Object Oriented Programming with C++

SECOND EDITION

Sourav Sahay
Lead Consultant
Capgemini
Detroit, Michigan

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
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

First Edition Published in 2006
Second Edition Published in 2012

ISBN-13: 978-0-19-806530-2
ISBN-10: 0-19-806530-2

Typeset in Times New Roman
by Recto Graphics, Delhi 110096
Printed in India by Adage Printers (P) Ltd., Noida 201301 U.P.

Preface to the First Edition

C++ made its advent in the early 1980s and enabled programmers to write their programs the object-oriented way. For this reason, the language quickly gained popularity and became a programming language of choice. Despite the development of a number of competing object-oriented languages including Java, C++ has successfully maintained its position of popularity.

C++ starts where C stops. C++ is a superset of C. All the language features of C language appear in C++ with little or no modification. Over and above such features, C++ provides a number of extra features, which provide the language its object-oriented character.

About the Book

The continued popularity of C++ has led to considerable literature. Innumerable books, journals, magazines, and articles have been written on C++. So, why another book on C++?

The aim of the book is to thoroughly explain all aspects of the language constructs provided by C++. While doing full justice to the commonly explained topics of C++, the book does not neglect the advanced and new concepts of C++ that are not widely taught.

This book is a power-packed instruction guide for Object-Oriented Programming and C++. The purpose of this book is two-fold:

- To clarify the fundamentals of the Object-Oriented Programming System
- To provide an in-depth treatment of each feature and language construct of C++

This book emphasizes the Object-Oriented Programming System—its benefits and its superiority over the conventional Procedure-Oriented Programming System.

This book starts directly with C++ since the common features of C and C++ are anyway covered in books on C language. Each feature of C++ is covered from the practical point of view. Instead of brief introductions, this book gives an in-depth explanation of the rationale and proper use of each object-oriented feature of C++.

To help the readers assimilate the large volume of knowledge contained in this book, an adequate number of example programs, well-designed diagrams, and analogies with the real world have been given. Some program examples given in this book are abstract in nature to help readers focus on the concept being discussed.

Acknowledgements

First, I thank my parents for teaching me a number of valuable lessons of life including the value of hardwork and the value of good education (neither of which I have learnt yet!). I also thank my wife Madhvi for her patience, her encouragement, and also for having tolerated my long periods of silence and temper tantrums! Thanks (rather apologies) to my little daughters, Surabhi and Sakshi, who tolerated Papa's frequent refusals to take them on outings.

I thank Dr David Mulvaney and Dr Sekharjit Datta of the University of Loughborough for their valuable guidance, encouragement, and inspiration. My teachers always encouraged me to think big and to think independently. My sincerest gratitude to each one of them.

The editorial team of Oxford University Press deserves my heartfelt thanks for their guidance and for their timely reminders about the deadlines I would have definitely missed otherwise!

Feedback about the book is most welcome. Readers are requested and encouraged to send their feedback to the author's mail id sourav1903@yahoo.com.

Sourav Sahay

Preface to the Second Edition

The object-oriented programming system (OOPS) enables a programmer to model real-world objects. It allows the programmer to add characteristics like data security, data encapsulation, etc.

In the procedure-oriented programming system, procedures are dissociated from data and are not a part of it. Instead, they receive structure variables, or their addresses, and then work upon them. The code design is centered around procedures. While this may sound obvious, this programming pattern has its drawbacks, a major one being that the data is not secure. It can be manipulated by *any* procedure.

It is the lack of data security of the procedure-oriented programming system that led to OOPS, in which, with the help of a new programming construct and new keywords, associated functions of the data structure can be given exclusive rights to work upon its variables.

There is another characteristic of real-world objects—a guaranteed initialization of data. Programming languages that implement OOPS enable library programmers to incorporate this characteristic of real-world objects into structure variables. Library programmers can ensure a guaranteed initialization of data members of structure variables to the desired values. For this, application programmers do not need to write code explicitly.

OOPS further supports the following concepts:

- **Inheritance** This feature allows a class to inherit the data and function members of an existing class.
- **Data abstraction** Data abstraction is a virtue by which an object hides its internal operations from the rest of the program.
- **Modularity** This feature supports dividing a program into small segments and implement those segments using different functions.
- **Polymorphism** Through polymorphism, functions with different set of formal arguments can have the same name.

