

Graham Seed

# AN INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN C++

With applications in  
Computer Graphics



Book includes all program  
files available via the  
Springer ftp site



Springer

# An Introduction to Object-Oriented Programming in C++

## With Applications in Computer Graphics

---

**Springer**

*London*

*Berlin*

*Heidelberg*

*New York*

*Barcelona*

*Budapest*

*Hong Kong*

*Milan*

*Paris*

*Santa Clara*

*Singapore*

*Tokyo*

Graham M. Seed

---

# **An Introduction to Object-Oriented Programming in C++**

## **With Applications in Computer Graphics**

Springer

Graham M. Seed, BEng, MPhil, PhD  
Department of Mechanical & Chemical Engineering  
Heriot Watt University, Edinburgh EH14 4AS, UK

ISBN-13: 978-3-540-76042-9  
DOI: 10.1007/978-1-4471-3378-0

e-ISBN-13: 978-1-4471-3378-0

British Library Cataloguing in Publication Data  
Seed, Graham Mark

An introduction to object-oriented programming in C++ : with applications in computer graphics  
1.C++ (Computer program language) 2.Object-oriented programming (Computer Science)  
I.Title  
005.1'33

Library of Congress Cataloging-in-Publication Data  
Seed, Graham M., 1965-

An introduction to object-oriented programming in C++ : with applications in computer graphics /  
Graham M. Seed.

p. cm.

Includes bibliographical references and index.

1. Object-oriented programming (Computer science) 2. C++ (Computer program language) 3.  
Computer graphics. I. Title.

QA76.64.S375 1996

006.6'6--dc20

96-21786

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

© Springer-Verlag London Limited 1996  
Softcover reprint of the hardcover 1st edition 1996

The use of registered names, trademarks etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Typesetting: Ian Kingston Editorial Services, Nottingham  
Printed and bound by Bell & Bain Ltd, Glasgow, UK  
34/3830-543210 Printed on acid-free paper

*To the memory of  
Tom Bannister (1909-1992)  
what a guy!*

## Why Another Book on C++ and why Programming and Graphics?

Anyone who has browsed through the ‘Computing’ section of a bookshop (assuming it has one) will not need much convincing that there are a lot of C++ books out there. So why add yet another to the shelf?

This book attempts to introduce you to the C++ language via computer graphics because the object-oriented programming features of C++ naturally lend themselves to graphics. Thus, this book is based around a central theme: computer graphics and the development of ‘real’ object-oriented tools for graphical modelling. This approach is adopted (as opposed to learning by small, unrelated, often hypothetical, examples) because I didn’t want to introduce C++ as a collection of language features. While introducing the syntax and features of C++, it is just as important to demonstrate simultaneously the reason for such features and when to apply them – in other words, language and design are given equal priority. Also, a key objective in writing this book is to present you with a comprehensive introductory text on programming in the C++ language.

The first part of the book introduces the fundamentals of C++ programming. After the foundations have been laid, the object-oriented programming features of C++ are introduced via the development of classes such as Point and Line. These *primitives* are then put to good use in developing powerful two- and three-dimensional graphics classes such as Circle, Sphere, Triangle, Quadrilateral, Polygon and Tetrahedra. Along the way, *tool classes* are also developed, such as Vector, Vector3D, Matrix, Complex, String, LinkedList, ReadFile and WriteFile, World, DIBitmap and many more.

## What you Need to Know

No previous programming experience in C or C++ is required for this book. If you are considering learning C before C++, then my advice is: don’t. When I first started to learn C++ I asked myself the exact same question and unfortunately made the wrong decision and developed a good understanding of C before attempting to learn C++. I wouldn’t say that this proved to be a complete waste of time, but it is certainly not necessary to learn C in order to learn C++.

As a C++ programmer you will develop programs and think about program design in a completely different manner than the traditional procedural techniques in C. C++ offers a programmer object-oriented features such as encapsulation, abstraction, inheritance and multiple inheritance, and polymorphism through the use of classes and objects, member functions, operator overloading, virtual functions, templates, exception handling, run-time type information and so on. In addition, C++ offers a lot more. C++ provides many new features (for instance **const**, **new** and **delete**), stronger type checking and minor, but nevertheless important, modifications to C.

If you do know another procedural language, such as Fortran, Basic, Pascal or C, then great. Some skills that you have learned with such languages will undoubtedly prove useful (such as the meaning of variables, conditions and loops), whereas other skills will appear to conflict with the C++ way of programming.

This book does assume that you have had some contact with computers before and are familiar with Microsoft's DOS and Windows operating systems.

## **Hardware and Software Requirements**

It is assumed that readers of this book have access to a computer and a compiler, such as Borland C++, Microsoft Visual C++ or Symantec C++ for a Windows environment. It is difficult to specify general minimum hardware and software requirements that cover all compilers, but as an example, Borland C++ version 5.0 for Windows requires Windows 95 or Windows NT, a hard disk with 175 Mbyte free disk space for a full installation, at least 16 Mbyte RAM, a 3.5"/1.44 Mbyte floppy drive or CD-ROM and a Windows-compatible mouse. An Intel 486 or higher processor is recommended.

## **Obtaining the Program Files**

All of the source code, including solutions to chapter exercises, presented throughout the book is available by an Internet ftp connection following the procedure:

ftp host name:	ftp.springer.co.uk
login name:	ftp
password:	<your e-mail address>
ftp directory:	/pub/springer/Books/3540760423
get:	cpp_prog.zip

The program files have been compressed in to a single PKZIP file CPP\_PROG.ZIP. To preserve the chapter directory structure of the files use the -d option when unzipping the files. For instance:

```
c:\cpp_prog>pkunzip cpp_prog.zip -d
```

Alternatively, the program files can be obtained via the following URL:

```
ftp://ftp.springer.co.uk/pub/springer/Books/3540760423
```

All of the programs have been tested with the Borland C++ and Microsoft Visual C++ compilers.

## **Preparation of the Book**

The text of this book was written with the aid of Microsoft's Word for Windows version 6.0. The C++ program code was written with the Borland Integrated Development Environment (IDE) editor. The program code was transferred from IDE to

---

Word using *Copy* and *Paste*. Tables and figures were completed using Word and CorelDRAW version 5.0.

All program code developed for the book was developed exclusively on a desktop PC consisting of an Intel Pentium 90 MHz CPU, 16 Mbyte RAM, 1 Mbyte video memory and 1 Gbyte hard disk space.

## How this Book is Organised

It would be great to dive straight into the object-oriented programming (OOP) features of C++. Unfortunately, this is simply not possible for an introductory text. Therefore, the initial chapters lay the foundations of the C++ language by examining the ‘traditional’ procedural C features of C++.

If you already have experience of another programming language, particularly C, it may be tempting to skip the first few chapters and jump straight into the object-oriented features of C++. If you choose to do this then be careful. C++ has many differences from and additions to C, some of which you may miss if you jump for OOP. The object-oriented features *really* begin in Chapters 8 and 9, when structures and classes are introduced.

The book begins by presenting a general overview of the C++ programming language and object-oriented programming and what is meant by *an object*. We take a look at the history of C++, its relationship to C and what makes C++ an object-oriented programming language. Chapter 2 briefly discusses your programming environment and what support to expect from a C++ compiler.

Chapter 3 introduces several key elements of program construction in C++ via the simplest of programs. Each line of a simple C++ program is introduced in detail to help the reader who is learning C++ for the first time feel comfortable with the basics of program construction. The first elements of a C++ program discussed are user comments, preprocessor directives, the use of header files, a function definition, program statements, strings, the `cout` standard output stream object, the `endl` manipulator and the `<<` insertion operator. The importance of developing a consistent programming style is stressed and several different popular styles are presented.

The fundamental data types of the C++ language, and their corresponding data type modifiers, are introduced in Chapter 4. The notion of a constant or variable identifier is demonstrated with the aid of numerous examples for each fundamental data type, and the distinction between an identifier declaration and definition is noted. Once data types have been introduced, statements, operators, expressions, type conversions and casting are covered. Chapter 4 concludes by discussing storage class specifiers and declaring assembly language program code directly within a C++ program.

Chapter 5 examines the process of making decisions and performing repetition in C++. Decisions are implemented using either `if-else` or `switch` statements, whereas repetition is performed using a `for`, `while` or `do-while`-loop. In addition, C++ supports keywords such as `break` and `default` for `if-else` and `switch` statements and the keywords `break` and `continue` for loops which allow a programmer to interrupt program flow. Both decision-making and repetition require the use of the relational and logical operators, and Chapter 5 examines these operators and the bitwise and shift operators in detail.

The function is the basic tool of the majority of programming languages for modularising a large complex program. Chapter 6 discusses the C++ function: its

construction, definition, scope and application. Chapter 6 also discusses the powerful technique of function overloading, which C++ supports, and **inline** functions. The chapter concludes with an application of functions to the popular geometric problem of the intersection of two straight lines. Arrays are covered in Chapter 7. The chapter discusses what an array is, how the elements of an array are indexed, the dimension of an array and typical applications of arrays.

Chapter 8 presents structures, unions and enumerations for the creation of user-defined data types and **typedefs** for user-defined synonyms of data types. The **struct** statement allows a programmer to create a new data type which is a collection of similar or different data types. A **union** data type is similar to **struct** except that a **union** can hold only one of its data members at any given time. The user-defined **enum** data type is a collection of named data members which have equivalent integer values. A **typedef** is a synonym for an existing data type.

The **class** is the key object-oriented programming feature of the C++ language, and Chapter 9 covers the **class** in detail via a **class** called **Point**. Similar to **struct**, the **class** allows a programmer to create new data types by encapsulating data members, but the **class** also allows a programmer to encapsulate member functions which operate on a **class**'s data members. Operators and overloading operators for user-defined data types are examined in Chapter 10. Overloading operators for a user-defined **class** can significantly enhance the level of abstraction of the **class**.

Friend functions and classes are introduced in Chapter 11. Basically, a **friend** function or **class** has access to the **private** data members of an object of a given **class**, although the **friend** is not a member of that **class**.

Chapter 12 comprehensively examines the tricky subject of pointers. The dynamic memory allocation and deallocation operators **new** and **delete** and the **this** pointer are examined. Useful **Vector** and **Matrix** classes are developed which make good use of pointer manipulation and the **new** and **delete** operators, and help apply many of the topics covered in previous chapters.

Generic functions and parametrised data types are introduced in Chapter 13 through the **template** language-based feature. The exact type of object(s) that a function or **class** operates on is left unspecified in the function definition or **class** declaration until a function is called or an object is defined. The C++ **try-catch-throw** termination mechanism for handling runtime errors is presented in Chapter 14 and demonstrated for various classes.

The topic of **class** inheritance is covered in Chapter 15, which allows a derived **class** to inherit the features of one or more other base classes. Several properties of inheritance are discussed with reference to a **Shape** **class** hierarchy, including polymorphic classes which contain one or more **virtual** functions. Chapter 16 examines the run-time type information mechanism, which allows a programmer to obtain the type of an object at run-time, and the related topic of casting.

Both C and C++ streams are discussed in Chapter 17. Standard input and output and file streams are covered and an overview is given of the C function-based and C++ **class**-based stream libraries. Chapter 18 discusses the preprocessor, although it is not strictly part of the C++ language.

Chapter 19 discusses *namespaces*, one of the newest major features to be added to the C++ language. Namespaces are a language-based solution to the problem of name clashes resulting from C++'s single name space for a translational unit.

The final chapter concludes the book with an application of the C++ programming language to the development of a simple object-oriented ray-tracing program. The

ray-tracing application pulls together many of the C++ features presented, as well as the classes developed in previous chapters.

Each chapter concludes with an *Exercises* section with seven exercises which are chosen to reflect the key topics covered in the associated chapter. The solutions to all chapter exercises can be found with the accompanying chapter program files in the subdirectory \XERCISES on the ftp site.

*Graham M. Seed*

## Acknowledgements

Thanks to everyone who was involved in the development and publication of this book. A massive thanks to the book's editor, Beverley Ford, and the Springer production team, without whom this book would not have been possible. Thanks to Jon Ritchie for setting up and maintaining the ftp site. Heriot-Watt University has generously provided both time and resources for the completion of the book. I am indebted to Ian Chivers for performing an excellent review of a draft version of the book. Also, to the numerous anonymous reviewers – thanks!

I would especially like to acknowledge the continuing love and support of my parents, Jean and Terry, and the friendship and support of my Sheffield friends George E. Cardew, David H. Willis and, in particular, Catherine Harnick who said *go for it*. Thanks to all my friends and colleagues at work for their support and to the students that I have taught – I've learned more from them than they've learned from me! To Iain Grieve, the ex-office-neighbour: 'are you f'coffee?'. Finally, a thanks to Mark Radcliffe and Boy Lard ('Get to bed!') for music, poetry, philosophy and a good laugh through the late hours.

## About the Author

The author completed a B.Eng. degree at Leeds University, followed by an M.Phil. and a Ph.D. at Sheffield University in Mechanical Engineering. He has been programming in C and C++ for several years and lectures in programming techniques and applications in engineering to undergraduate and postgraduate students. His primary research interests involve computational mechanics and geometry and object-oriented methods. He is presently a lecturer at the Department of Mechanical and Chemical Engineering at Heriot-Watt University, Edinburgh.

