Files *604*
13.7.1 ASCII Text Files *604*
13.7.2 Binary Files *605*
13.8 Sequential Input and Output Functions *605*
13.8.1 Get() and Put() *606*
13.8.2 Read() and Write()Functions *609*
13.9 Error Handling During File Operations *617*
13.10 Accepting Command Line Arguments *618*
Case Study 6 629

14. Templates 631

Introduction *631*
14.1 Use of Templates *632*
14.2 Function Templates *632*
14.2.1 Templates Versus Macros *634*
14.2.2 Guidelines for Using Template Functions *634*
14.3 Class Template *643*
14.4 Class Templates and Friend Function *650*
14.5 Templates and Static Variables in C++ *653*
14.6 Class Templates and Inheritance *655*
14.7 Class Template with Operator Overloading *663*
14.8 Pros and Cons of Templates *665*

15. Exception Handling 672

Introduction *672*
15.1 Exception Handling *673*
15.1.1 Multiple Catch Statements *675*
15.1.2 Catch all Exceptions *676*
15.1.3 Exceptions in Invoked Function *677*
15.1.4 Stack Unwinding *679*
15.1.5 Rethrowing Exception *679*
15.1.6 Restricting the Exceptions that can be Thrown *680*
15.1.7 Catching Class Type as Exceptions *682*
15.2 Exceptions in Constructors and Destructors *687*
15.3 Exceptions in Operator Overloaded Functions *689*
15.4 Exceptions and Inheritance *690*
15.5 Exceptions and Templates *691*
15.6 Handling Uncaught Exceptions *698*
15.7 Standard Exceptions *701*
15.8 Advantages of Exception Handling *703*
15.9 Word of Caution *704*

16. STL and New Features in C++ 713

Introduction *713*
16.1 Containers *713*
16.2 Algorithms *715*
16.3 Iterators *717*
16.4 Using Containers *718*
16.4.1 Vector *718*
16.4.2 Dequeue *719*
16.4.3 List *720*
16.4.4 Maps *722*
16.5 String Class *725*
16.6 Data Type Boolean *728*
16.6.1 Uses of Boolean Data Type *729*
16.6.2 Applications of Operators *729*
16.7 `Wchar_t` Data Type *730*
16.8 Run-Time Type Information *731*
16.8.1 `Static_cast` Operator *731*
16.8.2 `Const_cast Operator` *732*
16.8.3 `Reinterpret_cast Operator` 733
16.8.4 `Dynamic_cast` Operator *734*
16.8.5 `Typeid` Operator *735*
16.9 `Explicit` Keyword *737*
16.10 Mutable Keyword *739*
16.11 Namespaces *740*
16.11.1 Nested Namespaces *741*
16.11.2 Unnamed Namespaces *742*
16.11.3 Similarity and Dissimilarity with Classes *743*
16.11.4 Classes Within Namespaces *743*
16.12 Operator Keywords *744*
16.13 Specifying Header Files *744*

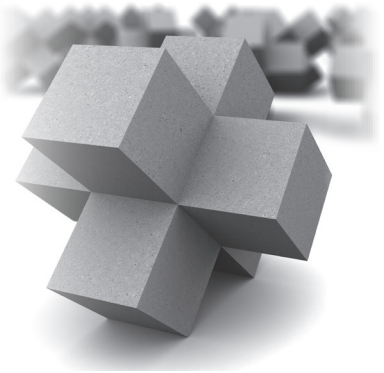
Annexure 4 – Volatile and Restrict Qualifiers 753
Annexure 5 – Smart Pointers 755

17. Object-Oriented System Analysis, Design, and Development 757

Introduction *757*
17.1 Traditional Software Development Process *758*
17.2 Building High-Quality Software *758*
17.3 Object Oriented Software Development Methodology *759*
17.4 Object-Oriented Systems Development *759*
17.4.1 Object-Oriented Analysis *760*
17.4.2 Object-Oriented Design *761*
17.4.3 Prototyping *762*
17.4.4 Objectives of OO Analysis and Design *763*
17.4.5 Tools in OO Analysis and OO Design *763*
17.4.6 Implementing Component-Based Development *764*
17.5 Unified Modelling Language *765*
17.5.1 Class Diagrams *765*
17.5.2 Object Diagram *766*
17.5.3 Component Diagram *766*
17.5.4 Deployment Diagram *766*
17.5.5 Use Case Diagram *767*
17.5.6 Sequence Diagram *767*
17.5.7 State Chart Diagram *767*
17.5.8 Activity Diagram *768*

Appendix A – C++ Standard Library Functions 772
Appendix B – C++ Interview Questions 775
Index 781
About the Author 784

CHAPTER 1



Introduction to Object Oriented Programming (OOP)

KEY Concepts

Generation of programming languages • Programming paradigms • Concepts of OOP • Merits, demerits, and applications of OOP • Differences between C and C++

INTRODUCTION

A programming language is a language specifically designed to express computations that can be performed by the computer. Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or can be used as a mode of human communication.

Usually, programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. The term 'programming language' usually refers to high-level languages such as BASIC, C, C++, COBOL, FORTRAN, ADA, and PASCAL, to name a few. All these languages have a unique set of keywords (words that the language understands) and a special syntax for organizing program instructions.

While high-level programming languages are easy for us to read and understand, the computer understands the machine language that consists only of numbers. Different types of central processing unit (CPU) have their own unique machine languages.

Assembly language is a type of language that exists in between machine languages and high-level languages. Assembly languages are similar to machine languages, but are much easier to program in because they allow a programmer to substitute names for numbers.

However, irrespective of what language the programmer uses, the program written using any programming language has to be converted into machine language so that the computer can understand the same. There are two ways to do this—compile the program or interpret the program.

The determination of the language is dependent on the following factors:

- The type of computer (microcontroller, microprocessor, etc.) on which the program has to be executed.
- The type of program (system program, application program, etc.).

- The expertise of the programmer. That is, the proficiency level of a programmer in a particular language.

For example, FORTRAN is particularly good for processing numerical data but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs but it is not as flexible as C language. C++ goes one step ahead of C by incorporating powerful object oriented features but it is complex and difficult to learn.

These programming languages have slowly evolved with time to be more user-friendly and task-oriented. In the next section, we will study the different generations of programming languages to understand their distinguishing features.

1.1 GENERATION OF PROGRAMMING LANGUAGES

We now know that programming languages are the primary tools for creating software. As of now, hundreds of programming languages exist in the market, some more used than others and each claiming to be the best. However, in the 1940s when computers were being developed, there was just one language—machine language.

The concept of generations of programming languages (also known as levels) is closely connected to the advances in technology. The five generations of programming languages include machine language, assembly language, high-level language (also known as the third generation language or 3GL), very high-level language (also known as the fourth generation language or 4GL), and fifth generation language that includes artificial intelligence.

1.1.1 First Generation: Machine Language

The first program was programmed using the machine language. This is the lowest level of programming language and is the only language that a computer understands. All the commands and data values are expressed using 0s and 1s, corresponding to the *off* and *on* electrical states in a computer.

In the 1950s, each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as *add* and *subtract*. Although there were similarities between each of the machine languages, a computer could not understand programs written in another machine language (refer to Fig. 1.1).

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of ones and zeroes.

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 0s and 1s. Although machine language programs are typically displayed with the *binary* numbers represented in *octal* (base 8) or *hexadecimal* (base 16) number systems, these programs are not easy for humans to read, write, or debug.

The main advantage of machine language is that the execution of the code is very fast and efficient since it is directly executed by the CPU. However, on the downside, machine language is difficult to learn and is far more difficult to edit if errors occur. Moreover, if we want to store some instructions in the memory at some location, then all the instructions after the insertion point would have to be moved down to make room in the memory to accommodate the new instructions. In addition, the code written in machine language is not portable, and to transfer the code to a different computer, it needs to be completely rewritten since the machine language for one computer could be significantly different from that for another computer. Architectural considerations make portability a tough issue to resolve. Table 1.1 lists the advantages and disadvantages of first generation language.