The first edition had covered the fundamentals of the object oriented programming system in depth. These explanations in the first edition hold true for any programming language that supports OOPS. This second edition enhances coverage, as listed below.

New to this Edition

- New chapter on data structures containing new and original algorithms, especially an elegant and simple recursive algorithm for inserting nodes into trees. The explanations are elaborate and full of diagrams.
- New sections on explicit constructors, command line arguments, and re-throwing exceptions.

- Expanded glossary.
- Accompanying CD contains all the program codes given in the text.

Key Features

- Simple and concise language eases the understanding of complex concepts that have made C++ powerful but enigmatic.
- Plenty of solved examples with complete program listings and test cases to reinforce learning.
- Review questions and program writing exercises at the end of each chapter to provide additional practice.
- Self-tests at the end of the book to prepare the students for examinations.

Organization of the Book

A brief knowledge of C language is a prerequisite for this book. The readers need to know how programs are written in C, data types, decision-making and looping constructs, operators, functions, header files, pointers, and structures.

Chapter 1 contains an explanation of the procedure-oriented programming system, the role played by structures in this system, its drawbacks and how these drawbacks led to the creation of OOPS. The meaning and method of modelling real-world objects by the object-oriented programming system have been clearly explained. The chapter includes a study of the non-object-oriented features of C++.

Chapter 2 is devoted to the study of objects and classes. It gives a thorough explanation of the class construct of C++. Superiority of the class construct of C++ over the structure construct of C language is explained. A description of the various types and features of member functions and member data is included. Other concepts included are namespaces, arrays of objects, arrays in objects, and nested classes.

Chapter 3 deals with dynamic memory management. It explains the use of the new and the delete operators. It also explains the method of specifying our own new handler for handling out-of-memory conditions.

Chapter 4 explains constructors and destructors. It discusses their importance, their features, and the method of defining them.

Chapter 5 is devoted to inheritance. Concepts like base class, derived class, base class pointer, and derived class pointer are covered. The protected keyword and the implications of deriving by different access specifiers are explained. This chapter describes various types of inheritance.

Chapter 6 gives a detailed explanation of one of the most striking features of C++—dynamic polymorphism. This chapter describes the virtual functions and how it enables C++ programmers to extend class libraries. The importance of pure virtual functions and clone functions is also explained.

Chapter 7 describes the standard C++ library for handling streams. It explains the two types of input and output—text mode and binary mode. Input and output from disk files are explained. The chapter also describes the use of error-handling routines of the standard C++ stream library and manipulators.

Chapter 8 is devoted to operator overloading, type conversion, new style casts, and RTTI. This chapter explains the various intricacies and the proper use of operator overloading. This chapter also explains how a C++ programmer can implement conventional style type

conversions. New style casts for implementing type conversions are explained next. This chapter ends with a treatment of run time type information (RTTI).

Chapter 9 explains and illustrates the most important data structures—linked lists and trees. It includes full-fledged programs that can be used to create various data structures.

Chapter 10 contains a detailed description of templates. The importance of function templates and class templates and their utilization in code reuse is explained. This chapter also provides an overview of the Standard Template Library (STL) of C++.

Chapter 11 explains the concept of exception handling. It begins with a section on conventional methods and their drawbacks. This is followed by an explanation of the try-catch-throw mechanism provided by C++ and its superiority over the conventional methods.

The appendices in the book include a case study, comparison of C++ with C, comparison of C++ with Java, an overview of object-oriented analysis and design, and self tests.

Acknowledgements

The blessings of my parents continue to give me the courage I need to overcome the obstacles that are associated with difficult ventures like writing books. Every achievement of my life, including this book, is because of the valuable education they gave me early in my life. Thanks to my wife Madhvi against whose wishes I decided to spend most of the weekends over the last 2 years on my laptop writing this edition. My daughters Surabhi and Sakshi continue to inspire and motivate me.

Thanks to Professor Shanmuka Swamy, Assistant Professor in the Sridevi Institute of Engineering and Technology, Tumkur, for pointing out a couple of printing mistakes in the first edition. These have been corrected.

The editorial staff members of the Oxford University Press deserve a special mention for its support and prompt responses.