## Correspondence

The author welcomes communication, comments and constructive criticism<sup>1</sup> on any part of the book. For correspondence contact the following virtual addresses:

Internet: g.m.seed@hw.ac.uk  
World-Wide Web: <http://www.hw.ac.uk/mecWWW/staff/gms.htm>

---

<sup>1</sup> 'chameleon, comedian, corinthian and caricature' ('The Bewlay Brothers', *Hunky Dory*, David Bowie)

# **Contents**

---

List of Programs . . . . .	xxvii
<b>1 Overview . . . . .</b>	<b>1</b>
1.1 Why C++? . . . . .	1
1.2 C++'s History . . . . .	3
1.3 C++'s Future . . . . .	4
1.4 There are Other OOP Languages . . . . .	5
1.5 Programming Paradigms . . . . .	6
1.6 Procedural and Modular Programming . . . . .	6
1.7 Data Abstraction . . . . .	8
1.8 Object-Oriented Programming . . . . .	9
1.8.1 Abstraction . . . . .	14
1.8.2 Encapsulation . . . . .	15
1.8.3 Modularity . . . . .	15
1.8.4 Inheritance . . . . .	15
1.8.5 Polymorphism . . . . .	16
1.8.6 Typing . . . . .	16
1.8.7 Automatic Memory Management . . . . .	18
1.8.8 Concurrency . . . . .	18
1.8.9 Persistence . . . . .	18
1.8.10 Operator Overloading . . . . .	18
1.9 Objects . . . . .	19
1.10 Syntax, Semantics and Pragmatics . . . . .	19
1.10.1 Syntax . . . . .	19
1.10.2 Semantics . . . . .	19
1.10.3 Pragmatics . . . . .	20
1.11 Applications of C++ . . . . .	20
1.12 References for C++ . . . . .	20
1.13 References for Graphics . . . . .	22
1.14 Notation . . . . .	22
1.15 Summary . . . . .	22
Exercises . . . . .	22
<b>2 The Development Environment . . . . .</b>	<b>25</b>
2.1 Hardware, Software and Setup Used . . . . .	25
2.1.1 Hardware . . . . .	25
2.1.2 Software . . . . .	25
2.2 The Borland Integrated Development Environment . . . . .	26
2.2.1 Editor . . . . .	26
2.2.2 Text Highlighting . . . . .	27
2.2.3 Project Manager . . . . .	27
2.2.4 Messages . . . . .	27
2.2.5 Browser . . . . .	27
2.2.6 AppExpert . . . . .	27

2.2.7	ClassExpert . . . . .	28
2.2.8	Debugger . . . . .	28
2.2.9	Resource Workshop . . . . .	28
2.2.10	Help . . . . .	28
2.2.11	Java Development Tools . . . . .	28
2.3	Container Class and ObjectWindows Libraries . . . . .	29
2.4	Summary . . . . .	30
	Exercises . . . . .	30
<b>3</b>	<b>Getting Started . . . . .</b>	<b>31</b>
3.1	The Simplest of Programs . . . . .	31
3.2	The Comment Line . . . . .	32
3.3	Preprocessor Directives . . . . .	34
3.4	What is IOSTREAM.H? . . . . .	35
3.5	" " or <> . . . . .	35
3.5.1	Other Header Files . . . . .	35
3.5.2	.H or .HPP . . . . .	36
3.6	Keywords . . . . .	36
3.7	The <i>main()</i> Function . . . . .	36
3.8	Program Statements . . . . .	38
3.8.1	The cout Identifier/Object . . . . .	39
3.8.2	The << Insertion Operator . . . . .	39
3.8.3	String Constants . . . . .	39
3.8.4	The endl Manipulator . . . . .	39
3.9	Compiling and Linking . . . . .	40
3.9.1	Compiling . . . . .	40
3.9.2	Linking . . . . .	40
3.10	A Note on Programming Style and Notation . . . . .	40
3.10.1	Be Consistent! . . . . .	43
3.11	Summary . . . . .	44
	Exercises . . . . .	44
<b>4</b>	<b>Fundamental Data Types, Declarations, Definitions and Expressions . . . . .</b>	<b>45</b>
4.1	Fundamental Data Types . . . . .	45
4.1.1	The <b>char</b> Data Type . . . . .	46
4.1.2	The <b>int</b> Data Type . . . . .	50
4.1.3	The <b>float</b> Data Type . . . . .	52
4.1.4	The <b>double</b> Data Type . . . . .	53
4.1.5	The <b>wchar_t</b> Type . . . . .	54
4.1.6	The <b>enum</b> Type . . . . .	55
4.1.7	The <b>bool</b> Type . . . . .	56
4.2	The <b>short</b> , <b>long</b> , <b>unsigned</b> , <b>signed</b> , <b>const</b> and <b>volatile</b> Data Type Modifiers . . . . .	58
4.2.1	The <b>short</b> Modifier . . . . .	59
4.2.2	The <b>long</b> Modifier . . . . .	59
4.2.3	The <b>unsigned</b> Modifier . . . . .	59
4.2.4	The <b>signed</b> Modifier . . . . .	59
4.2.5	The <b>const</b> Modifier . . . . .	59
4.2.6	The <b>volatile</b> Modifier . . . . .	61
4.3	Constants . . . . .	62
4.3.1	Integer Constants . . . . .	62
4.3.2	Floating-Point Constants . . . . .	62

---

4.3.3	Character and String Constants . . . . .	63
4.3.4	Enumeration Constants . . . . .	63
4.4	<b>const</b> or <b>#define</b> ? . . . . .	63
4.5	The <b>sizeof</b> Operator . . . . .	64
4.6	Keywords . . . . .	64
4.7	Declaration or Definition? . . . . .	64
4.8	Assignment . . . . .	66
4.9	Initialisation . . . . .	66
4.10	Expressions . . . . .	66
4.11	Operators . . . . .	67
4.11.1	Arithmetic Operators . . . . .	67
4.11.2	Chaining Operators . . . . .	67
4.11.3	Operator Shortcuts . . . . .	68
4.11.4	The Increment (++) and Decrement (--) Operators . . . . .	68
4.11.5	Operator Precedence, Associativity and Arity . . . . .	69
4.12	Manipulators . . . . .	70
4.12.1	The <code>endl</code> Manipulator . . . . .	72
4.12.2	The <code>flush</code> Manipulator . . . . .	72
4.12.3	The <code>setw()</code> Manipulator . . . . .	72
4.12.4	The <code>setprecision()</code> Manipulator . . . . .	73
4.12.5	The <code>setiosflags()</code> Manipulator . . . . .	73
4.12.6	The <code>resetiosflags()</code> Manipulator . . . . .	73
4.12.7	The <code>dec</code> , <code>hex</code> and <code>oct</code> Manipulators . . . . .	73
4.13	Basic File Input and Output . . . . .	74
4.14	Type Conversion and Casting . . . . .	76
4.15	The Storage Class Specifiers <b>auto</b> , <b>extern</b> , <b>register</b> and <b>static</b> . . . . .	77
4.15.1	The <b>auto</b> Specifier . . . . .	77
4.15.2	The <b>extern</b> Specifier . . . . .	78
4.15.3	The <b>register</b> Specifier . . . . .	79
4.15.4	The <b>static</b> Specifier . . . . .	80
4.16	The <b>asm</b> Declaration . . . . .	81
4.17	Summary . . . . .	82
	Exercises . . . . .	82
 5	 Making Decisions and Repetition . . . . .	85
5.1	Decisions . . . . .	85
5.1.1	The <b>if</b> Statement . . . . .	86
5.1.2	The <b>if-else</b> Statement . . . . .	87
5.1.3	Nested <b>ifs</b> . . . . .	92
5.1.4	Relational Operators . . . . .	94
5.1.5	Logical Operators . . . . .	95
5.1.6	The <b>switch</b> Statement . . . . .	99
5.1.7	The Conditional Operator (? :) . . . . .	102
5.2	Repetition . . . . .	103
5.2.1	The <b>for</b> -Loop . . . . .	103
5.2.2	The <b>while</b> -Loop . . . . .	107
5.2.3	The <b>do-while</b> Loop . . . . .	109
5.3	Forever Loops . . . . .	110
5.4	Nested Loops . . . . .	110
5.5	The <b>break</b> Statement . . . . .	111
5.6	The <b>continue</b> Statement . . . . .	113
5.7	The <b>goto</b> Statement . . . . .	114

5.8	Loop Body Visibility . . . . .	115
5.9	The Bitwise and Shift Operators . . . . .	115
5.9.1	Bitwise AND Operator (&) . . . . .	116
5.9.2	Bitwise OR Operator ( ) . . . . .	118
5.9.3	Bitwise Exclusive OR Operator (^) . . . . .	119
5.9.4	Bitwise One's Complement Operator (~) . . . . .	120
5.9.5	The Shift Operators . . . . .	120
5.9.6	The Bitwise and Shift Assignment Operators . . . . .	121
5.10	Summary . . . . .	122
	Exercises . . . . .	123
<b>6</b>	<b>Functions . . . . .</b>	<b>125</b>
6.1	A Look at Functions . . . . .	126
6.2	The Body of a Function . . . . .	126
6.3	Function Name . . . . .	127
6.4	Function Body . . . . .	127
6.5	Function Return Type . . . . .	128
6.5.1	<b>void</b> Return Type . . . . .	128
6.5.2	Default Return Type . . . . .	129
6.6	The <b>return</b> Statement . . . . .	129
6.7	Function Calling . . . . .	130
6.8	Function Definition . . . . .	131
6.9	Function Declaration . . . . .	132
6.10	Function Scope . . . . .	133
6.10.1	Local Scope . . . . .	133
6.10.2	Global Scope . . . . .	134
6.10.3	Precedence of Scope . . . . .	135
6.10.4	Pros and Cons of Local and Global Scope . . . . .	135
6.10.5	Scope Resolution Operator . . . . .	136
6.11	Function Arguments . . . . .	136
6.11.1	No Argument List . . . . .	137
6.11.2	By Value or by Reference? . . . . .	137
6.11.3	<b>const</b> Reference Arguments . . . . .	140
6.11.4	<b>const</b> Return Value . . . . .	140
6.12	Returning More than One Value . . . . .	141
6.13	Return by Reference . . . . .	142
6.14	Why Use Functions? . . . . .	143
6.15	Overloaded Functions . . . . .	150
6.16	Default Arguments . . . . .	155
6.17	Local <b>static</b> Variables . . . . .	156
6.18	<b>inline</b> Functions . . . . .	158
6.19	The <i>main()</i> Function . . . . .	159
6.19.1	Command Line Arguments . . . . .	160
6.19.2	<i>WinMain()</i> . . . . .	160
6.20	External Linkage . . . . .	161
6.21	Library Functions . . . . .	162
6.21.1	The ANSI C Standard Library . . . . .	163
6.22	Header Files . . . . .	166
6.22.1	ANSI C Header Files . . . . .	166
6.22.2	C++ Header Files . . . . .	167
6.22.3	Additional Header Files . . . . .	167
6.23	Not in the Library . . . . .	168
6.24	Intersection of Two Two-Dimensional Line Segments . . . . .	168

---

6.24.1	Parametric Lines . . . . .	168
6.24.2	Bounding Boxes and Parametric Intersection . . . . .	172
6.24.3	Clockwise or Anticlockwise? . . . . .	173
6.25	Summary . . . . .	179
	Exercises . . . . .	181
<b>7</b>	<b>Arrays . . . . .</b>	<b>183</b>
7.1	An Array . . . . .	183
7.2	The Size of an Array . . . . .	185
7.3	The <b>sizeof</b> Operator . . . . .	186
7.4	Array Indexing . . . . .	187
7.4.1	First Element . . . . .	187
7.4.2	Setting and Getting an Element . . . . .	187
7.4.3	Last Element . . . . .	188
7.5	Initialisation of One-Dimensional Arrays . . . . .	189
7.6	Two-Dimensional Arrays . . . . .	190
7.7	Initialisation of Two-Dimensional Arrays . . . . .	192
7.8	How are Two-Dimensional Arrays Used? . . . . .	193
7.9	Three- and Higher-Dimensional Arrays . . . . .	194
7.10	Strings . . . . .	195
7.10.1	Variable Strings . . . . .	195
7.10.2	Variable Strings with Embedded Spaces . . . . .	196
7.10.3	Constant Strings . . . . .	198
7.11	Wasted Space . . . . .	199
7.12	Arrays as Function Arguments . . . . .	200
7.12.1	One-Dimensional Arrays . . . . .	200
7.12.2	Two-Dimensional Arrays . . . . .	202
7.13	Calling Functions with Arrays as Arguments . . . . .	204
7.14	Strings as Function Arguments . . . . .	205
7.15	Passing the Size of an Array as an Argument . . . . .	206
7.16	Small, Medium, Compact, Large and Huge Memory Models . . . . .	209
7.16.1	Small and Medium Memory Models . . . . .	210
7.16.2	Compact, Large and Huge Memory Models . . . . .	210
7.16.3	Libraries . . . . .	210
7.16.4	Windows Programming . . . . .	210
7.17	Summary . . . . .	211
	Exercises . . . . .	211
<b>8</b>	<b>Structures, Unions, Enumerations and Typedefs . . . . .</b>	<b>215</b>
8.1	Structures . . . . .	215
8.1.1	Declaring a Structure . . . . .	217
8.1.2	Structure Tag or Name . . . . .	218
8.1.3	Defining a Structure Variable or Object . . . . .	218
8.1.4	Accessing a Structure's Data Members . . . . .	218
8.1.5	The Size of a Structure . . . . .	219
8.1.6	Declaring and Defining a Structure in a Single Statement . . . . .	220
8.1.7	Defining a Structure Variable or Object Without a Structure Name . . . . .	220
8.1.8	Output of a Structure's Data Members . . . . .	221
8.1.9	Assignment . . . . .	221
8.1.10	Arrays of Structures . . . . .	223
8.1.11	Structures as Function Arguments . . . . .	224

8.1.12	Nested Structures . . . . .	225
8.1.13	A Point Structure . . . . .	227
8.1.14	Data and Functions . . . . .	230
8.1.15	Private or Public? . . . . .	231
8.1.16	Point, Line and Triangle Structures . . . . .	232
8.1.17	Bit Fields . . . . .	235
8.2	Unions . . . . .	236
8.2.1	<b>sizeof</b> a union . . . . .	238
8.2.2	Application of a union . . . . .	239
8.3	Enumerations . . . . .	239
8.4	<b>typedefs</b> . . . . .	242
8.4.1	<b>typedefs</b> for Windows . . . . .	244
8.4.2	<b>typedefs</b> Offer Little Type Safety . . . . .	244
8.4.3	<b>typedef</b> Style . . . . .	245
8.5	Summary . . . . .	245
	Exercises . . . . .	246
9	The C++ Class . . . . .	249
9.1	A Point <b>class</b> . . . . .	249
9.2	<b>class</b> Declaration Syntax . . . . .	253
9.3	Objects and Instances . . . . .	254
9.4	A <b>public</b> Point <b>class</b> . . . . .	254
9.5	Member Functions . . . . .	256
9.6	Naming of Member Functions . . . . .	260
9.7	Calling Member Functions . . . . .	260
9.8	Defining Member Functions . . . . .	261
9.9	<b>static</b> Data Members and Member Functions . . . . .	265
9.10	<b>const</b> and <b>mutable</b> Data Members . . . . .	267
9.11	<b>volatile</b> Data Members . . . . .	269
9.12	Bit Fields . . . . .	269
9.13	Constructors . . . . .	270
9.13.1	Constructors are Called Automatically . . . . .	272
9.13.2	Default Initialisation . . . . .	273
9.13.3	Overloaded Constructors . . . . .	274
9.13.4	Default Constructors . . . . .	275
9.13.5	Data Member Initialisation . . . . .	276
9.13.6	Copy Constructor . . . . .	278
9.13.7	Constructors, <b>const</b> and <b>volatile</b> . . . . .	284
9.14	Destructors . . . . .	285
9.14.1	Destructors, <b>const</b> and <b>volatile</b> . . . . .	287
9.15	<b>inline</b> Member Functions . . . . .	287
9.15.1	Automatic <b>inline</b> Member Functions . . . . .	288
9.15.2	<b>inline</b> Constructors . . . . .	289
9.16	<b>const</b> Member Functions . . . . .	290
9.17	<b>volatile</b> Member Functions . . . . .	293
9.18	Default Member Function Arguments . . . . .	293
9.19	Functions with Object Arguments and Functions that Return Objects . . . . .	295
9.19.1	Returning by Reference . . . . .	299
9.20	Combining Constructor and Member Function Calls . . . . .	301
9.21	Arrays of Objects . . . . .	303
9.22	Local Classes . . . . .	304
9.23	Nested Classes . . . . .	305