This is an example of a machine language program that will add two numbers and find their average. It is in hexadecimal notation instead of binary notation because that is how the computer presented the code to the programmer. The program was run on a VAX/VMS computer, a product of the Digital Equipment Corporation.

```
                                                    000 0000A   0000
                                                    000 0000F   0008
                                                    000 0000B   0008
                                                                0008
                                                                0058
                                                                00Θ0
FF55   CF   FF54   CF   FF53   CF   C1              00A9
       FF24   CF   FF27   CF   D2   C7              00CC
                                                    00E4
                                                    010D
                                                    013D
```

Figure 1.1 A machine language program

Table 1.1 Advantages and disadvantages of first generation language

Advantages	Disadvantages
• Code can be directly executed by the computer. • Execution is fast and efficient. • Programs can be written to efficiently utilize memory.	• Code is difficult to write. • Code is difficult to understand by other people. • Code is difficult to maintain. • There is more possibility for errors to creep in. • It is difficult to detect and correct errors. • Code is machine dependent and thus non-portable.

1.1.2 Second Generation: Assembly Language

Second-generation programming languages (2GLs) comprise the assembly languages. Assembly languages are symbolic programming languages that use symbolic notations to represent machine language instructions. These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since it is close to machine language, assembly language is also a low-level language. Nearly all computer systems have an assembly language available for use.

Assembly language developed in the mid-1950s was a great leap forward. It used symbolic codes, also known as *mnemonic* codes, which are easy-to-remember abbreviations, rather than numbers. Examples of these codes include ADD for add, CMP for compare, and MUL for multiply.

Assembly language programs consist of a series of individual statements or instructions to instruct the computer what to do. Basically, an assembly language statement consists of a label, an operation code, and one or more *operands*.

Labels are used to identify and refer instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation to be performed, such as *move*, *add*, *subtract*, or *compare*. The operand specifies the register or the location in the main memory where the data to be processed is located.

However, like machine language, the statement or instruction in assembly language will vary from machine to machine, because the language is directly related to the internal architecture of the computer and is not designed to be machine independent. This makes the code written in assembly language less portable, as the code written to be executed on one machine will not run on machines from a different manufacturer or sometimes even the same manufacturer.

Nevertheless, the code written in assembly language will be very efficient in terms of execution time and main memory usage, as the language is similar to computer language.

Programs written in assembly language need a translator, often known as the assembler, to convert them into machine language. This is because the computer will understand only the language of 0s and 1s. It will not understand mnemonics such as ADD and SUB.

The following instructions are part of an assembly language code to illustrate addition of two numbers:

```
MOV AX,4      Stores the value 4 in the AX register of CPU
MOV BX,6      Stores the value 6 in the BX register of CPU
ADD AX,BX     Adds the contents of AX and BX registers; stores the result in AX register
```

Although it is much easier to work with assembly language than with machine language, it still requires the programmer to think on the machine's level. Even today, some programmers use assembly language to write those parts of applications where speed of execution is critical; for example, videogames, but most programmers have switched to 3GL or 4GL even to write such codes. Table 1.2 lists the advantages and disadvantages of using second generation language.

Table 1.2 Advantages and disadvantages of second generation language

Advantages	Disadvantages
• It is easy to understand. • It is easier to write programs in assembly language than in machine language. • It is easy to detect and correct errors. • It is easy to modify. • It is less prone to errors.	• Code is machine dependent and thus non-portable. • Programmers must have a good knowledge of the hardware and internal architecture of the CPU. • The code cannot be directly executed by the computer.

Assembler

Since computers can execute only codes written in machine language, a special program, called the assembler, is required to convert the code written in assembly language into an equivalent code in machine language, which contains only 0s and 1s. The working of an assembler is shown in Fig. 1.2; it can be seen that the assembler takes an assembly language program as input and gives a code in machine language (also called object program) as output. There is a one-to-one correspondence between the assembly language code and the machine language code. However, if there is an

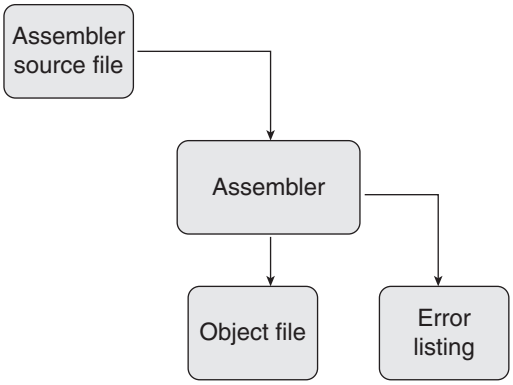


Figure 1.2 Assembler

error, the assembler gives a list of errors. The object file is created only when the assembly language code is free from errors. The object file can be executed as and when required.

1.1.3 Third Generation: High-level Language

Third-generation programming languages are a refinement of 2GLs. The second generation brought logical structure to software. The third generation was introduced to make the languages more programmer friendly.

The 3GLs spurred the great increase in data processing that occurred in the 1960s and 1970s. In these languages, the program statements are not closely related to the internal characteristics of the computer. Hence, these languages are often referred to as high-level languages.

Note An assembler only translates an assembly program into machine language. The result is an object file that can be executed. However, the assembler itself does not execute the object file.

In general, a statement written in a high-level programming language will expand into several machine language instructions. This is in contrast to assembly languages, where one statement would generate one machine language instruction. 3GLs made programming easier, efficient, and less prone to errors.

High-level languages fall somewhere between natural languages and machine languages. 3GLs include FORTRAN and COBOL, which made it possible for scientists and entrepreneurs to write programs using familiar terms instead of obscure machine instructions.

The widespread use of high-level languages in the early 1960s changed programming into something quite different from what it had been. Programs were written in languages that were more English-like, making them more convenient to use and giving the programmer more time to address a client's problems.

Although 3GLs relieve the programmer of demanding details, they do not provide the flexibility available in low-level languages. However, a few high-level languages such as C and FORTH combine some of the flexibility of assembly languages with the power of high-level languages, but these languages are not well suited to programmers at the beginner level.

Some high-level languages were specifically designed to serve a specific purpose (such as controlling industrial robots or creating graphics), whereas other languages were flexible and considered to be general purpose. Most programmers preferred to use general-purpose high-level languages such as BASIC, FORTRAN, Pascal, COBOL, C++, or Java to write the code for their applications.

Again, a translator is needed to translate the instructions written in a high-level language into the computer-executable machine language. Such translators are commonly known as interpreters and compilers. Each high-level language has many compilers, and there is one for each type of computer.

For example, the machine language generated by one computer's C compiler is not the same as the machine language of some other computer. Therefore, it is necessary to have a C compiler for each type of computer on which the C programs are to be executed.

The 3GLs make it easy to write and debug a program and give a programmer more time to think about its overall logic. Programs written in such languages are portable between machines. For example, a program written in standard C can be compiled and executed on any computer that has a standard C compiler. Table 1.3 provides the advantages and disadvantages of 3GLs.

Table 1.3 Advantages and disadvantages of third generation languages

Advantages	Disadvantages
• The code is machine independent. • It is easy to learn and use the language. • There are few errors. • It is easy to document and understand the code. • It is easy to maintain the code. • It is easy to detect and correct errors.	• Code may not be optimized. • The code is less efficient. • It is difficult to write a code that controls the CPU, memory, and registers.

Compiler

A compiler is a special type of program that transforms the source code written in a programming language (the *source language*) into machine language, which uses only two digits—0 and 1 (the *target language*). The resultant code in 0s and 1s is known as the object *code*. The object code is used to create an executable program.

Therefore, a compiler (refer to Fig. 1.3) is used to translate the source code from a high-level programming language to a lower-level language (e.g., assembly language or machine code). There is a one-to-one correspondence between the high-level language code and machine language code generated by the compiler.

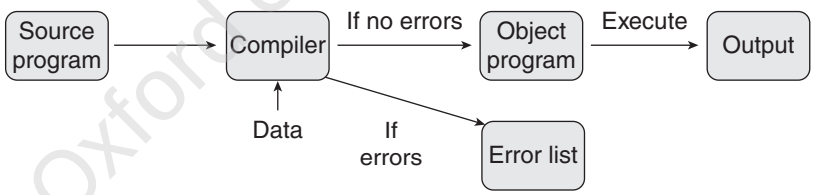


Figure 1.3 Compiler

If the source code contains errors, then the compiler will not be able to do its intended task. Errors that limit the compiler in understanding a program are called syntax errors. Examples of syntax errors are spelling mistakes, typing mistakes, illegal characters, and use of undefined variables. The other type of error is the logical error, which occurs when the program does not function accurately. Logical errors are much harder to locate and correct than syntax errors. Whenever errors are detected in the source code, the compiler generates a list of error messages indicating the type of error and the line in which the error has occurred. The programmer makes use of this error list to correct the source code.

The work of a compiler is only to translate the human-readable source code into a computer-executable machine code. It can locate syntax errors in the program (if any) but cannot fix it. Unless the syntactical error is rectified, the source code cannot be converted into the object code.