Please continue to send your valuable feedback and questions to my e-mail id sourav1903@yahoo.com.

Sourav Sahay

Brief Contents

Preface to the Second Edition *iii*

Preface to the First Edition *vi*

Detailed Contents *xi*

1. Introduction to C++ 1
2. Classes and Objects 31
3. Dynamic Memory Management 78
4. Constructors and Destructors 92
5. Inheritance 117
6. Virtual Functions and Dynamic Polymorphism 153
7. Stream and File Handling 172
8. Operator Overloading, Type Conversion, New Style Casts, and RTTI 211
9. Data Structures 283
10. Templates 372
11. Exception Handling 393

Appendix A: Case Study—A Word Query System 417
Appendix B: Comparison of C++ with C 425
Appendix C: Comparison of C++ with Java 427
Appendix D: Object-Oriented Analysis and Design 437
Appendix E: Glossary 449
Appendix F: Self Tests 454

Bibliography *460*

Index *461*

Detailed Contents

Preface to the Second Edition *iii*

Preface to the First Edition *vi*

Brief Contents *ix*

1. Introduction to C++ **1**

1.1 A Review of Structures *1*
1.1.1 The Need for Structures *1*
1.1.2 Creating a New Data Type Using Structures *4*
1.1.3 Using Structures in Application Programs *5*
1.2 Procedure-Oriented Programming Systems *5*
1.3 Object-Oriented Programming Systems *7*
1.4 Comparison of C++ with C *8*
1.5 Console Input/Output in C++ *9*
1.5.1 Console Output *9*
1.5.2 Console Input *12*
1.6 Variables in C++ *13*
1.7 Reference Variables in C++ *14*
1.8 Function Prototyping *19*
1.9 Function Overloading *21*
1.10 Default Values for Formal Arguments of Functions *23*
1.11 Inline Functions *25*

2. Classes and Objects **31**

2.1 Introduction to Classes and Objects *31*
2.1.1 Private and Public Members *33*
2.1.2 Objects *36*
2.1.3 Scope Resolution Operator *37*
2.1.4 Creating Libraries Using the Scope Resolution Operator *38*
2.1.5 Using Classes in Application Programs *39*
2.1.6 `this` Pointer *40*
2.1.7 Data Abstraction *45*
2.1.8 Explicit Address Manipulation *47*
2.1.9 Arrow Operator *47*
2.1.10 Calling One Member Function from Another *48*

2.2 Member Functions and Member Data *49*
2.2.1 Overloaded Member Functions *49*
2.2.2 Default Values for Formal Arguments of Member Functions *51*
2.2.3 Inline Member Functions *52*
2.2.4 Constant Member Functions *52*
2.2.5 Mutable Data Members *54*
2.2.6 Friends *54*
2.2.7 Static Members *59*
2.3 Objects and Functions *65*
2.4 Objects and Arrays *66*
2.4.1 Arrays of Objects *67*
2.4.2 Arrays Inside Objects *67*
2.5 Namespaces *68*
2.6 Nested Inner Classes *71*

3. Dynamic Memory Management **78**
3.1 Introduction *78*
3.2 Dynamic Memory Allocation *79*
3.3 Dynamic Memory Deallocation *84*
3.4 `set_new_handler()` function *88*

4. Constructors and Destructors **92**
4.1 Constructors *92*
4.1.1 Zero-argument Constructor *94*
4.1.2 Parameterized Constructors *97*
4.1.3 Explicit Constructors *103*
4.1.4 Copy Constructor *105*
4.2 Destructors *109*
4.3 Philosophy of OOPS *112*

5. Inheritance **117**
5.1 Introduction *117*
5.1.1 Effects of Inheritance *118*
5.1.2 Benefits of Inheritance *120*
5.1.3 Inheritance in Actual Practice *120*
5.1.4 Base Class and Derived Class Objects *121*
5.1.5 Accessing Members of the Base Class in the Derived Class *121*
5.2 Base Class and Derived Class Pointers *122*
5.3 Function Overriding *127*
5.4 Base Class Initialization *129*
5.5 Protected Access Specifier *132*
5.6 Deriving by Different Access Specifiers *133*
5.6.1 Deriving by the Public Access Specifier *133*
5.6.2 Deriving by the Protected Access Specifier *135*
5.6.3 Deriving by the Private Access Specifier *136*
5.7 Different Kinds of Inheritance *139*
5.7.1 Multiple Inheritance *139*
5.7.2 Ambiguities in Multiple Inheritance *141*