---

9.24	Another <b>class</b> Declaration . . . . .	308
9.25	<b>struct</b> and <b>class</b> . . . . .	308
9.26	Intersection of Two Two-Dimensional Line Segments . . . . .	309
9.27	Summary . . . . .	313
	Exercises . . . . .	315
<b>10</b>	<b>Operators and Overloading</b> . . . . .	319
10.1	Overloading the Arithmetic Binary Operators . . . . .	319
10.1.1	The <b>operator</b> Keyword . . . . .	322
10.2	The Unary Increment (++) and Decrement (--) Operators . . . . .	325
10.3	Physical Meaning of Overloaded Point Operators . . . . .	328
10.4	The Relational Operators . . . . .	329
10.5	The Assignment Operator . . . . .	329
10.5.1	Default Assignment . . . . .	329
10.5.2	Overloaded Assignment Operator . . . . .	330
10.6	Non-Member Overloaded Operator Functions . . . . .	331
10.7	The Array Subscript Operator . . . . .	332
10.8	Composite Operators . . . . .	334
10.9	Conversions . . . . .	337
10.10	A String <b>class</b> . . . . .	340
10.11	Overloading the C++ Input Extraction (>>) and Output Insertion (<<) Operators . . . . .	343
10.12	Operators Which Cannot be Overloaded . . . . .	349
10.13	Restrictions Attached to Operator Overloading . . . . .	349
10.14	Choose Carefully Which Operators to Overload . . . . .	349
10.15	Summary . . . . .	350
	Exercises . . . . .	351
<b>11</b>	<b>Friends</b> . . . . .	353
11.1	<b>friend</b> Functions . . . . .	353
11.1.1	So Why Use <b>friend</b> Functions? . . . . .	355
11.1.2	<b>friend</b> Functions: Good, Bad or Ugly? . . . . .	358
11.2	<b>friend</b> Classes . . . . .	358
11.2.1	Vectors . . . . .	362
11.3	Friends and Overloaded Operators . . . . .	363
11.3.1	Complex Numbers . . . . .	364
11.3.2	Complex <b>class</b> . . . . .	366
11.3.3	The Use of Conversion Functions . . . . .	369
11.4	Overloading the C++ Input Extraction (>>) and Output Insertion (<<) Operators . . . . .	372
11.5	Summary . . . . .	373
	Exercises . . . . .	373
<b>12</b>	<b>Pointers</b> . . . . .	377
12.1	A First Look at Pointers . . . . .	377
12.2	The Address-of Operator . . . . .	379
12.3	Multiple Pointer Definitions in a Single Statement . . . . .	381
12.4	A Note on Pointer Definitions . . . . .	382
12.5	Another Look at Pointers . . . . .	382
12.6	Valid Pointers . . . . .	385
12.7	The NULL Pointer . . . . .	385
12.8	Don't Mix Types . . . . .	386
12.9	<b>void</b> Pointers . . . . .	387

12.10	<b>const</b> Pointers . . . . .	389
12.11	Pointers and Function Arguments . . . . .	390
12.12	Pointer Arithmetic . . . . .	393
12.12.1	Pointer Addition . . . . .	393
12.12.2	Pointer Subtraction . . . . .	394
12.12.3	Increment and Decrement Operators and Pointers . . . . .	395
12.13	Pointers and Relational Operators . . . . .	397
12.14	Pointers are Quicker . . . . .	398
12.15	Pointers and Arrays . . . . .	398
12.15.1	One-Dimensional Arrays . . . . .	399
12.15.2	Two-Dimensional Arrays . . . . .	400
12.16	Pointers, Arrays and Function Arguments . . . . .	401
12.17	Pointers to Pointers . . . . .	404
12.18	Arrays of Pointers and Pointers to Arrays . . . . .	405
12.19	Pointers to String Constants . . . . .	409
12.20	Pointers to Functions . . . . .	409
12.21	Returning a Pointer From a Function . . . . .	415
12.22	Casting . . . . .	416
12.23	Recap . . . . .	419
12.24	Pointers to Objects . . . . .	419
12.25	Pointers to Data Members and Member Functions . . . . .	422
12.26	The <b>new</b> and <b>delete</b> Operators . . . . .	424
12.26.1	Objects . . . . .	424
12.26.2	Arrays . . . . .	431
12.26.3	Objects, Arrays and <b>delete</b> . . . . .	437
12.26.4	Data Member Initialisation Lists and <b>new</b> . . . . .	437
12.26.5	Overloaded <b>new</b> and <b>delete</b> Operators . . . . .	438
12.26.6	Placement . . . . .	447
12.27	Returning Local Objects from Member Functions . . . . .	448
12.28	Passing Objects to Functions . . . . .	452
12.29	The <b>this</b> Pointer . . . . .	454
12.29.1	Further Illustration of <b>this</b> . . . . .	460
12.29.2	Restrictions on <b>this</b> . . . . .	462
12.30	A Matrix <b>class</b> . . . . .	462
12.30.1	A Subscript Operator for <b>class</b> Matrix . . . . .	476
12.30.2	Gaussian Elimination of a System of Linear Equations . . . . .	477
12.30.3	Return Value of <i>GaussElimination()</i> . . . . .	480
12.31	Summary . . . . .	481
	Exercises . . . . .	482
13	<b>Templates</b> . . . . .	485
13.1	Function Overloading . . . . .	485
13.2	Template Functions . . . . .	487
13.2.1	A Note on Style . . . . .	489
13.2.2	Templates or Macros? . . . . .	490
13.2.3	Template Function Arguments . . . . .	490
13.2.4	Multiple Type Arguments . . . . .	490
13.2.5	Template Functions do not Perform Implicit Casting . . . . .	491
13.2.6	Overloading Template Functions . . . . .	492
13.3	Template Classes . . . . .	493
13.3.1	Declaring Template Classes in a Header File . . . . .	500
13.3.2	<b>friend</b> Functions . . . . .	501

---

13.3.3	Multiple Type and Non-Type Arguments . . . . .	502
13.3.4	Overloading Template Classes? . . . . .	505
13.3.5	Combined Use of Template Functions and Classes . . . . .	509
13.3.6	Nested Classes . . . . .	510
13.3.7	The <b>typename</b> Keyword . . . . .	511
13.4	Smart Pointers . . . . .	512
13.5	A GlobalMemory Template <b>class</b> for Windows . . . . .	513
13.6	Vector and Matrix Template Classes . . . . .	515
13.7	Iterators . . . . .	530
13.7.1	Iterator Functions . . . . .	530
13.7.2	Iterator Classes . . . . .	535
13.8	A LinkedList <b>class</b> . . . . .	538
13.8.1	A LinkedListIterator <b>class</b> . . . . .	547
13.9	A Polygon <b>class</b> . . . . .	548
13.9.1	A <b>class</b> Representation of a Polygon . . . . .	548
13.9.2	A Point Inside or Outside a Polygon . . . . .	552
13.9.3	Convex and Concave Polygons, Sorting Vertices and Convex Hulls . . . . .	554
13.10	Summary . . . . .	563
	Exercises . . . . .	564
<b>14</b>	<b>Exception Handling</b> . . . . .	567
14.1	<b>try, catch</b> and <b>throw</b> . . . . .	567
14.2	Multiple <b>catch</b> Statements . . . . .	572
14.3	Handling all Exceptions . . . . .	573
14.4	Functions and Exception Handling . . . . .	574
14.4.1	Exception Specification . . . . .	576
14.5	Re-Throwing an Exception . . . . .	578
14.6	EXCEPT.H Header File . . . . .	579
14.6.1	The <i>terminate()</i> Function . . . . .	579
14.6.2	The <i>set_terminate()</i> Function . . . . .	579
14.6.3	The <i>unexpected()</i> Function . . . . .	581
14.6.4	The <i>set_unexpected()</i> Function . . . . .	581
14.6.5	The <i>xmsg</i> and <i>xalloc</i> Classes . . . . .	582
14.7	Constructors, Copy Constructors and Destructors . . . . .	583
14.8	Exception Handling and a Vector <b>class</b> . . . . .	584
14.8.1	Exception Classes . . . . .	590
14.8.2	Constructors . . . . .	591
14.8.3	<i>Assert()</i> <b>template</b> Function . . . . .	591
14.8.4	Declaring Exceptions . . . . .	592
14.9	Passing Information with an Exception . . . . .	593
14.9.1	Range and Index Classes . . . . .	597
14.10	Exception Handling and the <b>new</b> Operator . . . . .	606
14.11	Summary . . . . .	608
	Exercises . . . . .	609
<b>15</b>	<b>Inheritance</b> . . . . .	611
15.1	Base and Derived Classes . . . . .	612
15.2	Inheritance by Declaration . . . . .	614
15.3	Access Specifiers . . . . .	615
15.3.1	Default Specifier . . . . .	616
15.3.2	Access Declaration . . . . .	617
15.4	Properties of Base and Derived Classes . . . . .	618

15.5	Deriving from a Derived <b>class</b> . . . . .	619
15.6	<b>protected</b> Data Members . . . . .	621
15.7	<b>protected</b> Access Specifier . . . . .	622
15.8	Base and Derived <b>class</b> Constructors and Destructors . . . . .	623
15.8.1	No Derived <b>class</b> Constructors or Destructor . . . . .	623
15.8.2	Derived <b>class</b> Constructors and Destructor . . . . .	623
15.8.3	Use of the <code>xalloc</code> and <code>xmsg</code> Classes with the C++ string Library <b>class</b> . . . . .	629
15.9	Copy Constructors, Overloaded Assignment Operator and Inheritance . . . . .	630
15.9.1	Copy Constructors . . . . .	630
15.9.2	Overloaded Assignment Operator . . . . .	633
15.10	<b>new</b> and <b>delete</b> Operators and Inheritance . . . . .	637
15.11	OVERRIDING <b>class</b> Member Functions . . . . .	639
15.12	Overloaded Operator Functions and Inheritance . . . . .	643
15.13	Friendship and Inheritance . . . . .	643
15.14	Increased Functionality and Code Reusability . . . . .	644
15.15	A Vector <b>class</b> for Three-dimensional Space . . . . .	649
15.16	Containment Versus Inheritance . . . . .	655
15.16.1	Containment . . . . .	660
15.16.2	Inheritance . . . . .	675
15.16.3	Containment or Inheritance? . . . . .	680
15.17	Multiple Inheritance . . . . .	680
15.17.1	Constructors, Destructors and Multiple Inheritance . . . . .	682
15.18	<b>virtual</b> Base Classes . . . . .	686
15.19	<b>virtual</b> Functions . . . . .	689
15.19.1	Pure <b>virtual</b> Functions and Abstract Classes . . . . .	696
15.19.2	An Application . . . . .	700
15.20	Templates and Inheritance . . . . .	722
15.20.1	A SortedVector <b>class</b> . . . . .	729
15.20.2	Memory, LocalMemory and GlobalMemory Classes for Windows . . . . .	734
15.20.3	The Matrix <b>class</b> Revisited . . . . .	738
15.20.4	Stack and Queue Classes . . . . .	741
15.20.5	VectorIterator and LinkedListIterator Classes . . . . .	745
15.21	Exception Handling and Inheritance . . . . .	748
15.21.1	Exception Specification . . . . .	751
15.21.2	Re-Throwing an Exception . . . . .	752
15.22	Summary . . . . .	752
	Exercises . . . . .	753
 16	Run-Time Type Information and Casting . . . . .	757
16.1	Recap . . . . .	757
16.2	Use of <b>virtual</b> Functions . . . . .	759
16.2.1	A Direct Method . . . . .	759
16.2.2	The Double-Dispatch Method . . . . .	761
16.3	The RTTI Approach . . . . .	763
16.3.1	The <code>dynamic_cast&lt;&gt;()</code> Operator . . . . .	764
16.3.2	The <code>typeid()</code> Operator . . . . .	767
16.3.3	The TYPEINFO.H Header File and the <code>type_info</code> , <code>Bad_cast</code> and <code>Bad_typeid</code> Classes . . . . .	768
16.4	Casting . . . . .	770

---

16.4.1	The <code>static_cast&lt;&gt;()</code> Operator . . . . .	770
16.4.2	The <code>reinterpret_cast&lt;&gt;()</code> Operator . . . . .	772
16.4.3	The <code>const_cast&lt;&gt;()</code> Operator . . . . .	773
16.5	Summary . . . . .	774
	Exercises . . . . .	775
<b>17</b>	<b>Input and Output, Files and Streams . . . . .</b>	<b>777</b>
17.1	Why Wait Until Now? . . . . .	778
17.2	Streams and Stream Classes . . . . .	778
17.2.1	The <code>ios</code> Base <b>class</b> . . . . .	778
17.2.2	The <code>streambuf</code> Base <b>class</b> . . . . .	778
17.2.3	Stream Classes Derived from <code>ios</code> and <code>streambuf</code> . . . . .	778
17.2.4	Header Files . . . . .	781
17.2.5	Predefined Streams . . . . .	781
17.3	Standard Stream Output . . . . .	782
17.4	Standard Stream Input . . . . .	783
17.5	Overloading the Insertion and Extraction Operators . . . . .	785
17.6	Formatted Input and Output . . . . .	788
17.6.1	Formatting via Manipulators . . . . .	788
17.6.2	Formatting via Member Functions of <b>class</b> <code>ios</code> . . . . .	795
17.7	Working with Disk Files . . . . .	798
17.7.1	File Input Using <b>class</b> <code>ifstream</code> . . . . .	799
17.7.2	File Output Using <b>class</b> <code>ofstream</code> . . . . .	803
17.7.3	File Input and Output Using <b>class</b> <code>fstream</code> . . . . .	805
17.7.4	Stream Opening mode and <code>access</code> . . . . .	806
17.7.5	Stream Status Flags . . . . .	807
17.7.6	The <code>open()</code> and <code>close()</code> Member Functions . . . . .	809
17.7.7	The <code>read()</code> and <code>write()</code> Member Functions . . . . .	812
17.7.8	Random File Access . . . . .	813
17.7.9	Objects and Files . . . . .	817
17.8	<code>ReadFile</code> , <code>WriteFile</code> and <code>ReadAndWriteFile</code> Classes . . . . .	821
17.9	Reading a Collection of Objects from a Disk File . . . . .	830
17.10	String Streams . . . . .	846
17.11	Console Streams . . . . .	847
17.12	Redirection . . . . .	849
17.12.1	Redirecting Output from the Display Screen to a Disk File . . . . .	850
17.12.2	Redirecting Input from the Keyboard to a Disk File . . . . .	851
17.12.3	Redirecting both Input and Output . . . . .	851
17.13	Command Line Arguments . . . . .	851
17.14	The C Approach to Streams . . . . .	855
17.14.1	The <code>STDIO.H</code> Header File . . . . .	855
17.14.2	Predefined Streams . . . . .	855
17.14.3	Output to the Display Screen Using the <code>printf()</code> Function . . . . .	856
17.14.4	Input from the Keyboard Using the <code>scanf()</code> Function . . . . .	860
17.14.5	The <code>vprintf()</code> , <code>vscanf()</code> , <code>vsprintf()</code> and <code>vsscanf()</code> Functions . . . . .	867
17.14.6	File Input and Output . . . . .	867
17.14.7	The <code>FILE</code> Structure . . . . .	867
17.14.8	The <code>fopen()</code> Function . . . . .	868