Each high-level language has a separate compiler. A compiler can translate a program in one particular high-level language into machine language. For a program written in some other programming language, a compiler for that specific language is needed.

How compilers work?

Compilers like other programs reside on the secondary storage. To translate a source code into its equivalent machine language code, the computer first loads the compiler and the source program from the secondary memory to the main memory. The computer then executes the compiler along with the source program as its input. The output of this execution is an object file which is also stored on the secondary storage. Whenever the program has to be executed, the computer loads the object file into memory and executes it. This means that it is not necessary to compile the program every time it has to be executed. Compilation will be needed only the source is modified.

Interpreter

Like the compiler, the interpreter executes instructions written in a high-level language. Basically, a program written in a high-level language can be executed in any of the two ways—by compiling the program or by passing the program through an interpreter.

The compiler translates instructions written in a high-level programming language directly into machine language; the interpreter, on the other hand, translates the instructions into an intermediate form, which it then executes. The interpreter takes one statement of high-level code, translates it into the machine level code, executes it, and then takes the next statement and repeats the process until the entire program is translated.

Note An interpreter not only translates the code into machine language but also executes it.

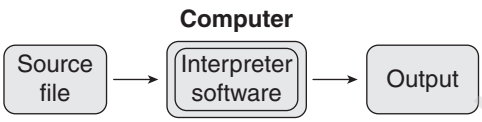


Figure 1.4 Interpreter

Figure 1.4 shows an interpreter that takes a source program as its input and gives the output. This is in contrast with the compiler, which produces an object file as the output of the compilation process. Usually, a compiled program executes faster than an interpreted program. Moreover, since there is no object file saved for future use, users will have to reinterpret the entire program each time they want to execute the code.

Overall, compilers and interpreters both achieve similar purposes, but they are inherently different as to how they achieve that purpose. The differences between compilers and interpreters are given in Table 1.4.

Table 1.4 Differences between compiler and an interpreter

Compiler	Interpreter
• It translates the entire program in one go. • It generates error(s) after translating the entire program. • Execution of code is faster. • An object file is generated. • Code need not be recompiled every time it is executed. • It merely translates the code. • It requires more memory space (to save the object file).	• It interprets and executes one statement at a time. • It stops translation after getting the first error. • Execution of code is slower as every time reinterpretation of statements has to be done. • No object file is generated. • Code has to be reinterpreted every time it is executed. • It translates as well as executes the code. • It requires less memory space (no object file).

Linker

Software development in the real world usually follows a modular approach as given in structured programming (discussed in Section 1.2.3). In this approach, a program is divided into various (smaller) modules as it is easy to code, edit, debug, test, document, and maintain them. Moreover, a module written for one program can also be used for another program. When a module is compiled, an object file of that module is generated.

Once the modules are coded and tested, the object files of all the modules are combined together to form the final executable file. Therefore, a linker, also called a link editor or binder, is a program that combines the object modules to form an executable program (refer to Fig.1.5). Usually, the compiler automatically invokes the linker as the last step in compiling a program.

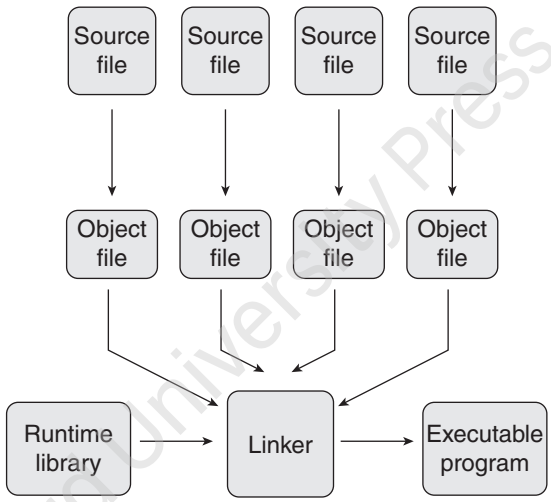


Figure 1.5 Role of linker

Loader

A loader is a special type of program that copies programs from a storage device to the main memory, where they can be executed. The functionality and complexity of most loaders are hidden from the users.

1.1.4 Fourth Generation: Very High-level Languages

With each generation, programming languages started becoming easier to use and more similar to natural languages. 4GLs are a little different from their prior generation because they are non-procedural. While writing a code using a procedural language, the programmer has to tell the computer how a task is done—add this, compare that, do this if the condition is true, and so on—in a very specific step-by-step manner. In striking contrast, while using a non-procedural language, programmers define what they want the computer to do but they do not supply all the details of how it has to be done. For example, in a procedural language like C++, to show the details of all the students we write programs that contains instructions specifying how the details are to be fetched and displayed but the same task in SQL (a non procedural language) can be written in a single sentence (select * from Students)

Although there is no standard rule that defines a 4GL, certain characteristics of such languages include the following:

- The instructions of the code are written in English-like sentences.
- They are non-procedural, so users concentrate on the 'what' instead of the 'how' aspect of the task.

- The code written in a 4GL is easy to maintain.
- The code written in a 4GL enhances the productivity of programmers, as they have to type fewer lines of code to get something done. A programmer supposedly becomes 10 times more productive when he/she writes the code using a 4GL than using a 3GL.

A typical example of a 4GL is the query language, which allows a user to request information from a database with precisely worded English-like sentences. A query language is used as a database user interface and hides the specific details of the database from the user. For example, when working with structured query language (SQL), the programmer just needs to remember a few rules of *syntax* and *logic*, and therefore, it is easier to learn than COBOL or C.

Let us take an example in which a report needs to be generated. The report displays the total number of students enrolled in each class and in each semester. Using a 4GL, the request would look similar to the following:

```
TABLE FILE ENROLLMENT
SUM STUDENTS BY SEMESTER BY CLASS
```

Therefore, we see that a 4GL is very simple to learn and work with. The same task if written in C or any other 3GL would require multiple lines of code.

The 4GLs are still evolving, which makes it difficult to define or standardize them. The only downside of a 4GL is that it does not make efficient use of a machine's resources. However, the benefit of executing a program quickly and easily far outweighs the extra costs of running it.

1.1.5 Fifth Generation Programming Language

Fifth-generation programming languages (5GLs) are centred on solving problems using the constraints given to a program rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages form a part of the 5GLS. These languages are widely used in artificial intelligence research. Another aspect of a 5GL is that it contains visual tools to help develop a program. Typical examples of 5GLs include Prolog, OPS5, Mercury, and Visual Basic.

Therefore, taking a forward leap, 5GLs are designed to make the computer solve a given problem without the programmer. While working with a 4GL, programmers have to write a specific code to do a work, but with a 5GL, they only have to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or an algorithm to solve them.

In general, 5GLs were built upon LISP, many originating on the LISP machine such as ICAD. There are also many frame languages such as KL-ONE.

In the 1990s, 5GLs were considered the wave of the future, and some predicted that they would replace all other languages for system development (except the low-level languages). During the period ranging from 1982 to 1993, Japan carried out extensive research on and invested a large amount of money into their fifth-generation computer systems project, hoping to design a massive computer network of machines using these tools. However, when large programs were built, the flaws of the approach became more apparent. Researchers began to observe that given a set of constraints defining a particular problem, deriving an efficient algorithm to solve it is itself a very difficult problem. All factors could not be automated and some still require the insight of a programmer.

However, today the fifth-generation languages are pursued as a possible level of computer language. Software vendors across the globe currently claim that their software meets the visual 'programming' requirements of the 5GL concept.

Knowing about the evolution of programming languages is just not enough until and unless we have a sound background knowledge about structure and style of writing of programs. The next section therefore talks about programming paradigms so that you can have an insight into how object oriented programming is better than the traditional paradigms.

1.2 PROGRAMMING PARADIGMS

A programming paradigm is a fundamental style of programming that defines how the structure and basic elements of a computer program will be built. The style of writing programs and the set of capabilities and limitations that a particular programming language has depends on the programming paradigm it supports. While some programming languages strictly follow a single paradigm, others may draw concepts from more than one. The sweeping trend in the evolution of high-level programming languages has resulted in a shift in programming paradigm. These paradigms, in sequence of their application, can be classified as follows:

- Monolithic programming—emphasizes on finding a solution
- Procedural programming—lays stress on algorithms
- Structured programming—focuses on modules
- Object-oriented programming—emphasizes on classes and objects
- Logic-oriented programming—focuses on goals usually expressed in predicate calculus
- Rule-oriented programming—makes use of 'if-then-else' rules for computation
- Constraint-oriented programming—utilizes invariant relationships to solve a problem

Each of these paradigms has its own strengths and weaknesses and no single paradigm can suit all applications. For example, for designing computation intensive problems, procedure-oriented programming is preferred; for designing a knowledge base, rule-based programming would be the best option; and for hypothesis derivation, logic-oriented programming is used. In this book, we will discuss only first four paradigms. Among these paradigms, object oriented paradigms supersede to serve as the architectural framework in which other paradigms are employed.