5.7.3 Multi-level Inheritance *145*
5.7.4 Hierarchical Inheritance *147*
5.7.5 Hybrid Inheritance *148*
5.8 Order of Invocation of Constructors and Destructors *149*

6. Virtual Functions and Dynamic Polymorphism 153
6.1 Need for Virtual Functions *153*
6.2 Virtual Functions *156*
6.3 Mechanism of Virtual Functions *160*
6.4 Pure Virtual Functions *162*
6.5 Virtual Destructors and Virtual Constructors *167*
6.5.1 Virtual Destructors *167*
6.5.2 Virtual Constructors *168*

7. Stream and File Handling 172
7.1 Streams *172*
7.2 Class Hierarchy for Handling Streams *172*
7.3 Text and Binary Input/Output *174*
7.3.1 Data Storage in Memory *174*
7.3.2 Input/Output of Character Data *175*
7.3.3 Input/Output of Numeric Data *175*
7.3.4 Note on Opening Disk Files for I/O *176*
7.4 Text Versus Binary Files *176*
7.5 Text Output/Input *177*
7.5.1 Text Output *177*
7.5.2 Text Input *181*
7.6 Binary Output/Input 185
7.6.1 Binary Output—`write()` Function 185
7.6.2 Binary Input—`read()` Function 189
7.7 Opening and Closing Files 193
7.7.1 `open()` Function 193
7.7.2 `close()` Function 194
7.8 Files as Objects of the `fstream` Class 194
7.9 File Pointers 194
7.9.1 `seekp()` Function 195
7.9.2 `tellp()` Function 196
7.9.3 `seekg()` Function 196
7.9.4 `tellg()` Function 196
7.10 Random Access to Files *197*
7.11 Object Input/Output Through Member Functions *197*
7.12 Error Handling *199*
7.12.1 `eof()` Function *199*
7.12.2 `fail()` Function *199*
7.12.3 `bad()` Function *200*
7.12.4 `clear()` Function *200*
7.13 Manipulators *201*
7.13.1 Pre-defined Manipulators *201*

7.13.2 User-defined Manipulators *203*
7.14 Command Line Arguments *204*

8. Operator Overloading, Type Conversion, New Style Casts, and RTTI 211
8.1 Operator Overloading *211*
8.1.1 Overloading Operators—The Syntax *212*
8.1.2 Compiler Interpretation of Operator-Overloading Functions *214*
8.1.3 Overview of Overloading Unary and Binary Operators *216*
8.1.4 Operator Overloading *216*
8.1.5 Rules for Operator Overloading *219*
8.2 Overloading Various Operators *221*
8.2.1 Overloading Increment and Decrement Operators (Prefix and Postfix) *221*
8.2.2 Overloading Unary Minus and Unary Plus Operator *224*
8.2.3 Overloading Arithmetic Operators *225*
8.2.4 Overloading Relational Operators *230*
8.2.5 Overloading Assignment Operator *234*
8.2.6 Overloading Insertion and Extraction Operators *240*
8.2.7 Overloading `new` and `delete` Operators *244*
8.2.8 Overloading Subscript Operator *261*
8.2.9 Overloading `Pointer-to-member` (->) Operator (Smart Pointer) *265*
8.3 Type Conversion *267*
8.3.1 Basic Type to Class Type *267*
8.3.2 Class Type to Basic Type *268*
8.3.3 Class Type to Class Type *269*
8.4 New Style Casts and the `typeid` Operator *271*
8.4.1 `dynamic_cast` Operator *271*
8.4.2 `static_cast` Operator *275*
8.4.3 `reinterpret_cast` Operator *276*
8.4.4 `const_cast` Operator *276*
8.4.5 `typeid` Operator *277*

9. Data Structures 283
9.1 Introduction *283*
9.2 Linked Lists *284*
9.3 Stacks *336*
9.4 Queues *340*
9.5 Trees *343*
9.5.1 Binary Trees *344*
9.5.2 Binary Search Trees *347*