17.14.9	The <i>fclose()</i> Function . . . . .	868
17.14.10	The <i>fgetc()</i> Function . . . . .	869
17.14.11	Reading a Disk File . . . . .	869
17.14.12	The <i>fgets()</i> Function . . . . .	871
17.14.13	The <i>fputc()</i> Function . . . . .	872
17.14.14	Writing to a Disk File . . . . .	872
17.14.15	The <i>fputs()</i> Function . . . . .	873
17.14.16	The <i>fread()</i> and <i>fwrite()</i> Functions . . . . .	874
17.14.17	The <i>fprintf()</i> and <i>fscanf()</i> Functions . . . . .	876
17.14.18	The <i>vfprintf()</i> and <i>vscanf()</i> Functions . . . . .	876
17.14.19	Random File Access . . . . .	876
17.14.20	The <i>fseek()</i> , <i>ftell()</i> , <i>fgetpos()</i> and <i>fsetpos()</i> Functions . . . . .	877
17.14.21	The <i>feof()</i> , <i>ferror()</i> , <i>fflush()</i> , <i>flushall()</i> , <i>freopen()</i> , <i>clearerr()</i> , <i>remove()</i> , <i>rename()</i> and <i>rewind()</i> Functions . . . . .	879
17.15	A Device-Independent Bitmap <b>class</b> for Windows . . . . .	880
17.15.1	The Windows Device-Independent Bitmap File Format . . . . .	881
17.15.2	The DIBitmap <b>class</b> . . . . .	883
17.16	Summary . . . . .	900
	Exercises . . . . .	901
18	<b>The Preprocessor</b> . . . . .	903
18.1	The Preprocessor . . . . .	903
18.2	Preprocessor Directives . . . . .	903
18.2.1	The # Null Directive . . . . .	904
18.2.2	The #define Directive . . . . .	904
18.2.3	The #error Directive . . . . .	907
18.2.4	The #include Directive . . . . .	908
18.2.5	The #line Directive . . . . .	908
18.2.6	The #pragma Directive . . . . .	909
18.2.7	The #undef Directive . . . . .	909
18.2.8	The #elif, #else, #endif, #if, #ifdef and #ifndef Conditional Directives . . . . .	910
18.3	The # and ## Operators . . . . .	912
18.4	Predefined Macros . . . . .	912
18.5	Summary . . . . .	915
	Exercises . . . . .	915
19	<b>Namespaces</b> . . . . .	917
19.1	Namespaces – What are They and Why do we Need Them? . . . . .	917
19.2	The <b>namespace</b> Declaration . . . . .	918
19.3	The <b>using</b> Declaration . . . . .	922
19.4	The <b>using</b> Directive . . . . .	923
19.5	The Nameless <b>namespace</b> Declaration . . . . .	924
19.6	Global Scope . . . . .	925
19.7	Overloading Namespaces . . . . .	925
19.8	Namespaces and Inheritance . . . . .	926
19.9	Summary . . . . .	930
	Exercises . . . . .	930

---

<b>20 An Application: a Simple Raytracing Program . . . . .</b>	<b>933</b>
20.1 Why an Application and Why Raytracing? . . . . .	934
20.2 Recursive Raytracing . . . . .	934
20.3 Tool Classes . . . . .	936
20.4 Ray-object intersections . . . . .	953
20.4.1 Ray-Plane Intersection . . . . .	955
20.4.2 Ray-Polygon, Ray-Triangle and Ray-Quadrilateral Intersections . . . . .	962
20.4.3 Ray-Tetrahedra Intersection . . . . .	969
20.4.4 Ray-Sphere Intersection . . . . .	971
20.4.5 Ray-Circle Intersection . . . . .	975
20.4.6 Other Ray-Object Intersections . . . . .	979
20.5 The Viewer . . . . .	980
20.6 The View Plane and Screen . . . . .	981
20.7 Object Surfaces, Illumination and Reflectance . . . . .	986
20.7.1 Light Source . . . . .	988
20.7.2 Ambient Light . . . . .	989
20.7.3 Diffuse Reflection . . . . .	990
20.7.4 Specular Reflection . . . . .	992
20.7.5 Reflected and Transmitted Light . . . . .	994
20.7.6 Distance Attenuation . . . . .	996
20.7.7 Shadows . . . . .	996
20.7.8 Simplifications and Extensions to the Surface Illumination and Reflectance Model . . . . .	998
20.8 The World . . . . .	998
20.8.1 Object and World Data Files . . . . .	1006
20.9 Windows Program . . . . .	1008
20.10 Summary . . . . .	1016
Exercises . . . . .	1016
<b>Conclusion . . . . .</b>	<b>1019</b>
<b>Appendices</b>	
A C++ Keywords . . . . .	1021
B ASCII Character Set . . . . .	1023
C Operators: Precedence, Associativity and Arity . . . . .	1029
D Glossary . . . . .	1031
<b>References . . . . .</b>	<b>1035</b>
<b>Index . . . . .</b>	<b>1039</b>

# List of Programs

---

Programs are listed in the order in which they appear.

## Chapter 3

MAIN.CPP	the simplest of C++ programs.
INFO.H	header file of information.
CAPITALM.CPP	illustrates that C++ is a case sensitive language.

## Chapter 4

CHAR.CPP	illustrates the <b>char</b> data type.
INT.CPP	illustrates the <b>int</b> data type.
FLOAT.CPP	illustrates the <b>float</b> data type.
DOUBLE.CPP	illustrates the <b>double</b> data type.
BOOL.CPP	illustrates <b>bool</b> .
LIMITS.CPP	examines LIMITS.H.
FLOATS.CPP	examines FLOAT.H.
MANIP.CPP	illustrates the stream manipulators.
OFILE.CPP	output of strings to a file.
OFILE1.CPP	output of integers to a file.
IFILE.CPP	get integers from a file.
AUTO.CPP	illustrates the <b>auto</b> storage class specifier.
FILE.CPP	some variable definitions.
EXTERN.CPP	illustrates the <b>extern</b> storage class specifier.
REGISTER.CPP	illustrates the <b>register</b> storage class specifier.
FILE1.CPP	some <b>static</b> variable definitions.
STATIC.CPP	illustrates the <b>static</b> storage class specifier.
ASM.CPP	illustrates the <b>asm</b> keyword.

## Chapter 5

IF.CPP	illustrates the <b>if</b> statement.
IFIF.CPP	illustrates the <b>if</b> statement.
IF_ELSE.CPP	illustrates the <b>if-else</b> statement.
IF_COMP.CPP	illustrates the <b>if-else</b> compound statement.
IF_MANIP.CPP	illustrates the <b>if-else</b> statement and the dec, hex and oct stream manipulators.
IF_NEST.CPP	illustrates nested <b>ifs</b> .
LOG_AND.CPP	illustrates the logical AND operator (&&).
LOG_OR.CPP	illustrates the logical OR operator (  ).
LOG_NOT.CPP	illustrates the logical NOT operator (!).
SWITCH.CPP	illustrates the <b>switch</b> statement.
B_SWITCH.CPP	illustrates the <b>switch</b> statement and the keyword <b>break</b> .
DEFAULT.CPP	illustrates the <b>switch</b> statement, <b>break</b> and <b>default</b> .
COND.CPP	illustrates the conditional operator (? :).
FOR.CPP	illustrates the <b>for</b> -loop.

WHILE.CPP	illustrates the <b>while</b> -loop.
DO_WHILE.CPP	illustrates the <b>do-while</b> loop.
B_FOR.CPP	illustrates the <b>for</b> -loop and <b>break</b> statement.
CONTINUE.CPP	illustrates the <b>continue</b> statement.
BIT_AND.CPP	illustrates the bitwise AND operator (&).
BIT_OR.CPP	illustrates the bitwise OR operator ( ).
BIT_XOR.CPP	illustrates the bitwise XOR operator (^).
BIT_ONE.CPP	illustrates the bitwise one's complement operator (~) by finding the two's complement of a number.
SHIFT.CPP	illustrates the left and right shift operators << and >>.
BS_ASSGN.CPP	illustrates the bitwise and shift assignment operators.

### Chapter 6

FUNC.CPP	illustrates a C++ function.
FUNC_DEC.CPP	illustrates a C++ function declaration.
L_SCOPE.CPP	illustrates local scope.
G_SCOPE.CPP	illustrates global scope.
RES_OP.CPP	illustrates the scope resolution operator :: .
P_VALUE.CPP	illustrates the passing of arguments by value.
P_REF.CPP	illustrates the passing of arguments by reference.
P_CONST.CPP	illustrates the use of <b>const</b> for passing arguments by reference.
RETURN2.CPP	illustrates returning more than 1 value.
R_REF.CPP	illustrates returning by reference.
MY_LIB.CPP	implementation file for my library of functions.
MY_LIB.H	header file for MY_LIB.CPP.
NEW_FUNC.CPP	demonstrates the use of implementation and header files.
OVER.CPP	illustrates function overloading.
OVERLOAD.CPP	illustrates function overloading.
GET_SET.CPP	illustrates function overloading.
DEF_ARG.CPP	illustrates default function arguments.
STATIC.CPP	illustrates local <b>static</b> variables.
INLINE.CPP	illustrates an <b>inline</b> function.
FILE.CPP	a function definition.
EXTERN.CPP	illustrates the <b>extern</b> storage class specifier.
EXTERN1.CPP	further illustrates the <b>extern</b> storage class specifier.
LI_LIB.CPP	implementation of clockwise/anticlockwise and line intersection functions.
LI_LIB.H	header file for LI_LIB.CPP.
LI.CPP	intersection of two two-dimensional line segments.

### Chapter 7

ARRAY.CPP	illustrates a simple one-dimensional array.
STATIC.CPP	illustrates that a C++ array must be defined constant.
A_SIZE.CPP	illustrates the <b>sizeof</b> operator for determining the number of elements and size of an array.
INDEXING.CPP	illustrates indexing an array out of bounds.
INITIAL.CPP	illustrates initialising arrays.
ARRAY_2D.CPP	illustrates two-dimensional arrays.
ARRAY_3D.CPP	illustrates three-dimensional arrays.
V_STRING.CPP	illustrates variable strings.

---

CIN_GET.CPP	illustrates the <code>istream::get()</code> function.
C_STRING.CPP	illustrates constant strings.
INIT_STR.CPP	illustrates initialising a 2D array of strings.
FUNC_ARG.CPP	illustrates passing of array to functions.
ADDRESS.CPP	examines an array's address.
F_ARG_2D.CPP	illustrates passing two-dimensional arrays as function arguments.
STRCMP.CPP	illustrates passing strings as function arguments.
ARG.CPP	illustrates that a function does not know the size of an array passed as an argument.
<b>Chapter 8</b>	
STRUCT.CPP	illustrates structures.
SCOPE.CPP	illustrates a structure's scope.
SIZEOF.CPP	illustrates determining the size of a structure.
ASSIGN.CPP	illustrates assignment of structure variables.
ARRAY.CPP	illustrates an array of structures.
FUNC_ARG.CPP	illustrates a structure function argument.
NEST.CPP	illustrates nested structures.
POINT.CPP	illustrates a Point structure.
DAT&FUN.CPP	illustrates a structure declaration with data function members.
PLT.CPP	Point, Line and Triangle structures.
BIT_FLD.CPP	illustrates bit fields.
UNION.CPP	illustrates unions.
ENUM.CPP	illustrates enum.
TYPEDEF.CPP	illustrates the <code>typedef</code> .
<b>Chapter 9</b>	
CLASS_1.CPP	introduces the <code>class</code> .
PRI_PUB.CPP	illustrates <code>private</code> and <code>public</code> data members.
PUBPOINT.CPP	a Point <code>class</code> with <code>public</code> data members.
MEMPOINT.CPP	illustrates <code>class</code> member functions.
MEM_OVER.CPP	illustrates overloaded <code>class</code> member functions.
OUTSIDE.CPP	a Point <code>class</code> with member functions defined outside the <code>class</code> declaration.
PT_LIB.H	header file for Point <code>class</code> .
PT_LIB.CPP	implementation file for Point <code>class</code> .
MY_PROG.CPP	application of Point <code>class</code> .
STATIC.CPP	illustrates <code>static</code> data and function members of a <code>class</code> .
AU_EX_RE.CPP	illustrates that the storage class specifiers <code>auto</code> , <code>extern</code> and <code>register</code> cannot be used for declaring <code>class</code> data members.
CON_MUT.CPP	illustrates <code>const</code> and <code>mutable</code> .
VOL.CPP	illustrates <code>volatile</code> data members.
BIT_FLD.PP	illustrates bit fields.
CONSTR.CPP	illustrates constructors.
OVER_CON.CPP	illustrates overloaded constructors.
DEF_CON.CPP	illustrates the use of default constructors.
CON_INIT.CPP	illustrates data member initialisation.
COPY_CON.CPP	illustrates the copy constructor.
OVER_CC.CPP	overloads Point's copy constructor.