1.2.1 Monolithic Programming

Programs written using monolithic programming languages such as assembly language and BASIC consist of global data and sequential code. The global data can be accessed and modified (knowingly or mistakenly) from any part of the program, thereby, posing a serious threat to its integrity.

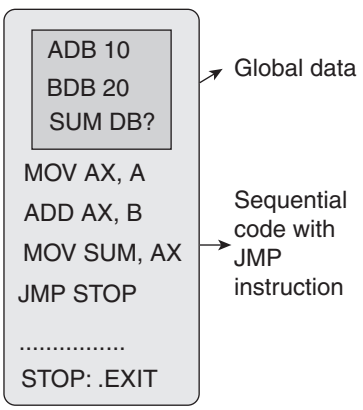


Figure 1.6 Structure of a monolithic program

A sequential code is one in which all instructions are executed in the specified sequence. In order to change the sequence of instructions, jump statements or 'goto' statements are used. Figure 1.6 shows the structure of a monolithic program.

As the name suggests, monolithic programs have just one program module as such programming languages do not support the concept of subroutines. Therefore, all the actions required to complete a particular task are embedded within the same application itself. This not only makes the size of the program large but also makes it difficult to debug and maintain.

For all these reasons, monolithic programming language is used only for very small and simple applications where reusability is not a concern.

1.2.2 Procedural Programming

In procedural languages, a program is divided into *n* number of subroutines that access global data. To avoid repetition of code, each subroutine

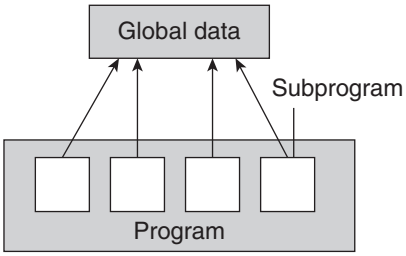


Figure 1.7 Structure of a procedural program

performs a well-defined task. A subroutine that needs the service provided by another subroutine can call that subroutine. Therefore, with 'jump', 'goto', and 'call' instructions, the sequence of execution of instructions can be altered. Figure 1.7 shows the structure of a procedural language.

FORTRAN and COBOL are two popular procedural programming languages.

Advantages

- The only goal is to write correct programs
- Programs were easier to write as compared to monolithic programming

Disadvantages

- Writing programs is complex.
- No concept of reusability.
- Requires more time and effort to write programs.
- Programs are difficult to maintain.
- Global data is shared and therefore may get altered (mistakenly).

1.2.3 Structured Programming

Structured programming, also referred to as modular programming, was first suggested by mathematicians, Corrado Bohm and Guiseppe Jacopini. It was specifically designed to enforce a logical structure on the program to make it more efficient and easier to understand and modify. Structured programming was basically defined to be used in large programs that require large development team to develop different parts of the same program.

Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules. This allows the code to be loaded into memory more efficiently and also be reused in other programs. Modules are coded separately and once a module is written and tested individually, it is then integrated with other modules to form the overall program structure (refer to Fig.1.8).

Structured programming is, therefore, based on modularization which groups related statements together into modules. Modularization makes it easier to write, debug, and understand the program.

Ideally, modules should not be longer than a page. It is always easy to understand a series of 10 single-page modules than a single 10-page program.

For large and complex programs, the overall program structure may further require the need to break the modules into subsidiary pieces. This process continues until an individual piece of code can be written easily.

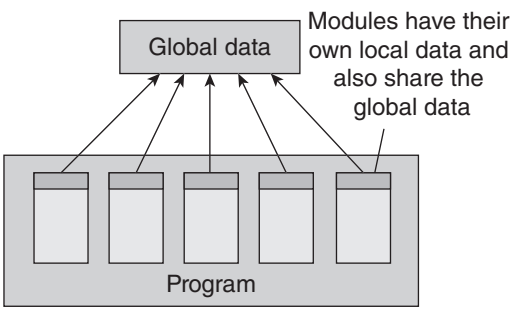


Figure 1.8 Structured program

Almost every modern programming language similar to C, Pascal, etc. supports the concepts of structured programming. Even OOP can be thought of as a type of structured programming. In addition to the techniques of structured programming for writing modules, it also focus on structuring its data.

In structured programming, the program flow follows a simple sequence and usually avoids the use of 'goto' statements. Besides sequential flow, structured programming also supports selection and repetition as mentioned here.

- Selection allows for choosing any one of a number of statements to execute, based on the current status of the program. Selection statements contain keywords such as if, then, end if, or switch that help to identify the order as a logical executable.
- In repetition, a selected statement remains active until the program reaches a point where there is a need for some other action to take place. It includes keywords such as repeat, for, or do… until. Essentially, repetition instructs the program as to how long it needs to continue the function before requesting further instructions.

Advantages

- The goal of structured programming is to write correct programs that are easy to understand and change.
- Modules enhance programmer's productivity by allowing them to look at the big picture first and focus on details later.
- With modules, many programmers can work on a single, large program, with each working on a different module.
- A structured program takes less time to be written than other programs. Modules or procedures written for one program can be reused in other programs as well.
- Each module performs a specific task.
- Each module has its own local data.
- A structured program is easy to debug because each procedure is specialized to perform just one task and every procedure can be checked individually for the presence of any error. In striking contrast, unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. Their logic is cluttered with details and, therefore, difficult to follow.
- Individual procedures are easy to change as well as understand. In a structured program, every procedure has meaningful names and has clear documentation to identify the task performed by it. Moreover, a correctly written structured program is self-documenting and can be easily understood by another programmer.
- More emphasis is given on the code and the least importance is given to the data.
- Global data may get inadvertently changed by any module using it.
- Structured programs were the first to introduce the concept of functional abstraction.

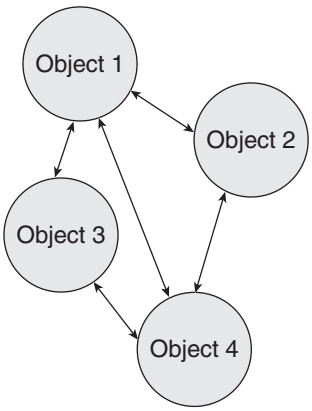
Note Functional abstraction allows a programmer to concentrate on what a function (or module) does and not on how it does.

Disadvantages

- Structured programming is not data-centered.
- Global data is shared and therefore may get inadvertently modified.
- Main focus is on functions.

1.2.4 Object Oriented Programming

With the increase in size and complexity of programs, there was a need for a new programming paradigm that could help to develop maintainable programs. To implement this, the flaws in previous paradigms had to be corrected. Consequently, OOP was developed. It treats data as a critical element in the program development and restricts its flow freely around the system. We have seen that monolithic, procedural, and structured programming paradigms are task-based as they focus on the actions the software should accomplish. However, the object oriented paradigm is task-based and data-based. In this paradigm, all the relevant data and tasks are grouped together in entities known as objects (refer to Fig. 1.9).



Objects of a program interact by sending messages to each other

Figure 1.9 Object oriented paradigm

For example, consider a list of numbers stored in an array. The procedural or structured programming paradigm considers this list as merely a collection of data. Any program that accesses this list must have some procedures or functions to process this list. For example, to find the largest number or to sort the numbers in the list, we needed specific procedures or functions to do the task. Therefore, the list was a passive entity was it is maintained by a controlling program rather than having the responsibility of maintaining itself.

However, in the object oriented paradigm, the list and the associated operations are treated as one entity known as an object. In this approach, the list is considered an object consisting of the list, along with a collection of routines for manipulating the list. In the list object, there may be routines for adding a number to the list, deleting a number from the list, sorting the list, etc.

The striking difference between OOP and traditional approaches is that the program accessing this list need not contain procedures for performing tasks; rather, it uses the routines provided in the object. In other words, instead of sorting the list as in the procedural paradigm, the program asks the list to sort itself.

Therefore, we can conclude that the object oriented paradigm is task-based (as it considers operations) as well as data-based (as these operations are grouped with the relevant data).