10. Templates 372
10.1 Introduction *372*
10.2 Function Templates *373*
10.3 Class Templates *378*
10.3.1 Nested Class Templates *382*
10.4 Standard Template Library *382*
10.4.1 `list` Class *383*

10.4.2 `vector` Class *386*
10.4.3 `pair` Class *387*
10.4.4 `map` Class *387*
10.4.5 `set` Class *389*
10.4.6 `multimap` Class *389*
10.4.7 `multiset` Class *390*

11. Exception Handling **393**
11.1 Introduction *393*
11.2 C-Style Handling of Error-generating Code *394*
11.2.1 Terminate the Program *394*
11.2.2 Check the Parameters before Function Call *395*
11.2.3 Return a Value Representing an Error *396*
11.3 C++-Style Solution—the try/throw/catch Construct *397*
11.3.1 It is Necessary to Catch Exceptions *400*
11.3.2 Unwinding of the Stack *401*
11.3.3 Need to Throw Class Objects *404*
11.3.4 Accessing the Thrown Object in the Catch Block *406*
11.3.5 Throwing Parameterized Objects of a Nested Exception Class *408*
11.3.6 Catching Uncaught Exceptions *409*
11.3.7 Re-throwing Exceptions *410*
11.4 Limitation of Exception Handling *414*

Appendix A: Case Study—A Word Query System **417**
Problem Statement *417*
A Sample Run *417*
The Source Code *418*
Explanation of the Code *420*

Appendix B: Comparison of C++ with C **425**
Non-object-oriented Features Provided in C++ that are Absent in C Language *425*
Object-oriented Features Provided in C++ to make it Comply with the Requirements of the Object-Oriented Programming System *426*

Appendix C: Comparison of C++ with Java **427**
C.1 Similarities between C++ and Java *427*
C.2 Differences between C++ and Java *428*

Appendix D: Object-Oriented Analysis and Design **437**
D.1 Introduction *437*
Why Build Models? *437*
Overview of OOAD *437*
D.2 Object-Oriented Model *438*
Object Model *438*
Dynamic Model *442*
Functional Model *444*
D.3 Analysis *446*

Overview of Analysis *446*
Object Modelling *446*
Dynamic Modelling *446*
Functional Modelling *447*
D.4 System Design *447*
Breaking the System into Sub-systems *447*
Layers *447*
Partitions *447*
D.5 Object Design *448*
Overview of Object Design *448*
D.6 Implementation *448*

Appendix E: Glossary **449**

Appendix F: Self Tests **454**
Test 1 *454*
Test 2 *456*
Test 3 *458*

Bibliography *460*

Index *461*

1 Introduction to C++

OVERVIEW

This chapter introduces the reader to the fundamentals of object-oriented programming systems (OOPS).

The chapter begins with an overview of structures, the reasons for their inclusion as a language construct in C language, and their role in procedure-oriented programming systems. Use of structures for creating new data types is described. Also, the drawbacks of structures and the development of OOPS are elucidated.

The middle section of the chapter explains OOPS, supplemented with suitable examples and analogies to help in understanding this tricky subject.

The concluding section of the chapter includes a study of a number of new features that are implemented by C++ compilers but do not fall under the category of object-oriented features. (Language constructs of C++ that implement object-oriented features are dealt with in the next chapter.)

1.1 A Review of Structures

In order to understand procedure-oriented programming systems, let us first recapitulate our understanding of structures in C. Let us review their necessity and use in creating new data types.

1.1.1 The Need for Structures

There are cases where the value of one variable depends upon that of another variable.

Take the example of date. A date can be programmatically represented in C by three different integer variables taken together. Say,

```
int d,m,y; //three integers for representing dates
```

Here 'd', 'm', and 'y' represent the day of the month, the month, and the year, respectively. Observe carefully. Although these three variables are not grouped together in the code, they actually belong to the same group. *The value of one variable may influence the value of the other two*. In order to understand this clearly, consider a function `next_day()` that accepts the addresses of the three integers that represent a date and changes their values to represent the next day. The prototype of this function will be

```
void next_day(int *,int *,int *);  //function to calculate
                                   //the next day
```

Suppose,

```
d=1;
m=1;
y=2002;                              //1st January, 2002
```

Now, if we write

```
next_day(&d,&m,&y);
```

'd' will become 2, 'm' will remain 1, and 'y' will remain 2002.