COPY_FUN.CPP	illustrates the copy constructor with function arguments and return values.
CON_CV.CPP	illustrates constructors, <b>const</b> and <b>volatile</b> .
DESTR.CPP	illustrates destructors.
DES_CV.CPP	illustrates destructors, <b>const</b> and <b>volatile</b> .
INLINE.CPP	illustrates <b>inline</b> member functions.
AUTO_IN.CPP	illustrates automatic <b>inline</b> member functions.
CON_MEM.CPP	illustrates <b>const</b> member functions.
CON_MEM1.CPP	further illustrates <b>const</b> member functions.
VOL_MEM.CPP	illustrates <b>volatile</b> member functions.
DEF_ARG.CPP	illustrates default member function arguments.
ARG_RET.CPP	illustrates passing object arguments to functions and returning an object from a function.
ARG_RET2.CPP	illustrates passing object arguments to functions and assigning values to an object's data members.
RET_REF.CPP	illustrates returning by reference for both constant and non-constant objects.
NO_OBJ.CPP	illustrates a constructor and member function call in a single statement.
ARRAY_OB.CPP	illustrates an array of objects.
LOCAL.CPP	illustrates local classes.
NESTED.CPP	illustrates nested classes.
NESTED1.CPP	further illustrates nested classes.
LI_CLASS.CPP	Point and Line classes with Point clockwise/anticlockwise and Line intersection routines.

## Chapter 10

ADD.CPP	attempts to add two Point objects using the addition operator + before overloading.
OVER_AS.CPP	overloads the arithmetic (+) and (-) operators for <b>class Point</b> .
ARITH.CPP	overloads the arithmetic and arithmetic assignment operators.
INC&DEC.CPP	overloads the prefix and postfix increment and decrement operators.
COMPARE.CPP	overloads the relational operators.
ASSIGN.CPP	illustrates overloading the assignment operator.
NON_MEM.CPP	illustrates non-member overloaded operator functions.
SUBSCRPT.CPP	illustrates overloading the subscript operator.
COMPOSIT.CPP	illustrates overloading an array indexing operator.
CONVERT.CPP	illustrates conversions between different types.
EXPLICIT.CPP	illustrates the <b>explicit</b> specifier.
OVER_IO.CPP	overloads C++'s insertion, <<, and extraction, >>, operators.
OVER_IO1.CPP	further illustrates overloading the insertion and extraction operators.
OVER_IO2.CPP	another approach to overloading the insertion and extraction operators.
STRING.CPP	illustrates a String <b>class</b> .

## Chapter 11

FRI_FUNC.CPP	illustrates <b>friend</b> functions.
ABC.CPP	illustrates a function which is a <b>friend</b> of three classes A, B and C.
FR_CLASS.CPP	illustrates <b>friend</b> classes.

---

FR_TRAN.CPP	illustrates that friendship is not transitive.
FR_OP.CPP	illustrates operator overloading using <b>friend</b> functions.
FR_CPLX0.CPP	illustrates overloaded operators using <b>friend</b> functions via a Complex <b>class</b> .
FR_CPLX1.CPP	illustrates overloaded operators using <b>friend</b> functions via a Complex <b>class</b> and a conversion function.
FR_IO.CPP	illustrates overloading C++'s insertion and extraction operators for <b>class</b> Complex.
<b>Chapter 12</b>	
PTR_1ST.CPP	illustrates pointer variables.
AOO.CPP	illustrates the address-of operator &.
PTR.CPP	illustrates the pointer variables and the address-of operator &, another look at pointers.
PTR_NEXT.CPP	illustrates the NULL pointer.
NULL.CPP	illustrates the mixing of types with pointers.
MIX.CPP	illustrates pointers to <b>void</b> .
PTR_VOID.CPP	illustrates the use of <b>const</b> with pointers
CONST.CPP	illustrates passing function arguments by pointer.
FUNC_ARG.CPP	illustrates pointer addition.
P_ADD.CPP	illustrates pointer subtraction.
P_SUB.CPP	illustrates the increment operator and pointers.
P_INC.CPP	illustrates pointers and arrays.
P&ARRAY.CPP	illustrates incrementing a pointer.
P_INC1.CPP	illustrates pointers and two-dimensional arrays.
P_A_2D.CPP	illustrates pointers, arrays and function arguments.
PAAFA.CPP	illustrates passing a general two-dimensional array to a function.
F_P_A_2D.CPP	illustrates a pointer to a pointer.
PTR_PTR.CPP	illustrates an array of pointers.
ARRAY_P.CPP	illustrates arrays of pointers to strings.
A_P_STR.CPP	illustrates pointers to arrays.
P_ARRAY.CPP	illustrates pointers to string constants.
PTR_STR.CPP	header file for Newton-Raphson method.
NR.H	implementation of Newton-Raphson method.
NR.CPP	illustrates pointers to functions.
P_FUNC.CPP	illustrates functions that return a pointer.
F_RET_P.CPP	illustrates casting pointers.
CAST_P.CPP	illustrates pointers to objects.
P_OBJ.CPP	illustrates the pointer-to-member operators . * and ->*.
PTR_MEM.CPP	illustrates the <b>new</b> and <b>delete</b> operators applied to objects.
ND_OBJ.CPP	illustrates the <b>delete</b> operator.
DELETE.CPP	illustrates the <b>new</b> and <b>delete</b> operators with a String <b>class</b> .
ND_STR.CPP	illustrates arrays of pointers to String objects.
A_P_STR1.CPP	a Vector <b>class</b> that illustrates the use of the <b>new</b> and <b>delete</b> operators for dynamically allocating and deallocating memory for arrays.
VECT.CPP	a Vector <b>class</b> that caters for an increasing/decreasing number of elements.
VECT1.CPP	

NEW_INIT.CPP	illustrates the combined use of the <b>new</b> operator with a <b>class</b> 's constructor data member initialisation list
OVER_ND.CPP	illustrates overloading the <b>new</b> and <b>delete</b> operators.
WIN_MEM.CPP	illustrates a global memory <b>class</b> for programming in a Windows environment.
PLACEMNT.CPP	illustrates placement.
RET_PROB.CPP	illustrates a problem with returning an object from a function.
PASS_PROB.CPP	illustrates a problem with passing an object by value to a function.
THIS.CPP	illustrates the <b>this</b> pointer.
THIS_PT.CPP	illustrates the <b>this</b> pointer applied to <b>class</b> Point.
THIS_STR.CPP	illustrates the use of the <b>this</b> pointer with the <b>operator=()</b> function for <b>class</b> String.
THIS_EXP.CPP	experiments with the <b>this</b> pointer.
MTRX.CPP	a Matrix <b>class</b> .
VECTOR.H	header file for Vector <b>class</b> .
VECTOR.CPP	implementation file for Vector <b>class</b> .
MATRIX.H	header file for Matrix <b>class</b> .
MATRIX.CPP	implementation file for Matrix <b>class</b> .
MTRX_TST.CPP	tests the modified Matrix <b>class</b> which constructs an array of Vector objects and allows C++-style subscripting of Matrix elements.

### Chapter 13

FUN_OVER.CPP	implements a <i>Max()</i> function with different argument types by function overloading.
TMP_FUN.CPP	implements a <i>Max()</i> function as a <b>template</b> function.
TMP_DEC.CPP	illustrates a <b>template</b> function declaration.
MULT_ARG.CPP	illustrates multiple type arguments for <b>template</b> functions.
DIF_TYPE.CPP	illustrates that <b>template</b> functions do not perform implicit casting.
TMP_OVER.CPP	illustrates that <b>template</b> functions can be overloaded.
TMP_O1.CPP	illustrates overloading a <b>template</b> function.
PT_TMP.CPP	illustrates a Point <b>template class</b> .
PT_TMP.H	<b>template class</b> Point.
TMP_TST.CPP	tests the <b>template class</b> Point which is declared and defined in the header file PT_TMP.H.
FRI_FUNC.CPP	illustrates a <b>friend</b> function of a <b>template class</b> .
MTCA.CPP	multiple <b>template class</b> arguments.
EXP_TMP.CPP	illustrates that a <b>template class</b> can have an explicit <b>template class</b> declaration
TMP_C_F.CPP	illustrates the use of a <b>template class</b> as an argument to a <b>template</b> function.
NESTED.CPP	illustrates nested <b>template</b> classes.
TYPENAME.CPP	illustrates <b>typename</b> .
SMRT_PTR.CPP	illustrates smart pointers.
WIN_TMP.CPP	illustrates a global memory <b>template class</b> for programming in a Windows environment.
VEC_TMP.H	<b>template class</b> Vector.
MTX_TMP.H	<b>template class</b> Matrix.
MTMP_TST.CPP	tests the Point, Vector and Matrix <b>template</b> classes.

---

V_IT.F.H	<b>template class</b> Vector with iterators.
IT_F_TST.CPP	tests the <b>template class</b> Vector with iterators.
V_IT.C.H	<b>template class</b> VectorIterator.
IT_C_TST.CPP	tests the <b>template class</b> VectorIterator.
LL.H	<b>template class</b> LinkedList.
LL_TST.CPP	tests the LinkedList and LinkedListIterator <b>template</b> classes.
LLI.H	<b>template class</b> LinkedListIterator.
POLY.H	<b>class</b> Polygon.
POLY.CPP	Polygon implementation file.
PT&LINE.CPP	classes Point and Line.
PT&LINE.CPP	Point and Line implementation file.
POLY_TST.CPP	test <b>class</b> Polygon.
<b>Chapter 14</b>	
TCT.CPP	introduces the <b>try</b> , <b>catch</b> and <b>throw</b> keywords.
TERM.CPP	illustrates abnormal program termination.
TYPES.CPP	illustrates different handler types.
MULT.CPP	illustrates multiple <b>catch</b> statements.
CATCH_AL.CPP	illustrates the <b>catch-all</b> statement.
INT&ALL.CPP	illustrates both <b>catch-int</b> and <b>catch-all</b> statements.
OUT_TRY.CPP	illustrates throwing an exception from outside a <b>try</b> block.
TCT_FUNC.CPP	illustrates the <b>try-catch-throw</b> setup from within a function.
EX_SPEC.CPP	illustrates exception specifications.
RETHROW.CPP	illustrates re-throwing an exception.
SET_TERM.CPP	illustrates the <b>set_terminate()</b> function.
SET_UNIX.CPP	illustrates the <b>set_unexpected()</b> function.
CON&DES.CPP	illustrates calls to constructors, copy constructors and destructors with exception handling.
VEC_X.H	<b>template class</b> Vector with exception handling.
VEC_TST.CPP	tests the exception handling features of the <b>template class</b> Vector.
VEC_X1.H	<b>template class</b> Vector with exception handling.
VEC_TST1.CPP	tests the exception handling features of the <b>template class</b> Vector having modified the Memory, Size and Range exception classes.
ASSERT.H	<b>Assert()</b> <b>template</b> function.
RANGE.H	<b>class</b> Range.
INDEX.H	<b>class</b> Index.
RNG_INDX.CPP	illustrates the use of Range and Index classes with a Vector <b>template class</b> .
X_NEW.CPP	illustrates exception handling and the <b>new</b> operator.
SET_HAND.CPP	illustrates the <b>set_new_handler()</b> function.
<b>Chapter 15</b>	
B&D.CPP	introduces base and derived classes.
ACC_SPEC.CPP	examines the <b>public</b> and <b>private</b> access specifiers for derived classes.
CHNG_ACC.CPP	illustrates changing an access specification associated with data members of a derived <b>class</b> .

PROPS.CPP	illustrates certain properties of inheritance.
PROT.CPP	illustrates the <b>protected</b> keyword.
DER_DER.CPP	illustrates deriving from a derived <b>class</b> .
PRO_ACCS.CPP	examines the <b>protected</b> access specifier for derived classes.
NO_D_C&D.CPP	illustrates a derived <b>class</b> with no constructor or destructor.
D_C&D.CPP	illustrates a derived <b>class</b> with constructor and destructor.
D_C&D&D.CPP	illustrates a derived <b>class</b> with constructor, destructor and data members.
ODD_C&D.CPP	further illustrates a derived <b>class</b> with constructor and destructor.
D_TO_B.CPP	illustrates passing constructor argument objects from a derived <b>class</b> constructor to a base <b>class</b> constructor.
D_TO_B1.CPP	furher illustrates passing constructor argument objects from a derived <b>class</b> constructor to a base <b>class</b> constructor.
CC_INH.CPP	illustrates a base <b>class</b> copy constructor and inheritance.
CC_INH1.CPP	illustrates the use of both base and derived <b>class</b> copy constructors and inheritance.
OP_INH.CPP	illustrates the use of a base <b>class</b> overloaded assignment operator and inheritance.
OP_INH1.CPP	illustrates the use of both base and derived <b>class</b> overloaded assignment operators and inheritance.
OP_INH2.CPP	illustrates base and derived <b>class</b> overloaded assignment operator functions and inheritance
OP_INH3.CPP	illustrates the use of a copy constructor and overloaded assignment operator function for both base and derived classes.
ND_INH.CPP	illustrates overloading the <b>new</b> and <b>delete</b> operators.
OVERRIDE.CPP	illustrates derived <b>class</b> member functions which override base <b>class</b> member functions.
OO.CPP	illustrates overriding base <b>class</b> overloaded member functions.
SCPE_RES.CPP	illustrates the use of the scope resolution operator to access an overridden base <b>class</b> member function from a derived <b>class</b> .
FR_INH.CPP	illustrates friendship and inheritance.
FR_INH1.CPP	furher illustrates friendship and inheritance.
POINT.H	<b>class</b> Point.
POINT.CPP	implementation file for <b>class</b> Point.
NUM_PT.CPP	a numbered Point <b>class</b> .
VEC.H	header file for Vector <b>class</b> .
VEC.CPP	implementation file for Vector <b>class</b> .
VEC3D.H	header file for <b>class</b> Vector3D.
VEC3D.CPP	implementation file for Vector3D <b>class</b> .
VEC_TST.CPP	tests base and derived classes Vector and Vector3D.
PT.H	<b>class</b> Point.
PT.CPP	implementation file for Point <b>class</b> .
LINE.H	header file for Line <b>class</b> .
LINE.CPP	implementation file for <b>class</b> Line.
TRI_CON.H	header file for Triangle <b>class</b> to illustrate containment.
TRI_CON.CPP	implementation file for Triangle <b>class</b> .
TET_CON.H	header file for Tetrahedra <b>class</b> to illustrate containment.
TET_CON.CPP	implementation file for Tetrahedra <b>class</b> .
TTT_CON.CPP	tests the Triangle and Tetrahedra classes.