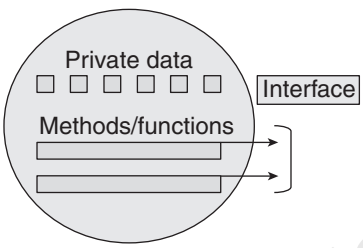


Figure 1.10 Object

Figure 1.10 represents a generic object in the object oriented paradigm. Every object contains some data and the operations, methods, or functions that operate on that data. While some objects may contain only basic data types such as characters, integers, floating types, the other object, the other objects on the other hand may incorporate complex data types such as trees or graphs.

Programs that need the object will access the object's methods through a specific interface. The interface specifies how to send a message to the object, that is, a request for a certain operation to be performed.

For example, the interface for the list object may require that any message for adding a new number to the list should include the number to be added. Similarly, the interface might also require that any message for sorting specify whether the sort should be ascending or descending. Hence, an interface specifies how messages can be sent to the object.

Note OOP is used for simulating real world problems on computers because real world is made of objects.

The striking features of OOP include the following:

- The programs are data-centered.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.
- Data is hidden and not accessible by external functions.
- New data and functions can be easily added as and when required.
- Follows a bottom-up approach for problem solving.

Top-down vs Bottom-up approach

While top-down approach follows a stepwise refinement by decomposing the algorithm into manageable modules, the bottom-up approach on the other hand defines a module and then groups together several modules to form a new higher level module.

In a top-down approach, we start from an abstract design and then at each step, this design is refined into more concrete levels until a level is reached that requires no further refinement Therefore, this approach is highly appreciated for ease in documenting the modules, generation of test cases, implementation of code, and debugging. However, it is also criticized because the sub-modules are analysed in isolation without concentrating on their communication with other modules or on reusability of components and little attention is paid to data, thereby ignoring the concept of information hiding.

While the bottom-up approach allows information hiding as it first identifies what has to be encapsulated within a module and then provides an abstract interface to define the module's boundaries as seen from the clients. But all this is difficult to be done in a strict bottom-up strategy. Some top-down activities need to be performed for this. Whether the top-down strategy should be followed or a bottom-up is a question that can be answered depending on the application at hand.

1.3 FEATURES OF OBJECT ORIENTED PROGRAMMING

The object oriented language must support mechanisms to define, create, store, manipulate objects, and allow communication between objects. In this section, we will read about the underlying concepts of OOP. These are as follows:

- Classes
- Objects
- Methods
- Message passing
- Inheritance
- Polymorphism
- Containership
- Genericity
- Reusability
- Delegation
- Data Abstraction and Encapsulation

1.3.1 Classes

Almost every language has some basic data types such as int, float, long, and so on, but not all real world objects can be represented using these built-in types. Therefore, OOP, being specifically designed to solve real world problems, allows its users to create user defined data types in the form of classes.

A class is used to describe something in the world, such as occurrences, things, external entities, and so on. A class provides a template or a blueprint that describes the structure and behaviour of a set of similar objects. Once we have the definition for a class, a specific instance of the class can be easily created. For example, consider a class student. A student has attributes such as roll number, name, course, and marks. The operations that can be performed on its data may include 'get_details', 'set_details', 'edit_details', and so on (refer to Fig. 1.11). Therefore, we can say that a class describes one or more similar objects.

```
class student
{
     private:
        int roll_no;
        char name[20];
        float marks;
     public:
        get_details();
        show_details();
};
```

Figure 1.11 A sample student class

It must be noted that this data and the set of operations that we have given here can be applied to all students in the class. When we create an instance of a student, we are actually creating an object of class student. Therefore, once a class is declared, a programmer can create any number of objects of that class.

Note Classes define properties and behaviour of objects.

Therefore, a class is a collection of objects of similar type. It is a user-defined data type that behaves same as the built-in data types. This can be realized by ensuring that the syntax of creating an object is same as that of creating an int variable. For example, to create an object (`stud`) of class student, we write

```
student stud;
```

Note Defining a class does not create any object. Objects have to be explicitly created by using the syntax as follows:

```
class-name object-name;
```

1.3.2 Objects

In the previous section, we have taken an example of student class and have mentioned that a class is used to create instances, known as objects. Therefore, if student is a class, then all the 60 students in a course (assuming there are maximum 60 students in a particular course) are the objects of the student class. Therefore, all students such as Aditya, Chaitanya, Deepti, and Esha are objects of the class.

Hence, a class can have multiple instances.

Every object contains some data and functions (also called methods) as shown in Fig. 1.12. These methods store data in variables and respond to messages that they receive from other objects by executing their methods (procedures).

Note While a class is a logical structure, an object is a physical actuality.

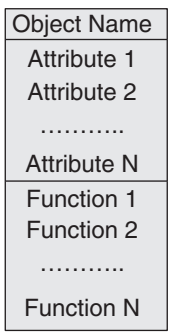


Figure 1.12 Representation of an object

1.3.3 Method and Message Passing

A method is a function associated with a class. It defines the operations that the object can execute when it receives a message. In object oriented language, only methods of the class can access and manipulate the data stored in an instance of the class (or object). Figure 1.13 shows how a class is declared using its data members and member functions.

Every object of the class has its own set of values. Therefore, two distinguishable objects can have the same set of values. Generally, the set of values that the object takes at a particular time is known as the *state* of the object. The state of the object can be changed by applying a particular method. Table 1.5 shows some real world objects along with their data and operations.

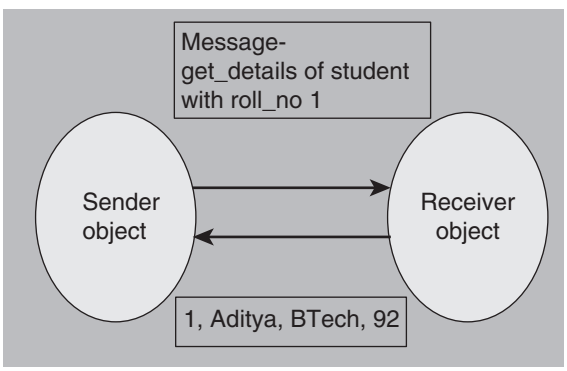


Figure 1.13 Objects sending messages

Table 1.5 Objects with data and functions

Object	Data or attributes	Functions or methods
Person	Name, age, sex	Speak(), walk(), listen(), write()
Vehicle	Name, company, model, capacity, colour	Start(), stop(), accelerate()
Polygon	Vertices, border, colour	Draw(), erase
Account	Type, number, balance	Deposit(), withdraw(), enquire()
City	Name, population, area, literacy rate	Analyse, data(), display()
Computer	Brand, resolution, price	Processing(), display(), printing()

Note An object is an instance of a class which can be uniquely identified by its name. Every object has a state which is given by the values of its attributes at a particular time.

Two objects can communicate with each other through messages. An object asks another object to invoke one of its methods by sending it a message. In Fig. 1.13 in which a sender object is sending a message to the receiver object to get the details of a student. In reply to the message, the receiver sends the results of the execution to the sender.

In the figure, sender has asked the receiver to send the details of student having `roll_no 1`. This means that the sender is passing some specific information to the receiver so that the receiver can send the correct and precise information to the sender. The data that is transferred with the message is called parameters. Here, `roll_no 1` is the parameter.

Therefore, we can say that messages that are sent to other objects consist of three aspects—the receiver object, the name of the method that the receiver should invoke, and the parameters that must be used with the method.

1.3.4 Inheritance

Inheritance is a concept of OOP in which a new class is created from an existing class. The new class, often known as a sub-class, contains the attributes and methods of the parent class (the existing class from which the new class is created).

The new class, known as sub-class or derived class, inherits the attributes and behaviour of the pre-existing class, which is referred to as super-class or parent class (refer to Fig. 1.14). The inheritance relationship of sub- and super classes generates a hierarchy. Therefore, inheritance relation is also called 'is-a' relation. A sub-class not only has all the states and behaviours associated with the super-class but has other specialized features (additional data or methods) as well.