But if

```
d=28;
m=2;
y=1999; //28th February, 1999
```

and we call the function as

```
next_day(&d,&m,&y);
```

'd' will become 1, 'm' will become 3, and 'y' will remain 1999.
Again, if

```
d=31;
m=12;
y=1999;                              //31st December, 1999
```

and we call the function as

```
next_day(&d,&m,&y);
```

'd' will become 1, 'm' will become 1, and 'y' will become 2000.

As you can see, 'd', 'm', and 'y' actually belong to the same group. A change in the value of one may change the value of the other two. *But there is no language construct that actually places them in the same group*. Thus, members of the wrong group may be accidentally sent to the function (Listing 1.1)!

Listing 1.1 Problem in passing groups of programmatically independent but logically dependent variable

```
d1=28;  m1=2;   y1=1999;          //28th February, 1999
d2=19;  m2=3;   y2=1999;          //19th March, 1999
next_day(&d1,&m1,&y1);            //OK
next_day(&d1,&m2,&y2);            //What? Incorrect set passed!
```

As can be observed in Listing 1.1, there is nothing in the language itself that prevents the wrong set of variables from being sent to the function. Moreover, integer-type variables that are not meant to represent dates might also be sent to the function!

Let us try arrays to solve the problem. Suppose the `next_day()` function accepts an array as a parameter. Its prototype will be

```
void next_day(int *);
```

Let us declare date as an array of three integers.

```
int date[3];
date[0]=28;
date[1]=2;
date[2]=1999;          //28th February, 1999
```

Now, let us call the function as follows:

```
next_day(date);
```

The values of 'date[0]', 'date[1]', and 'date[2]' will be correctly set to 1, 3, and 1999, respectively. Although this method seems to work, it certainly appears unconvincing. After all *any* integer array can be passed to the function, even if it does not necessarily represent a date. There is no data type of date itself. Moreover, this solution of arrays will not work if the *variables are not of the same type*. The solution to this problem is to create a data type called date itself using structures

```
struct date                          //a structure to represent dates
{
   int d, m, y;
};
```

Now, the `next_day()` function will accept the address of a variable of the structure date as a parameter. Accordingly, its prototype will be as follows:

```
void next_day(struct date *);
```

Let us now call it as shown in Listing 1.2.

Listing 1.2 The need for structures

```
struct date d1;
d1.d=28;
d1.m=2;
d1.y=1999;
next_day(&d1);
```

'd1.d', 'd1.m', and 'd1.y' will be correctly set to 1, 3, and 1999, respectively. Since the function takes the address of an entire structure variable as a parameter at a time, there is no chance of variables of the different groups being sent to the function.

Structure is a programming construct in C that allows us to put together variables that should be together.

Library programmers use structures to create new data types. Application programs and other library programs use these new data types by declaring variables of this data type.

```
struct date d1;
```

They call the associated functions by passing these variables or their addresses to them.

```
d1.d=31;
d1.m=12;
d1.y=2003;
next_day(&d1);
```

Finally, they use the resultant value of the passed variable further as per requirements.

```
printf("The next day is: %d/%d/%d\n", d1.d, d1.m, d1.y);
```

Output

The next day is: 01/01/2004

1.1.2 Creating a New Data Type Using Structures

Creation of a new data type using structures is loosely a three-step process that is executed by the library programmer.

Step 1: Put the structure definition and the prototypes of the associated functions in a header file, as shown in Listing 1.3.

Listing 1.3 Header file containing definition of a structure variable and prototypes of its associated functions

```
/*Beginning of date.h*/
/*This file contains the structure definition and
prototypes of its associated functions*/

struct date
{
   int d,m,y;
};
void next_day(struct date *);      //get the next date
void get_sys_date(struct date *);  //get the current
                                       //system date
/*
   Prototypes of other useful and relevant functions to
   work upon variables of the date structure
*/
/*End of date.h*/
```

Step 2: As shown in Listing 1.4, put the definition of the associated functions in a source code and create a library.