---

TRI_INH.H	header file for Triangle <b>class</b> to illustrate inheritance.
TRI_INH.CPP	implementation file for Triangle <b>class</b> .
TET_INH.H	header file for Tetrahedra <b>class</b> to illustrate inheritance.
TET_INH.CPP	implementation file for Tetrahedra <b>class</b> .
TTT_INH.CPP	tests the Triangle and Tetrahedra classes.
MULT_INH.CPP	illustrates multiple inheritance.
M_NO_C&D.CPP	illustrates a <b>class</b> with no constructor and destructor derived from two base classes.
M_C&D.CPP	illustrates a <b>class</b> with a constructor and destructor derived from two base classes.
D_TO_BS.CPP	illustrates passing constructor argument objects from a derived <b>class</b> constructor to base <b>class</b> constructors.
M_PROB.CPP	illustrates ambiguous accessing of data members by a derived <b>class</b> .
M_PROB1.CPP	further illustrates ambiguous accessing of data members by a derived <b>class</b> .
V_B_CLSS.CPP	illustrates <b>virtual</b> base classes.
NON_VIRT.CPP	illustrates the use of pointers to non- <b>virtual</b> member functions.
VIRT.CPP	illustrates the use of <b>virtual</b> member functions.
V_HIER.CPP	illustrates that <b>virtual</b> functions are hierarchical.
V_DEST.CPP	illustrates the need for <b>virtual</b> destructors
PURE&ABS.CPP	illustrates pure <b>virtual</b> functions and abstract classes.
SHAPE.H	header file for Shape <b>class</b> .
PLANE_V.H	header file for Plane <b>class</b> .
PLANE_V.CPP	implementation file for Plane <b>class</b> .
POLY_V.H	header file for Polygon <b>class</b> .
POLY_V.CPP	implementation file for Polygon <b>class</b> .
TRI_V.H	header file for Triangle <b>class</b> .
TRI_V.CPP	implementation file for Triangle <b>class</b> .
RECT_V.H	header file for Rect <b>class</b> .
RECT_V.CPP	implementation file for Rect <b>class</b> .
CIRC_V.H	header file for Circle <b>class</b> .
CIRC_V.CPP	implementation file for Circle <b>class</b> .
TET_V.H	header file for Tetrahedra <b>class</b> .
TET_V.CPP	implementation file for Tetrahedra <b>class</b> .
SHAPES.CPP	tests abstract base <b>class</b> Shape and derived classes.
TMP_INH.CPP	illustrates the use of <b>template</b> classes and inheritance.
RV_TMP.H	<b>template class</b> Vector.
RV_TST.CPP	tests the <b>template class</b> RVector, which is inherited from <b>template class</b> Vector.
CONVERT.CPP	illustrates inheritance, <b>template</b> classes and conversion.
VEC_PTR.H	Vector of pointers <b>template class</b> .
PTRV_TST.CPP	tests the <b>template class</b> PtrVector, which is inherited from <b>template class</b> Vector.
VEC_SORT.H	<b>template class</b> SortedVector.
SORT_TST.CPP	tests the <b>template class</b> SortedVector, which is inherited from <b>template class</b> Vector.
WIN_LGM.CPP	illustrates local and global memory <b>template</b> classes for programming in a Windows environment.
MTX_TMP.H	<b>template class</b> Matrix and floating-point classes DoubleMatrix and FloatMatrix.

MTX_TST.CPP	tests the Point, Vector and Matrix <b>template</b> classes and the DoubleMatrix and FloatMatrix classes. <b>template class</b> Stack.
QUEUE.H	<b>template class</b> Queue.
S_Q_TST.CPP	tests the LinkedList <b>template class</b> .
ITER.H	<b>template class</b> Iterator.
V.H	<b>template class</b> Vector.
VI.H	<b>template class</b> VectorIterator.
LL.H	<b>template class</b> LinkedList.
LLI.H	<b>template class</b> LinkedListIterator.
ITER_TST.CPP	tests the iterator classes VectorIterator and LinkedListIterator.
EXP_INH.CPP	illustrates exception handling and inheritance.
BD_X_INH.CPP	illustrates base and derived classes, exception handling, and inheritance.
B_PRI_D.CPP	illustrates base and privately derived classes, exception handling, and inheritance.
BD_CATCH.CPP	illustrates catching a derived <b>class</b> exception.
X_SP_INH.CPP	illustrates function exception specifications and inheritance.
 <b>Chapter 16</b>	
WHO.CPP	illustrates detecting the type of an object through the use of an abstract base <b>class</b> code and type casting an abstract base <b>class</b> pointer.
DBL_DIS.CPP	illustrates double dispatch.
DYN_CSTP.CPP	illustrates the <code>dynamic_cast&lt;T*&gt;()</code> operator.
DYN_CSTR.CPP	illustrates the <code>dynamic_cast&lt;T&amp;&gt;()</code> operator.
V_B_C.CPP	illustrates the use of the <code>dynamic_cast&lt;T*&gt;()</code> operator with <b>virtual</b> base classes.
TYPEID.CPP	illustrates the use of the <code>typeid()</code> operator.
TYPEINFO.CPP	illustrates the <code>Type_info</code> <b>class</b> .
STAT_CST.CPP	illustrates the <code>static_cast&lt;&gt;()</code> operator.
REIN_CST.CPP	illustrates the <code>reinterpret_cast&lt;&gt;()</code> operator.
CNST_CST.CPP	illustrates the <code>const_cast&lt;&gt;()</code> operator.
 <b>Chapter 17</b>	
COUT.CPP	illustrates the <code>cout</code> stream object and insertion operator <code>&lt;&lt;</code> .
CIN.CPP	illustrates the <code>cin</code> stream object and extraction operator <code>&gt;&gt;</code> .
USR_TYPE.CPP	illustrates overloading the insertion and extraction operators for a user-defined <b>class</b> .
IN_OUT.CPP	further illustrates overloading the insertion and extraction operators for <b>class</b> Point.
MANIP.CPP	illustrates the C++ stream manipulators.
MY_MANIP.CPP	illustrates user-defined non-parametrised stream manipulators.
P_MANIP.CPP	illustrates user-defined parametrised stream manipulators.
P_MANIP1.CPP	an alternative approach to user-defined parametrised stream manipulators.
IOS_F_MF.CPP	illustrates the format flags and the <code>setf()</code> , <code>unsetf()</code> and <code>flags()</code> member functions of <b>class</b> ios.
WIDTH.CPP	illustrates the <code>ios::width()</code> member function.

---

IFSTR.CPP	illustrates the <code>ifstream</code> stream <b>class</b> for handling file input.
IFSTR1.CPP	illustrates the <code>ifstream</code> stream <b>class</b> for handling file input and the <code>istream::getline()</code> member functions.
NO_LINES.CPP	illustrates the use of the <code>istream::getline()</code> member function to determine the number of lines of a file.
OFSTR.CPP	illustrates the <code>ofstream</code> stream <b>class</b> for handling file output.
FSTR.CPP	illustrates the <code>fstream</code> stream <b>class</b> for handling file input and output.
IOS_STAT.CPP	illustrates the <code>ios</code> stream status flags and associated member functions.
EOF.CPP	illustrates the <code>ios::eof()</code> member function for detecting the end of a file.
OP_CL.CPP	illustrates the <code>open()</code> and <code>close()</code> stream functions.
OP_OR_CL.CPP	illustrates the <code>is_open()</code> function for determining whether a file is open or closed.
R&W.CPP	illustrates the <code>read()</code> , <code>write()</code> and <code>gcount()</code> member functions.
CGP.CPP	illustrates the <code>seekg()</code> and <code>tellg()</code> member functions of <b>class</b> <code>istream</code> for extracting the current get position.
R&W1.CPP	further illustrates the <code>read()</code> , <code>write()</code> and <code>gcount()</code> member functions.
OB_FILE.CPP	illustrates the use of user-defined objects of <b>class</b> <code>NumberedPoint</code> and an input file stream. <b>template class</b> <code>Point</code> .
PT_TMP.H	a numbered Point <b>template class</b> .
NPT_TMP.H	further illustrates the use of user-defined objects of <b>class</b> <code>NumberedPoint</code> and an input file stream.
OB_FILE1.CPP	ReadFile, WriteFile and ReadAndWriteFile <b>class</b> declarations.
RW_FILE.H	implementation file for classes ReadFile, WriteFile and ReadAndWriteFile.
RW_FILE.CPP	tests the ReadFile, WriteFile and ReadAndWriteFile classes.
RW_TST.CPP	a World <b>class</b> .
WORLD.H	implementation file for <b>class</b> World.
WORLD.CPP	tests the World <b>class</b> by reading a set of objects from a disk file.
READ_OBJ.CPP	header file for Shape <b>class</b> .
SHAPE_VH	header file for Polygon <b>class</b> .
POLY_VH	illustrates the C++ input string stream <b>class</b> <code>istrstream</code> .
ISTRSTR.CPP	illustrates the C++ console stream <b>class</b> <code>constream</code> .
CONSTR.CPP	illustrates redirection.
REDIRECT.CPP	illustrates command line arguments and the passing of strings.
HELLO.CPP	illustrates command line arguments and the passing of numerics.
SQ_ROOT.CPP	illustrates command line arguments.
CPP_CPY.CPP	illustrates the <code>printf()</code> function.
PRINTF0.C	further illustrates the <code>printf()</code> function.
PRINTF1.C	further illustrates the <code>printf()</code> function.
PRINTF2.C	illustrates the <code>sprintf()</code> function.
SPRINTC.C	illustrates the <code>scanf()</code> function.
SCANF0.C	illustrates the <code>scanf()</code> function.

SCANF1.C	further illustrates the <i>scanf()</i> function.
SCANF2.C	further illustrates the <i>scanf()</i> function.
SCANF3.C	further illustrates the <i>scanf()</i> function.
SCANF4.C	further illustrates the <i>scanf()</i> function.
SCANF5.C	illustrates the <i>scanfset</i> feature of the <i>scanf()</i> function.
SCANF6.C	further illustrates the <i>scanfset</i> feature of the <i>scanf()</i> function.
SCANF7.C	further illustrates the <i>scanfset</i> feature of the <i>scanf()</i> function.
SSCANF.C	illustrates the <i>sscanf()</i> function.
O&R&D&C.C	illustrates opening, reading, displaying and closing a file using the C <i>fopen()</i> , <i>fgetc()</i> and <i>fclose()</i> file handling functions.
O&R&D&C1.C	illustrates opening, reading, displaying and closing a file using the C <i>fopen()</i> , <i>fgets()</i> and <i>fclose()</i> file handling functions.
C_CPY.C	illustrates the <i>fputc()</i> function.
C_CPY1.C	illustrates the <i>fputs()</i> function.
FR&FW.C	illustrates the <i>fread()</i> and <i>fwrite()</i> functions.
C_RND_AC.C	illustrates random file access in C using the <i>fseek()</i> , <i>ftell()</i> , <i>fgetpos()</i> and <i>fsetpos()</i> functions.
DIB.H	device independent bitmap <b>class</b> for Windows.
DIB.CPP	implementation file.
DIB_TST.CPP	a DIB displayer for Windows.
DIB_TST.RC	resource script file.
DIB_TST.RH	resource script header file.
DIB_TST.DEF	module definition file.

### Chapter 18

DEFINE.CPP	illustrates the <code>#define</code> preprocessor directive.
CPP_VER.CPP	a C++ version of DEFINE.CPP.
ERROR.CPP	illustrates the <code>#error</code> preprocessor directive.
INC.CPP	illustrates the <code>#include</code> preprocessor directive.
LINE.CPP	illustrates the <code>#line</code> preprocessor directive.
UNDEF.CPP	illustrates the <code>#undef</code> preprocessor directive.
IF.CPP	illustrates the <code>#if</code> preprocessor directive.
IFNDEF.CPP	illustrates the <code>#ifndef</code> preprocessor directive.
###.CPP	illustrates the <code>#</code> and <code>##</code> preprocessor operators.
PREDEF.CPP	illustrates the predefined macros.
ASSERT.CPP	illustrates the <code>assert()</code> macro.

### Chapter 19

NCLASH.CPP	illustrates name clashes.
N_CLASH1.CPP	illustrates name clashes.
NSPACE.CPP	illustrates the <b>namespace</b> declaration.
NS_NEST.CPP	illustrates nested namespaces.
NO_CLASH.CPP	modified N_CLASH.CPP using namespaces.
POLYGON.H	<b>class</b> Polygon.
SPHERE.H	<b>class</b> Sphere.
USE_DEC.CPP	illustrates the <b>using</b> declaration.
USE_DIR.CPP	illustrates the <b>using</b> directive.

---

GLOBAL.CPP	namespaces and global space.
OVERLOAD.CPP	overloading namespaces.
INHERIT.CPP	illustrates namespaces and inheritance.
UN_HIDE.CPP	a namespaces solution to the problem of a derived <b>class</b> member hiding a base <b>class</b> member.
MULTIPLE.CPP	namespaces and multiple inheritance.
<b>Chapter 20</b>	
TOL.H	various tolerances.
BOOL.H	enumerated type Boolean.
MIN_MAX.H	<i>Min()</i> & <i>Max()</i> <b>template</b> functions.
PT.H	<b>class</b> Point.
PT.CPP	implementation file for Point <b>class</b> .
LINE.H	<b>class</b> Line.
LINE.CPP	implementation file for <b>class</b> Line.
VEC.H	<b>template class</b> Vector.
VEC3D.H	<b>class</b> Vector3D.
VEC3D.CPP	implementation file for <b>class</b> Vector3D.
LL.H	<b>template class</b> LinkedList.
LLI.H	<b>template class</b> LinkedListIterator.
CPLX.H	<b>class</b> Complex.
CPLX.CPP	implementation file for <b>class</b> Complex.
RGB.H	classes RGBColour and NormalisedRGBColour.
RGB.CPP	implementation file for classes RGBColour and NormalisedRGBColour.
BBOX.H	<b>class</b> BoundingBox.
SHAPE.H	abstract base <b>class</b> Shape.
PLANE.H	<b>class</b> Plane.
PLANE.CPP	implementation file for <b>class</b> Plane.
POLY.H	<b>class</b> Polygon.
POLY.CPP	implementation file for <b>class</b> Polygon.
TRI.H	<b>class</b> Triangle.
TRI.CPP	implementation file for Triangle <b>class</b> .
QUAD.H	<b>class</b> Quadrilateral.
QUAD.CPP	implementation file for <b>class</b> Quadrilateral.
TET.H	<b>class</b> Tetrahedra.
TET.CPP	implementation file for <b>class</b> Tetrahedra.
SPHERE.H	<b>class</b> Sphere.
SPHERE.CPP	implementation file for <b>class</b> Sphere.
CIRC.H	<b>class</b> Circle.
CIRC.CPP	implementation file for <b>class</b> Circle.
VIEWER.H	<b>class</b> Viewer.
VIEWER.CPP	implementation file for <b>class</b> Viewer.
RECT_WIN.H	<b>template class</b> RectWindow.
VIEW_PL.H	<b>class</b> ViewPlane.
VIEW_PL.CPP	implementation file for <b>class</b> ViewPlane.
SCREEN.H	<b>class</b> Screen.
SCREEN.CPP	implementation file for <b>class</b> Screen.
SURF.H	<b>class</b> Surface.
SURF.CPP	implementation file for <b>class</b> Surface.
LIGHT.H	<b>class</b> PointLight.