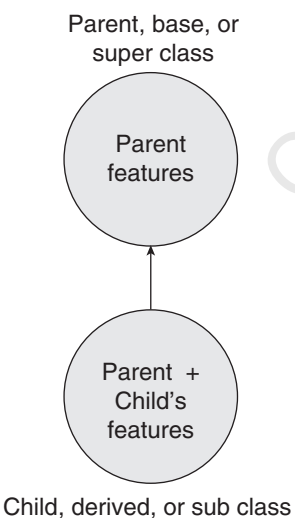


Figure 1.14 Inheritance

The main advantage of inheritance is the ability to reuse the code. When we want a specialized class, we do not have to write the entire code for that class from scratch. We can inherit a class from a general class and add the specialized code for the sub-class. For example, if we have a class student with following members:

```
Properties:  roll_number, name, course, and marks
Methods:     get_details, set_details
```

We can inherit two classes from the class `student`, namely, `under_graduate` students and `post_graduate` students (refer to Fig. 1.15). These two classes will have all the properties and methods of class students and in addition to that, will have even more specialized members.

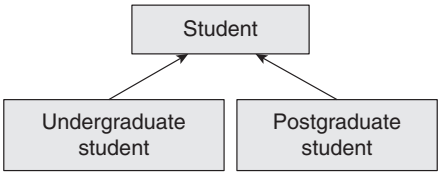


Figure 1.15 Example for Inheritance

When a derived class receives a message to execute a method, it finds the method in its own class. If it finds the method, then it simply executes it. If the method is not present, it searches for that method in its super class. If the method is found, it is executed, otherwise, an error message is reported. A sub class can inherit properties and methods from multiple parent classes. This is called multiple inheritance.

Note A sub class can inherit properties and methods from multiple parent classes. This is called multiple inheritance.

1.3.5 Polymorphism: Static Binding and Dynamic Binding

Polymorphism, one of the essential concepts of OOP, refers to having several different forms. While inheritance is related to classes and their hierarchy, polymorphism, on the other hand, is related to methods.

Polymorphism is a concept that enables the programmers to assign a different meaning or usage to a variable, function, or an object in different contexts.

When polymorphism is applied on variables, the variable with a given name may be allowed to have different forms. The program will then decide which form to use. For example, variable '`roll_no`' of a class student may be an integer (number) or an alphanumeric character (combination of numbers and alphabets). The program can be coded to distinguish between the two forms of the variable so that it can be handled in its own way.

Polymorphism can also be applied to a function in such a way that depending on the parameters it is given, a particular form of the function can be selected for execution. For example, if the roll number of the student is an integer, then its corresponding function will be executed. In case it consists of alphanumeric characters, another function having the same name will be executed. This type of polymorphism is called function overloading.

Polymorphism can also be applied to operators. For example, we know that operators can be applied only on basic data types that the programming language supports. Therefore, `a + b` will give the result of adding `a` and `b`. If `a = 2` and `b = 3`, then `a + b = 5`. When we overload the `+` operator to be used with strings, then `str1 + str2` gives the result `str2` concatenated with `str1`. Therefore, if `str1 = "Oxford"` and `str 2 = "University"` then `str1 + str2 = "Oxford University"`.

All types of polymorphism we have discussed so far are better known as compile time polymorphism or static binding. Dynamic binding or late binding, also known as run time polymorphism, is a feature that enables programmers to associate a function call with a code at the execution (or run) time. For example, if we have a function `print_result()` in class student, then both the inherited classes, under graduate and post graduate students will also have the same function implemented in their respective classes. Now, when there is a call to `print_result()`,

```
ptr_to_student-> print_result();
```

then, the decision regarding which version to call—the one in under graduate class or the one in post graduate class—will be taken at the execution time. Hence, the name.

Note Binding means associating a function call with the corresponding function code to be executed in response to the call.

1.3.6 Containership

The ability of a class to contain object(s) of one or more classes as member data. For example, class `One` can have an object of class `Two` as its data member. This would allow the object of class `One`

to call the public functions of class `Two`. Here, class `One` becomes the container, whereas class `Two` becomes the contained class.

Containership is also called composition because as in our example, class `One` is composed of class `Two`. In OOP, containership represents a 'has-a' relationship.

1.3.7 Genericity

To reduce code duplication and generate short, simpler code, C++ supports the use of generic codes (or templates) to define the same code for multiple data types. This means that a C++ function or a class can perform similar operations on different data types like integers, float, and double. This means that a generic function can be invoked with arguments of any compatible type.

Generic programs, therefore, act as a model of function or class that can be used to generate functions or classes. During program compilation, C++ compiler generates one or more functions or classes based on the specified template. Figure 1.16(a) and (b) clarifies this concept.

This discussion states that generic programming is a technique of programming in which a general code is written first. The code is instantiated only when need arises for specific types (provided as parameters).

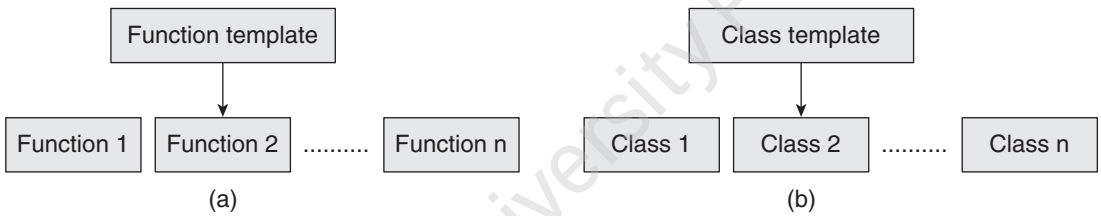


Figure 1.16 (a) Generated functions (b) Generated classes

Reusability

Reusability means developing codes that can be reused either in the same program or in different programs. C++ gives due importance to building programs that are reusable. Reusability is attained through inheritance, containership, polymorphism, and genericity.

1.3.8 Delegation

To provide maximum flexibility to programmers and to allow them to generate a reusable code, object oriented languages also support delegation, also known as composition or containership. In composition, an object can be composed of other objects and thus, the object exhibits a `'has-a'` relationship.

In delegation, more than one object is involved in handling a request. The object that receives the request for a service, delegates it to another object called its delegate. The property of delegation emphasizes on the ideology that a complex object is made of several simpler objects. For example, our body is made up of brain, heart, hands, eyes, ears, etc., the functioning of the whole body as a system rests on correct functioning of the parts it is composed of. Similarly, a car has a wheel, brake, gears, etc. to control it.

Delegation differs from inheritance in that two classes that participate in inheritance share an `'is-a'` relationship; however, in delegate, they have a `'has-a'` relationship.

1.3.9 Data Abstraction and Encapsulation

Data abstraction refers to the process by which data and functions are defined in such a way that only essential details are revealed and the implementation details are hidden. The main focus of data abstraction is to separate the interface and the implementation of a program. For example, as users of television sets, we can switch it on or off, change the channel, set the volume, and add external

devices such as speakers and CD or DVD players without knowing the details about how its functionality has been implemented. Therefore, the internal implementation is completely hidden from the external world.

Similarly, in OOP languages, classes provide public methods to the outside world to provide the functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.

Data encapsulation, also called data hiding, is the technique of packing data and functions into a single component (class) to hide implementation details of a class from the users. Users are allowed to execute only a restricted set of operations (class methods) on the data members of the class. Therefore, encapsulation organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines three access levels for data variables and member functions of the class. These access levels specify the access rights, explained as follows.

- Any data or function with access level as public can be accessed by any function belonging to any class. This is the lowest level of data protection.
- Any data or function with access level protected can be accessed only by that class or by any class that is inherited from it.
- Any data or function with access level private can be accessed only by the class in which it is declared. This is the highest level of data protection

Note Creating a new data type using encapsulated items that is well suited for an application is called data abstraction.