Listing 1.4 Defining the associated functions of a structure

```
/*Beginning of date.c*/
/*This file contains the definitions of the associated
functions*/
#include "date.h"

void next_day(struct date * p)
{
//calculate the date that immediately follows the one
//represented by *p and set it to *p.
}
void get_sys_date(struct date * p)
{
//determine the current system date and set it to *p
}
/*
   Definitions of other useful and relevant functions to work upon variables
   of the date structure
*/
/*End of date.c*/
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.

Creation of a structure and creation of its associated functions are two separate steps that together constitute one complete process.

1.1.3 Using Structures in Application Programs

The steps to use this new data type are as follows:
Step 1: Include the header file provided by the library programmer in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main( )
{
   . . . .
   . . . .
}
/*End of dateUser.c*/
```

Step 2: Declare variables of the new data type in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main( )
{
   struct date d;
   . . . .
   . . . .
}
/*End of dateUser.c*/
```

Step 3: As shown in Listing 1.5, embed calls to the associated functions by passing these variables in the source code.

Listing 1.5 Using a structure in an application program

```
/*Beginning of dateUser.c*/
#include"date.h"
void main()
{
   struct date d;
   d.d=28;
   d.m=2;
   d.y=1999;
   next_day(&d);
   . . . .
   . . . .
}
/*End of dateUser.c*/
```

Step 4: Compile the source code to get the object file.
Step 5: Link the object file with the library provided by the library programmer to get the executable or another library.

1.2 Procedure-Oriented Programming Systems

In light of the previous discussion, let us understand the procedure-oriented programming system. The foregoing pattern of programming divides the code into functions. Data (contained in structure variables) is passed from one function to another to be read from or written into. The focus is on procedures. This programming pattern is, therefore, a feature of the procedure-oriented programming system.

In the procedure-oriented programming system, procedures are dissociated from data and are not a part of it. Instead, they receive structure variables or their addresses and work upon them. The code design is centered around procedures. While this may sound obvious, this programming pattern has its drawbacks.

The drawback with this programming pattern is that the data is not secure. It can be manipulated by *any* procedure. Associated functions that were designed by the library programmer do not have the exclusive rights to work upon the data. They are not a part of the structure definition itself. Let us see why this is a problem.

Suppose the library programmer has defined a structure and its associated functions as described above. Further, in order to perfect his/her creation, he/she has rigorously tested the associated functions by calling them from small test applications. Despite his/her best efforts, he/she cannot be sure that an application that uses the structure will be bug free. The application program might modify the structure variables, not by the associated function he/she has created, but by some code inadvertently written in the application program itself. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.

Now, let us look at the situation from the application programmer's point of view. Consider an application of around 25,000 lines (quite common in the real programming world), in which variables of this structure have been used quite extensively. During testing, it is found that the date being represented by one of these variables has become 29th February 1999! The faulty piece of code that is causing this bug can be anywhere in the program. Therefore, debugging will involve a visual inspection of the entire code (of 25000 lines!) and will not be limited to the associated functions only.

The situation becomes especially grave if the execution of the code that is likely to corrupt the data is conditional. For example,

```
if(<some condition>)
   d.m++;     //d is a variable of date structure… d.m may
              //become 13!
```

The condition under which the bug-infested code executes may not arise during testing. While distributing his/her application, the application programmer cannot be sure that it would run successfully. Moreover, every new piece of code that accesses structure variables will have to be visually inspected and tested again to ensure that it does not corrupt the members of the structure. After all, compilers that implement procedure-oriented programming systems do not prevent unauthorized functions from accessing/manipulating structure variables.

Let us think of a compiler that enables the library programmer to assign exclusive rights to the associated functions for accessing the data members of the corresponding structure. If this happens, then our problem is solved. If a function which is not one of the intended associated functions accesses the data members of a structure variable, a compile-time error will result. To ensure a successful compile of his/her application code, the application programmer will be forced to remove those statements that access data members of structure variables. Thus, the application that arises out of a successful compile will be the outcome of a piece of code that is free of any unauthorized access to the data members of the structure variables used therein. Consequently, if a run-time error arises, attention can be focused on the associated library functions.

It is the lack of data security of procedure-oriented programming systems that led to object-oriented programming systems (OOPS). This new system of programming is the subject of our next discussion.