LIGHT.CPP	implementation file for <b>class</b> PointLight.
WORLD.H	a World <b>class</b> .
WORLD.CPP	implementation file for <b>class</b> World.
RAYTRACE.CPP	a raytracing program for Windows.
RAYTRACE.RC	resource script file.
RAYTRACE.RH	resource script header file.
RAYTRACE.DEF	module definition file.

# Overview

*I'm an explorer, okay? I like to find out.*  
Richard Feynman (Sykes, 1994)

*This chapter reviews C++: what it is, its history and its future. Alternative object-oriented programming languages are reviewed and we examine what is actually meant by object-oriented programming and what we should expect from an object-oriented programming language.*



## 1.1 Why C++?

In the words of the original designer of C++, B. Stroustrup (1991, Preface to the first Edition): 'C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types.... When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.' Note the emphasis placed on the programmer and the program.

C++'s major features are:

- A superset of the C language.
- Stronger type checking than C.
- Support for data abstraction and object-oriented programming.
- Classes and abstract classes that encapsulate data and functions which operate on a **class**'s data, defining a given structure and behaviour.
- Inheritance and multiple inheritance, enabling the creation of hierarchies of classes.
- Support for run-time polymorphism via **virtual** functions, allowing a **class** in an arbitrary inheritance hierarchy to redefine its parent's member functions.
- Overloading. Overloaded functions and member functions allow a function with the same name but with a different number or type of arguments, or both, to be overloaded. A **virtual** function of a base **class** can be overridden by a function in a derived **class** with the same name and the same number or type of arguments, or both, and return type. Overloaded operators allow you to give new functionality to existing operators when defining a **class**.

- Support for the creation of parametrised types, generics or templates and generic functions.
- Exception handling or error handling techniques.
- Run-time type information (RTTI) to obtain run-time identification of types and expressions.
- Namespaces to solve the problem of a single global namespace or scope.

Why choose C++ rather than C? C++ is built on C and is thus a superset of C. The C++ language adds more than just object-oriented programming capabilities to C. For example, compare the C++ **new** and **delete** operators with ‘equivalent’ C user-defined macros *NEW()* and *DELETE()*. The C version is:

```
#include <stdio.h>      // printf()
#include <stdlib.h>      // malloc(), free(), exit()

#define NEW(p, ptype) \
    if ((p=ptype*) malloc (sizeof (ptype)) == NULL) \
    { \
        printf ("Out of memory\n") ; \
        exit (0) ; \
    }
#define DELETE(p) \
    if (p) \
    { \
        free ((char*)p) ; \
        p = NULL ; \
    }

//...
struct X
{
    //...
};

void main ()
{
    struct X* xptr ;
    //...
    NEW (xptr, X);
    //...
    DELETE (xptr) ;
    //...
}
```

requiring the inclusion of two header files (STUDIO.H and STDLIB.H) and four function calls *malloc()*, *printf()*, *exit()* and *free()* in the definition of *NEW()* and *DELETE()*. *NEW()* and *DELETE()* are defined as macros rather than functions because C will not permit the manipulation of a variable without strict regard to type. Thus, it would be necessary to define separate functions for each different data type used, e.g. *New\_Int(p, int)*, *New\_X(p X)*, ... . Therefore, to avoid this, macros (as opposed to functions) are used, which are expanded inline prior to compilation, which can be prone to errors.

The equivalent C++ version is:

---

```

class X
{
//...
};

//...
void main ()
{
X* xptr = new X ;
//...
delete xptr ;
//...
}

```

No header files or function calls are required and **new** incorporates an implicit memory allocation check. The **new** and **delete** operators allow a programmer to perform memory allocation/deallocation in a structured and consistent language-based manner. In addition, the **new** operator supports a *placement syntax* to allow an object to be placed at a particular location in memory rather than relying on the operating system to determine where an object is to be stored. Thus, the **new** and **delete** operators should be viewed as part of a more general and comprehensive memory management system and not simply allocation/deallocation mechanisms. This example clearly illustrates the compactness and elegance of C++. Although certain features in C are still available in C++, they are seldom used.

The object-oriented programming features of C++ will be discussed shortly, but first let us take a look at the history of C++.

## 1.2 C++'s History

For a unique insight to the history of C++ refer to *The Design and Evolution of C++* by Bjarne Stroustrup (1994)<sup>1</sup>. In addition, this book is an invaluable text for the reader who wishes to develop a greater understanding of the C++ language, and I would certainly recommend reading it after you have developed a reasonable knowledge of C++.

The origin of C++ begins with the languages ALGOL 60, CPL and BCPL. In 1970, Ken Thompson of AT&T's Bell Laboratories began developing a language called B, which was based on an existing typeless language, BCPL, originally developed by Martin Richards in the mid-1960s (Richards and Whitney-Strevens, 1979). BCPL (Basic CPL) was a derivative of the CPL programming language developed at both Cambridge and London Universities. Shortly afterwards, it became apparent that B was not suitable for implementing the Unix operating system, also originally designed and implemented by Ken Thompson. At the same time, Dennis Ritchie, also of Bell Laboratories, was developing a successor to B that was to be a compact and robust language called C. The C language was so successful that approximately 90% of the Unix operating system was rewritten in C. Today, most of Unix is written in C.

The C language quickly became very popular, particularly in the university environment, due to the availability of inexpensive compilers and to the success of Unix. In 1978, Kernighan and Ritchie published the book *The C Programming Language* (frequently referred to as the *white book*) which contained a C reference manual as an appendix. Following the publication

---

<sup>1</sup> See also the articles by Ritchie (1993) and Stroustrup (1993a) on the development of C and C++ in the ACM SIGPLAN Second History of Programming Languages Conference.

**Table 1.1** Key events in C++'s history.

1979	C with Classes development commences
1982	First paper on C with Classes (Stroustrup, 1982)
1983	C++ implementation in use; C++ named
1984	First C++ manual
1985	First commercial release (Cfront Release 1.0)
1986	<i>The C++ Programming Language</i> , 1st edn (Stroustrup, 1986)
1987	First USENIX C++ conference
1988	Zortech C++ Release
1989	Cfront Release 2.0
1990	Borland C++ Release <i>The Annotated C++ Reference Manual</i> (Ellis and Stroustrup, 1990)
1991	Cfront Release 3.0 <i>The C++ Programming Language</i> , 2nd edn (Stroustrup, 1991)
1992	DEC, Microsoft and IBM C++ Releases
1994	Draft ANSI/ISO standard

of this key book, many compilers were referred to as 'K&R compliant'. (Incidentally, the Borland C++ compiler (version 5.0) still supports K&R language compliance.) By the mid-1980s, there were more than 20 C compilers for MS-DOS, and C was in danger of being fragmented into several dialects. As a result, in 1982 an American National Standards Institute (ANSI) standardisation committee was formed to produce a standard definition of the C language. Seven years later, the ANSI X3.159-1989 standard was produced. This standard is frequently referred to as the 'ANSI C' standard.

As a consequence of languages such as Simula and Smalltalk<sup>2</sup>, in the late 1970s object-oriented programming was becoming an increasingly popular style of programming. In 1979, Bjarne Stroustrup, of AT&T's Computing Science Research Centre of Bell Laboratories in Murray Hill, New Jersey, began developing a language called *C with Classes*, which essentially added Simula classes to C via a preprocessor called Cpre. Initially, C with Classes was used exclusively by employees of AT&T. By 1982, C with Classes was sufficiently successful for it to be redesigned into a new language, which was to be called C++. Also, around this time C++ received a new compiler front-end implementation (Cfront). The first commercial release of C++ was in 1985 (Cfront Release 1.0). From 1989 onwards, the emphasis was placed by the ANSI/ISO standards committee, X3J16, on developing a C++ standard, and a draft standard was first issued in 1994. Refer to Table 1.1 for an overview of key events in C++'s history.

C++ got its name from Rick Mascitti in 1983. The Clanguage provides the `++` unary operator for incrementing a variable or pointer. For example, a common operation is incrementing a variable, say *i*, by adding the constant 1; i.e.  $i = i + 1$ , or, using the `++` operator, `i++`. Thus, C++ indicates the 'next' C.

## 1.3 C++'s Future

It has been stated above and throughout the literature that C++ is a superset of C. As C++ continues to develop this is gradually becoming less and less the case. C++ is becoming a separate language, gradually abandoning its complete compatibility with C. C++ is a continually evolving language, a *living language*. Fortunately, it is evolving to meet the needs of users.

---

2 For a fascinating discussion of the early history of Smalltalk refer to Kay (1993).

## 1.4 There are Other OOP Languages

C++ is not the only programming language that supports object-oriented programming features. For instance, two languages that support OOP which have greatly influenced the development of C++ are Simula and Smalltalk. Three good works covering Simula and Smalltalk are Kirkerud (1989), Goldberg and Robson (1983) and Pinson and Weiner (1988).

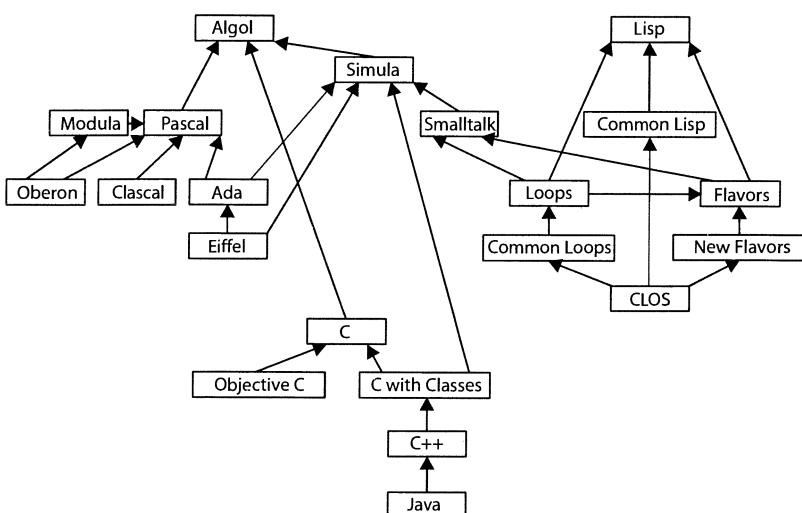
Other object-oriented programming languages are Common Lisp Object System (CLOS) (Bobrow *et al.*, 1988), Eiffel (Meyer, 1991), Ada (Ada Reference Manual, 1983) and Oberon-2 (Reiser and Wirth, 1992; Mössenböck, 1993). Figure 1.1 illustrates the development of several of the more well-known object-oriented programming languages to date. For the interested reader, Saunders (1989) provides a survey of more than 80 object-oriented programming languages.

Although not the first object-oriented language, Smalltalk was instrumental in the development of object-oriented programming by demonstrating that object-oriented programming is a feasible solution to software development. Smalltalk's strengths are its excellent user interface and class library. Smalltalk is a dynamically typed language, and as a result suffers from static typing. For an introduction to Smalltalk, refer to, for example, Budd (1987).

Oberon-2 evolved from Oberon, which in turn evolved from Pascal and Modula-2. In Oberon-2, classes are represented as records which encapsulate both data and procedures, rather than by introducing an additional class construct. Oberon-2 integrates with the Oberon operating system, which provides run-time support for Oberon-2 programs. Refer to Mössenböck (1993) for details of how to obtain the Oberon-2 compiler and Oberon system.

Eiffel, although not as well-known as C++, is nevertheless a very powerful and industrially popular language and is more of a *pure* object-oriented language than C++, having not evolved primarily from a non-object-oriented language. Even though Eiffel is an object-oriented programming language, it does not forsake the ability to generate efficient program code such as that generated by C. Eiffel is a strong statically typed language and is particularly strong with regard to inheritance.

New object-oriented programming languages are continually emerging. One of the more popular ones to recently emerge is Java, developed by Sun Microsystems and derived from C++. Java is a simple, compact implementation of C++, but with several C++ features removed and



**Fig. 1.1** Evolution of several object-oriented programming languages.

with a few important additional features. Some of the major features removed are pointers, templates, multiple inheritance and operator overloading. Java also eliminates the preprocessor and header files. Multithreading is integral in the Java language, so that applications can be developed with multithreads; this is an important feature when developing applications for the Internet, where Java has been particularly successful. A Java language compiler generates architecture-neutral object files by generating bytecode instructions which are architecture-independent. This means that a single Java application can be developed which will execute on several operating systems. Java also supports automatic memory management. For further details of the Java programming language refer to Anuff (1996).

The language you choose (if you have this choice) to work with is ultimately up to you. However, I would like to add that C++ is a relatively new language and still very much growing. Personally, one of the most exciting things about the language is watching the language grow and mature. That's not to say that because the language is young it has the quirky characteristics of early versions of software. Remember that C++ has been built on C, with numerous object-oriented programming features *borrowed* from existing languages.

## 1.5 Programming Paradigms

There are a number of different programming styles that programmers generally adopt. These styles are mainly a result of the programming language used, just as different people trained in different disciplines, such as mathematics, physics, computer science or engineering, will tackle a given problem completely differently according to their own 'subject language', with which they are familiar or feel most comfortable. There are five main categories of programming style (Booch, 1994):

- procedure-oriented: algorithms
- object-oriented: classes and objects
- logic-oriented: goals
- rule-oriented: if-then rules
- constraint-oriented: invariant relationships

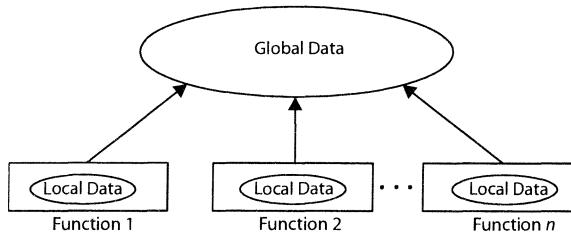
The following sections examine the current two most popular styles of programming adopted, namely procedural and data abstraction/object-oriented.

## 1.6 Procedural and Modular Programming

Examples of procedural (sometimes referred to as structured) programming languages are COBOL, Fortran 77/90, Pascal and C. An example of procedural programming in C is:

```
#include <stdio.h>
/* get/print your name */

void main ()
{
    char name[21] ;
    printf ("enter your name: ") ;
```



**Fig. 1.2** Procedural approach.

```

gets (name) ;
printf ("\nhello, %s", name) ;
}

```

which simply gets a person's name and displays a message based on the name entered. A procedural program is a sequential list of statements or instructions, focusing on processing. Provided the program size remains fairly small, this approach is bearable. However, for large programs a list of statements becomes incomprehensible. Thus the use of functions came about. Each function performs a specified operation, is clearly defined and interfaces with other functions and the program; see Fig. 1.2. In the above example, `gets()` and `printf()` are functions, and each performs a different input/output operation. Consider the following hypothetical header and implementation files, which declare and define several related mathematical functions.

```

/* my_math.h, declarations of floating point math routines */
//...
double cos (double x) ;
double sin (double x) ;
double sqrt (double x) ;
//...

/* my_math.c, implementation of floating point math routines */
#include "my_math.h"
//...
double cos (double x)
{
    /* code for calculating the cosine */
}
//...