1.4 MERITS AND DEMERITS OF OBJECT ORIENTED PROGRAMMING LANGUAGE

OOP offers many benefits to program developers and users. It not only provides a solution for many problems associated with software development and its quality but also enhances programmer productivity and reduces maintenance cost. Some key advantages of OOP include the following:

- Elimination of redundant code through inheritance (by extending existing classes)
- Higher productivity and reduced development time due to reusability of the existing modules
- Secure programs as data cannot be modified or accessed by any code outside the class
- Real world objects in the problem domain can be easily mapped objects in the program
- A program can be easily divided into parts based on objects
- The data-centred design approach captures more details of a model in a form that can be easily implemented
- Programs designed using OOP are expandable as they can be easily upgraded from small to large systems
- Message passing between objects simplifies the interface descriptions with external systems
- Software complexity becomes easily manageable
- With polymorphism, behaviour of functions, operators, or objects may vary depending upon the circumstances
- Data abstraction and encapsulation hides implementation details from the external world and provides it a clearly defined interface
- OOP enables programmers to write easily extendable and maintainable programs
- OOP supports code reusability to a great extent

However, the down side of OOP include the following:

- Programs written using object oriented languages have greater processing overhead as they demand more resources
- Requires more skills to learn and implement the concepts
- Beneficial only for large and complicated programs
- Even an easy to use software when developed using OOP is hard to be build
- OOP cannot work with existing systems
- Programmers must have a good command in software engineering and programming methodology

1.5 APPLICATIONS OF OBJECT ORIENTED PROGRAMMING

No doubt, the concepts of object oriented technology have changed the way of thinking, analyzing, planning, and implementing software. Software or applications developed using this technology are not only efficient but also easy to upgrade. Therefore, programmers and software engineers all over the world have shown their keen interest in developing applications using OOP. As a result, there has been a constant increase in areas where OOP has been successfully implemented. Some of these areas include the following:

- Designing user interfaces such as work screens, menus, windows, and so on
- Real-time systems
- Simulation and modelling
- Compiler design
- Client server system
- Object oriented databases
- Object oriented distributed database
- Artificial intelligence—expert systems and neural networks
- Parallel programming
- Decision control systems
- Office automation systems
- Networks for programming routers, firewalls, and other devices
- Computer-aided design (CAD) systems
- Computer-aided manufacturing (CAM) systems
- Computer animation
- Developing computer games
- Hypertext and hypermedia

1.6 DIFFERENCES BETWEEN PROGRAMMING LANGUAGES

Most of you have learnt programming in C. To appreciate the power of C++, let us first compare C++ with C language which is not an object oriented language and then try to understand how C++ is different from other object oriented languages.

Table 1.6 summarizes the differences between C and C++ and Table 1.7 highlights the differences between commonly used object oriented languages.

Table 1.6 Differences between C and C++

C	C++
• Procedural language • Does not support virtual functions • Does not support polymorphism • Does not support operator overloading	• Uses top-down approach to build complex programs • Does not support namespaces • Object oriented language

(*Contd*)

Table 1.6 (*Contd*)

C	C++
• Supports virtual functions • Supports polymorphism • Supports operator overloading • Uses bottom-up approach to build complex programs • Supports namespaces • Can have multiple declarations for global variables • Printf() and scanf() functions in stdio.h file are used for I/O • Difficult to determine which function can and cannot modify data • Allows main() to be called through other functions • All variables must be defined at the beginning of the function (or scope) • Does not support inheritance • Does not support exception handling • Uses malloc(), calloc(), and free() for dynamically allocating or de-allocating memory	• A character constant is automatically elevated to an integer • Identifiers cannot start with two or more consecutive underscores, but may contain them in other positions. • Cannot have multiple declarations for global variables • Objects cin and cout of iostream are used for input or output • Easy mapping between data and functions • Does not allow main() to be called through other functions • Variables can be defined at any location but before their first use. • Supports inheritance • Supports exception handing • Uses new and delete operators for dynamically allocating or de-allocating memory • A character constant is not elevated to integer • Identifiers are not allowed to contain two or more consecutive underscores in any position

Table 1.7 Comparison between commonly used object oriented languages

Attributes	EIFFEL	C++	JAVA	SMALLTALK
Static typing	Statically typed	Statically typed but supports C-style 'casts' which may lead to violation of type rules	Statically typed but needs dynamic typing for generic container structures	Dynamically typed
Proprietary status	Open and standardized	Open, ANSI standard	Licensed from Sun	Not standardized
Compilation technology	Combination of interpretation and compilation	Compiled	Interpreted and on the fly compilation	Initially interpreted but currently mix of interpretation and compilation
Efficiency of code	Fast executable	Fast executable	Performance problems	Executables require a 'Smalltalk image'

(*Contd*)

Table 1.7 (*Contd*)

Attributes	EIFFEL	C++	JAVA	SMALLTALK
Multiple inheritance	Supported and widely used	Supported but not widely used because of performance problems	Single inheritance	Single inheritance
Understandability	Clear and simple	Complex syntax	Complex syntax	In between simple and complex
Garbage collection	Supported automatically	Not supported	Supported automatically	Supported automatically
Encapsulation	Supported	Supported	Poor support	Supported
Binding (early or late)	Early	Both	Late	Late

1.7 C++ COMPILERS

In this section we will discuss some C++ compilers that are widely used today.

Clang Clang is a C++ compiler developed primarily by Apple. It supports various ISO (International Standards Organization) C++ language standards. This compiler has been released under the BSD license.

MinGW-w64 MinGW-w64 provides the library files, header files, and runtime support needed for the `GNU` C++ compilers to run on a Windows system. The w64 means that the project support files that allow users to create 64-bit programs in addition to 32-bit ones. Another appealing feature of MinGW-w64 is that it provides cross-compilers that allows users to compile a Windows program from a Linux system or vice versa. Applications generated using MinGW are faster than those generated by the Cygwin32 system and are free from the encumbrances of the GNU license.

Microsoft Visual Studio Express 2013 Visual Studio Express 2013 can be used on Windows desktop as well as on Windows Phone. Visual Studio Express is a package that has an integrated development environment (IDE), compilers for C++, C# and Visual Basic. However, the Express version does not include all the tools and features of the full Visual Studio, like the full MSDN library, resource editor, macro assembler, etc.

x86 Open64 compiler system It is a high performance code generation tool designed for high performance parallel computing workloads. It is a version of the `Open64` compiler suite that requires Linux and has been tuned for AMD processors. The C++ compiler conforms to the ISO C++ 98 standards and supports 32-bit and 64-bit code generation, vector and scalar code generation. It also comprises of an optimizer that supports a huge variety of optimizations (global, loop transformation, vectorization, feedback-directed, inter-procedural analysis, etc), multi-threading. The compiler creates a strong foundation for building robust, high performance parallel code. It also has an optimized AMD Core Math Library and documentation.

Open Source Watcom/OpenWatcom C/C++ Compiler An open source and free compiler which generates code for Win32, Windows 3.1 (Win16), OS/2, Netware NLM, MSDOS (16-bit and 32-bit protected mode), etc. The compiler includes support for C++ Standard Template Library.

Digital Mars C/C++ Compiler (Symantec C++ Replacement) Digital Mars C++ supports compiling programs for Win32, Windows 3.1, MSDOS, and 32-bit extended MSDOS. If the target

machine does not have a floating point processor the programmers can still link the floating point emulation into their programs. The compiler supports the C++ definition specified in The Annotated C++ Reference Manual (ARM) and the enhanced language features of AT&T version 3.0. Features like templates, nested classes, nested types, exception handling, and runtime type identification, command line and GUI versions, tutorials, sample code, online updates, and much more all are supported by this compiler.

Bloodshed Dev-C++ Compiler A Win32 IDE that includes the egcs C++ compiler and GNU debugger from the Mingw32 environment along with an editor and other facilities to make program development using the Mingw32 gcc compiler easier on Windows. This compiler also comprises of an installer for application programs.

DJGPP C++ Compilers It is a development system based on GNU C++ compiler that generates 32-bit MSDOS executables. It is a complete system that comprises of IDEs, graphics libraries, lexical analyser generators (flex), parser generators (bison), text processing utilities (like grep, sed), a program maintenance utility (i.e. make), a dos extender, etc.

Cygwin C++ Compilers The compiler generates Win32 GUI and console applications. It comes with source code for the compiler, libraries, and tools. The compiler makes it easy for programmers to distribute their source code (while compiling and linking with their libraries).

Sun Studio It is a high-performance, optimizing compiler for the Solaris OS on SPARC and ×86/×64 platforms. It provides multi platform support and comprises of command-line tools plus a NetBeans-based Integrated Development Environment (IDE) for application performance analysis and debugging of mixed source language applications.

Borland C++ provides a programming environment for MS-DOS and Microsoft Windows. It has an IDE and is considered as a successor to Turbo C++. Borland has a library that contains a set of classes to make it easier to develop professional graphical Windows applications as DOS applications.

Turbo C++ Turbo C++ is a utility tool that helps programmers to code their C++ programs easily and effectively. Although Turbo C++ lacks some of the advanced features but it includes all features that any user might need to execute their programs. On the positive side, it is a free software which is simple to use and has an intuitive interface. However, on the downside it has very few advanced features.

Points to Remember

- If a procedure is formally defined, it must be implemented using some formal language, and such a language is often known as a programming language.
- Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or used as a mode of human communication.
- Machine language was used to program the first stored-program computer systems. This is the lowest level of programming language.
- Assembly languages are symbolic programming languages that use symbolic notation to represent machine-language instructions.
- 5GLs are centred on solving problems using constraints given to the program.
- Object oriented programming (OOP) emphasizes on classes and objects.
- Programs written using monolithic programming languages such as assembly language and BASIC consist of global data and sequential code.
- In procedural languages, a program is divided into *n* number of subroutines that access global data.
- Structured programming employs a top-down approach in which the overall program structure is broken down into separate modules.
- OOP treats data as a critical element in the program development and restricts its flow freely around the system.
- A class provides a template or a blueprint that describes the structure and behaviour of a set of similar objects.

Glossary

Assembler System software that translates a code written in assembly language into machine language

Compiler A special type of program that transforms source code written in a third programming language into machine language

Data abstraction Creating a new data type using encapsulated items that is well suited for an application

Data encapsulation It is also called data hiding, and is the technique of packing data and functions into a single component (class) to hide implementation details of a class from users

Functional abstraction A technique that allows a programmer to concentrate on what a function (or module) does and not on how it does

Inheritance A concept of object oriented programming in which a new class is created from an existing class

Linker A program that combines object modules to form an executable program

Loader A program that copies programs from a storage device to main memory, where they can be executed

Method Function associated with a class

Multiple inheritance A technique that allows a sub class to inherit properties and methods from multiple parent classes

Object An instance of a class

Polymorphism A concept that enables programmers to assign a different meaning or usage to a variable, function, or an object in different contexts

Programming language A language specifically designed to express computations that can be performed the computer

Programming paradigm A fundamental style of programming that defines how the structure and basic elements of a computer program will be built

Repetition A technique that allows a selected statement to remain active until the program reaches a point where there is a need for some other action to take place

Sequential code Code in which all the instructions are executed in the specified sequence one by one

Selection A technique that allows for choosing any one of a number of statements to execute, based on the current status of the program

Exercises

Fill in the Blanks

1. Programming languages have a vocabulary of _____ and _____ for instructing a computer to perform specific tasks.
2. Assembly language uses _____ to write programs.
3. An assembly language statement consists of a _____, _____, and ____.
4. _____ are used to identify and reference instructions in the program.
5. The output of an assembler is a _____ file.
6. A typical example of a 4GL is the _____.
7. Examples of a 5GL include _____, _____, and ____.
8. _____ defines the structure of a program.
9. _____ programming emphasizes on classes and objects.
10. Logic-oriented programming focus on _____ expressed in _____.
11. Two examples of languages that support monolithic programming paradigm are ____ and _____.
12. ____ and ____ statements are used to change the sequence of execution of instructions.
13. FORTRAN and COBOL are two popular _____ programming languages.
14. Functional abstraction was first supported by _____ programming.
15. An object contains ____ and _____.
16. _____ paradigm supports bottom-up approach of problem solving.
17. _____ provides a template that describes the structure and behaviour of an object.
18. While _____ is a logical structure, ____ is a physical actuality.
19. State defines the _____.
20. The data that is transferred with the message is called _____.
21. A message consists of _____, _____, and _____.

22. Inheritance relation is also called as _____ relation.
23. _____ is related to classes and their hierarchy.
24. Polymorphism is related to _____.
25. Any data or function with access level _____ can be accessed by any function belonging to any class.

State True or False

1. A programming language provides a blueprint to write a program to solve a particular problem.
2. Machine language is the lowest level of language.
3. Code written in machine language is not portable.
4. Compiler not only translates the code into machine language but also executes it.
5. An interpreted program executes faster than a compiled program.
6. Nonprocedural code that illustrates the 'how' aspect of the task is a feature of 3GL.
7. Constraint-based programming is used for hypothesis derivation.
8. In monolithic paradigm, global data can be accessed and modified from any part of the program.
9. Monolithic program has two modules.
10. Monolithic programs are easy to debug and maintain.
11. Structured programming is based on modularization.
12. Object oriented programming supports modularization.
13. Structured programming heavily used goto statements.
14. Modules enhance the programmer's productivity.
15. A structured program takes more time to be written than other programs.
16. The interface specifies how to send a message to the object.
17. OOP does not support modularization.
18. A class is a user-defined data type.
19. Once a class is declared, a programmer can create maximum 10 objects of that class.
20. Polymorphism means several different forms.
21. Any data or function with access level private can be accesses only by that class or by any class that is inherited from it.
22. OOP helps to develop secure programs.
23. It is difficult to manage software complexity in object oriented programs.
24. Programs written using object oriented languages have greater processing overhead.

Multiple Choice Questions

1. Which language is good for processing numerical data?
   (a) C (b) C++
   (c) FORTRAN (d) Java
2. Which is the fastest and the most efficient language?
   (a) Machine level
   (b) Assembly
   (c) High level
   (d) Artificial intelligence
3. FORTRAN, COBOL, and Pascal are examples of which generation language?
   (a) First (b) Second
   (c) Third (d) Fourth
4. In which generation language does the code comprise instructions written in English-like sentences?
   (a) First (b) Second
   (c) Third (d) Fourth
5. Which feature is affected by programming paradigm?
   (a) Style of programming
   (b) Capabilities
   (c) Limitations
   (d) All of these
6. Which programming paradigm utilizes invariant relationships to solve a problem?
   (a) Rule-based
   (b) Constraint-based
   (c) Structured
   (d) Object oriented
7. Which is the preferred paradigm for designing a knowledge base?
   (a) Rule-based
   (b) Constraint-based
   (c) Structured
   (d) Object oriented
8. Which type of programming does not support sub routines?
   (a) Monolithic (b) Structured
   (c) Rule-based (d) Object oriented

9. C and Pascal belong to which type of programming language?
   (a) Monolithic (b) Structured
   (c) Logic-oriented (d) Object oriented
10. Which paradigm holds data as a priority?
   (a) Monolithic (b) Structured
   (c) Logic-oriented (d) Object oriented
11. Two objects can communicate with each other through ____
   (a) Classes (b) Objects
   (c) Methods (d) Messages
12. Which concept enables programmers to assign a different meaning or usage to a variable, function, or an object in different contexts?
   (a) Inheritance
   (b) Message passing
   (c) Polymorphism
   (d) Abstraction
13. Which access level allows data and functions to be accessed only by the class in which it is declared?
   (a) Public (b) Private
   (c) Protected (d) None of these
14. In which of these applications is OOP applied?
   (a) CAD (b) CAM
   (c) Compiler design (d) All of these

Review Questions

1. What is a programming language?
2. Write a short note on generation of programming languages.
3. Differentiate between a compiler and an interpreter.
4. Differentiate between syntax errors and logic errors.
5. What do you understand by the term 'programming paradigm'?
6. Discuss any three programming paradigms in detail.
7. How is structured programming better than monolithic programming?
8. Describe the special characteristics of monolithic programming.
9. Explain how functional abstraction is achieved in structured programming.
10. Which programming paradigm is data-based and why?
11. Explain the concepts of OOP.
12. Differentiate between a class and an object.
13. How is a message related with a method?
14. Inheritance helps to make reusable code. Justify.
15. What do you understand by the term 'polymorphism'?
16. Why is data abstraction and encapsulation called the building blocks of OOP?
17. Explain the three levels of data protection.
18. What are the merits and demerits of OOP?

Answers

Fill in the Blanks

1. Syntax, semantics; **2.** Mnemonic codes; **3.** Label, an operation code, and one or more operands; **4.** Labels; **5.** Object; **6.** Query language; **7.** Prolog, OPS5, and Mercury; **8.** Programming paradigm; **9.** Object oriented; **10.** Goals, predicate; **11.** assembly language and BASIC; **12.** goto and call; **13.** 3GL; **14.** Structured; **15.** Data and methods; **16.** OOP; **17.** Class; **18.** Class, object; **19.** values of its attributes at a particular time; **20.** parameter; **21.** the receiver object, the name of the method that the receiver should invoke, and the parameters that must be used with the method; **22.** Is-a; **23.** Inheritance; **24.** methods; **25.** public

True or False

1. False	**2.** True	**3.** True	**4.** False	**5.** False	**6.** False	**7.** False	**8.** True
9. False	**10.** False	**11.** True	**12.** True	**13.** False	**14.** True	**15.** False	**16.** True
17. False	**18.** True	**19.** False	**20.** True	**21.** False	**22.** True	**23.** False	**24.** True

Multiple Choice Questions

1. (c); **2.** (a); **3.** (c); **4.** (d); **5.** (d); **6.** (b); **7.** (a); **8.** (a); **9.** (b); **10.** (d); **11.** (d); **12.** (c); **13.** (b); **14.** (d)