```

This grouping together of a number of related functions is often referred to as a *module*. It is worth mentioning that the concept of a module enables data, variables and functions to be hidden within the module.

A typical use of the module could be:

```

#include <stdio.h>
#include <stdlib.h>
#include <my_math.h>
/* print the square root of your age */

```

```

void main ()
{
    char buffer[9] ;
    double sqrt_age ;
    printf ("enter your age: ") ;
    sqrt_age = sqrt (atoi (gets (buffer)) ) ;
    printf ("\nsquare root of your age: %f", sqrt_age) ;
}

```

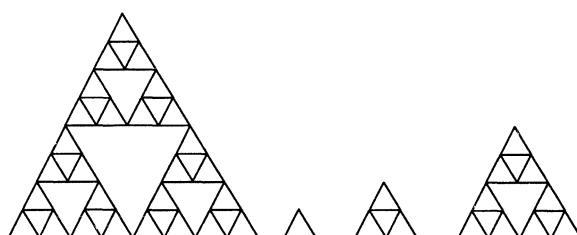
In general, functions perform operations on supplied data. The emphasis is placed on doing things: do this, do that, did it work? Thus, the idea of functional abstraction enhances an algorithmic problem-solving design approach rather than a feature-based abstraction approach.

## 1.7 Data Abstraction

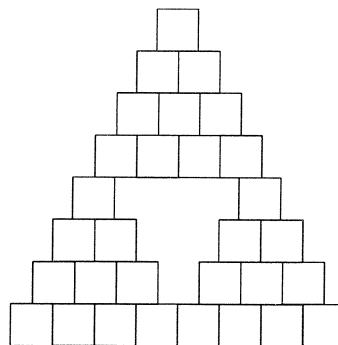
Data abstraction is the programming technique of being able to create new data types. The word *abstract* is used to reflect how a programmer abstracts features and concepts from a complex system into new data types. A programmer-defined data type is often referred to as an ADT (abstract data type) to distinguish it from built-in FDTs (fundamental data types), such as characters and integers. Apart from making a programmer think in terms of features, design and program organisation, data abstraction also makes programs more flexible, shorter and conceptually easier to understand. ADT's are implemented in C++ by declaring *classes*. Classes allow both data and functions to be associated with the **class** name. A **class** is similar to the C **struct** and the **RECORD** (combined with procedures) of Pascal and Oberon-2.

As an example of data abstraction, consider a complex system such as the Sierpinski gasket (also known as Sierpinski's sieve or carpet, after Vaclav Sierpinski) shown in Fig. 1.3. We could model the gasket solely in terms of integer and floating fundamental data types. Alternatively, we take the data abstraction route by observing that the gasket is composed purely of triangles placed in some kind of self-replicating fractal order. Similarly, each triangle can be decomposed into vertices or points, edges and a face. A face is composed of three edges, an edge consists of two end-points and a point is three floating-point coordinates.

An advantage of the abstract data approach is that our point, edge and face data types can now be applied to alternative geometrical constructions, such as the similar construction shown in Fig. 1.4, which comprises squares rather than triangles. Thus, data abstraction has allowed us to generate generic types that possess common features that can be realised and utilised by a whole class of problems. It is worth pointing out that all abstract data types, however complex and abstract, can be degenerated into fundamental data types.



**Fig. 1.3** Sierpinski's gasket and the first three stages of construction.



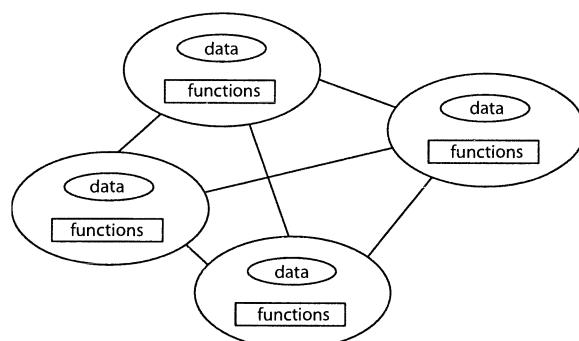
**Fig. 1.4** Alternative Sierpinski gasket.

Data abstraction forms the foundation of object-oriented programming, and as a result a more detailed discussion of data abstraction is postponed till the next section.

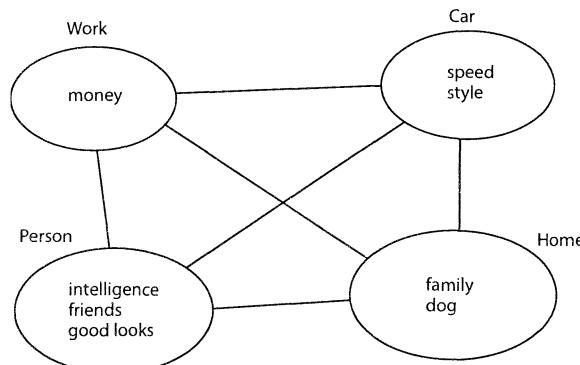
## 1.8 Object-Oriented Programming

There are many different and diverse approaches to program development and object-oriented programming is just one of them. Object-oriented programming is not a new concept. The key ideas of object-oriented programming date back to as early as 1967, from the Simula programming language, and the early 1970s, from the Smalltalk language. Simula was developed by Dahl, Myrhaug and Nygaard (1970), whereas Smalltalk was developed at Xerox Palo Alto Research Center. Simula was based on the Algol language, with the addition of encapsulation and inheritance. C++ is an object-oriented programming language and has inherited many of the object-oriented features of its predecessors.

So what actually is object-oriented programming and how do we put this programming approach in to practice in C++? By way of a typical example we shall now examine object-oriented programming in C++. An object-oriented program consists of a number of different objects, as shown in Fig. 1.5. Objects are literally everywhere, including ourselves: see Fig. 1.6. Each object ‘carries’ its own set of characteristics or data and is continually sending messages to and receiving messages from other objects.



**Fig. 1.5** Object approach. Objects talk to each other.

**Fig. 1.6** World objects.

Initially, the best way to think of the object-oriented paradigm is via ‘real’ objects: everyday objects around us. Therefore, consider the development of a program to characterise a variety of different shapes. At this point you will probably be unfamiliar with the C++ implementation details, so try to concentrate on the object-oriented approach. At a later stage you may want to refer back to this chapter in order to fully understand the ideas presented.

If you are familiar with the procedural way of thinking about program construction it is tempting at this stage to be focusing your attention more on the details and algorithmics of generating individual shapes, the division of such details into functions, and the flow of information through a shapes program into a series of steps. Let us take an alternative approach of decomposing the problem into key abstractions which embody their own unique structure and behaviour, each abstraction modelling an object in the real world.

What general properties do *all* objects around us have? They have colour, size, position relative to a given space, orientation and possibly taste, smell and so on. What kinds of object are there, or more importantly which of the infinite number of objects in the real world do we want to model? Let us limit our discussion to, say, a rectangle, a circle and a hexahedron (a three-dimensional object composed of quadrilateral faces, such as a cube). What specific properties do objects have? A circle has a centre, radius, line thickness etc. These features are *our* abstractions, or the essential characteristics of objects as we perceive them to be. Of course<sup>3</sup>, there is no right or wrong set of abstractions. Different sets of people choose different sets of abstractions. However, if there is a general rule – and there isn’t – try to choose abstractions that are as general as possible so that they do not become too specialised and known only to a small group of people.

A **class**, FuzzyShape, that defines the general properties of all shapes is:

```

class FuzzyShape
{
public:
    Colour   colour ;
    Size     size ;
    Position position ;
    //...
    FuzzyShape () ;
    //...
  
```

<sup>3</sup> Bob Reuben

---

```
};
```

assuming that we already have classes which define position, colour and size:

```
class Position
{
public:
    double x, y, z;           // (x, y, z) coordinates
                            // in space
    Position () ;
    //...
};

class Colour
{
public:
    int red, green, blue;   // 3 primary components
                            // of colour
    //...
};

class Size
{
//...
};
```

The FuzzyShape **class** captures our abstraction of the characteristic features of all shapes. The **class** is called FuzzyShape rather than Shape to emphasise that the **class** does not give a clear or *crisp* definition of any particular shape.

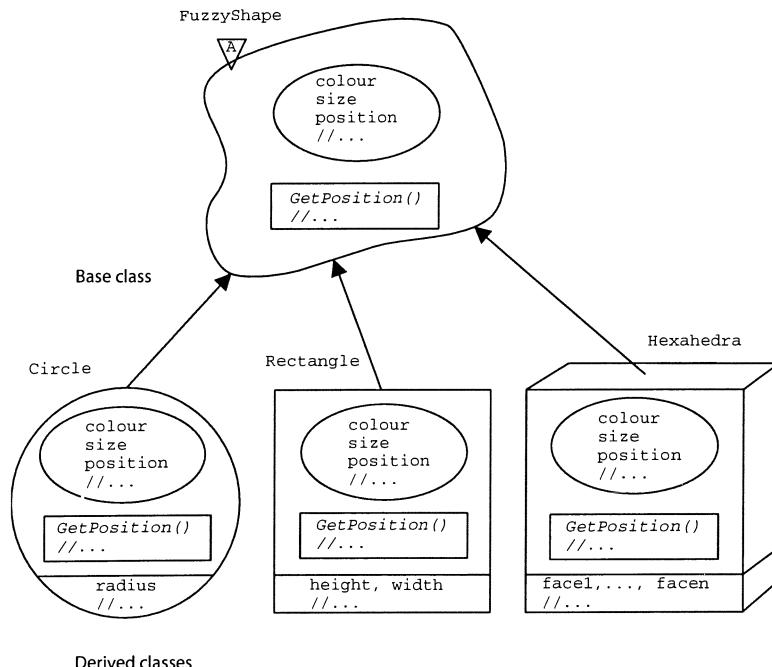
To create an object or an instance of FuzzyShape so that we can do things to the object we typically write:

```
FuzzyShape fuzzy_object ;
//...
cout << "fuzzy object's position: " << fuzzy_object.position ;
cout << "fuzzy object's colour: " << fuzzy_object.colour ;
//...
```

just as you might define a variable var of type **int**:

```
int var ;
//...
var = 7 ;
cout << "value of var: " << var ;
//...
```

At present, our FuzzyShape **class** is nothing more than an elaborate list of properties which are difficult to operate on and are of limited use. Also, you may have noticed that the colour, size and position data members were declared with **public** access, which allows access to all clients. Generally, we are interested in separating an object's data from its client interface by *data-hiding*. Data hiding allows an object complete control in specifying



**Fig. 1.7** Inheritance of shapes. The inverted triangle encompassing the uppercase letter A indicates an abstract base class in accordance with the Booch (1994) notation.

what data a client needs to have access to and what data is, or should be, of no interest to a client. Hiding the internal details of an object can greatly reduce the possibility of accidental object-data corruption.

In the above example we were able to create an object of **FuzzyShape class** when such an object is meaningless in the real world, since **FuzzyShape** models general properties of shapes but is not an abstraction of any real shape. The inheritance mechanism of C++ enables us to define **FuzzyShape** as a base **class** and shape classes such as **Rectangle**, **Circle** and **Hexahedra** to be *derived* from **class FuzzyShape**; see Fig. 1.7<sup>4</sup>. In fact, **FuzzyShape** is made an *abstract* base **class**, since we don't want to define any objects of this **class**. In other words **FuzzyShape** acts only as a base **class** for other classes. Also, in the following modified example the data members of **FuzzyShape** are now **protected** to enforce data-hiding and restrict access to the **class** itself, derived classes and **friends**:

```

class FuzzyShape
{
protected:
    Colour colour ;
    Size size ;
    Position position ;
    //...

```

<sup>4</sup> Where possible, the Booch (1994) notation has been adopted throughout this book for diagrammatically illustrating classes, objects and their interactions.

```
public:  
    FuzzyShape () ;  
    // member functions  
    Position GetPosition () const ;  
    void SetPosition (const Position& p) ;  
    //...  
    // pure virtual member functions  
    virtual void Draw () = 0 ;  
    //...  
};  
  
class Rectangle : public FuzzyShape  
{  
protected:  
    double height, width ;  
public:  
    Rectangle () ;  
    void Draw () ;  
    //...  
};  
  
class Circle : public FuzzyShape  
{  
protected:  
    double radius ;  
public:  
    Circle () ;  
    //...  
};  
  
class Hexahedra : public FuzzyShape  
{  
public:  
    Hexahedra () ;  
    //...  
};
```

The implementation file might be:

```
FuzzyShape::FuzzyShape ()  
{  
    position = 0.0 ;  
    //..  
}  
  
Position FuzzyShape::GetPosition ()  
{  
    return position ;  
}  
  
void FuzzyShape::SetPosition (const Position& p)
```

```
{  
position = p ;  
}  
//...
```

A typical use of the above implementation could be:

```
// error: can't create an instance of an abstract class  
FuzzyShape fuzzy_object ;  
  
Rectangle rectangle_object ;  
Circle    circle_object ;  
//...  
// error: not accessible  
cout << "circle position: " << circle_object.position ;  
  
// O.K., access thro' access member function  
cout << "circle position: " << circle_object.GetPosition () ;  
//...  
  
FuzzyShape* fs_array[N] ;           // array of pointers to  
                                  // FuzzyShapes  
//...  
fs_array[0] = &circle_object ;      // place addresses in  
                                  // pointer array  
fs_array[1] = &rectangle_object ;  
//...  
// draw all shapes using a single function call!  
for (int i=0; i<N; i++)  
    fs_array[i]->Draw () ;  
//...
```

noting that `position` now has restricted access.

The code above illustrates that after an array of pointers to `FuzzyShape` is created and the object addresses are assigned to the pointer array we are then able to draw a variety of shapes using a single function call. This example of calling completely different functions with a single function call is an example of *polymorphism*, made possible with the aid of inheritance and **virtual** functions.

It should be clear from the discussion above that object-oriented programming takes a great deal more planning than traditional procedural or modular programming, but is a more natural way of solving problems.

To be considered an object-oriented programming language it is often stated that a language must support the following key elements: *abstraction*, *encapsulation*, *modularity*, *inheritance* and *polymorphism*; see, for example, Meyer's ten key OO concepts (Meyer, 1995). A brief description of each follows.

### 1.8.1 Abstraction

Abstraction is concerned with formulating the *essential* characteristics of an object. In the above example we characterised a circle purely in terms of a radius as well as the inherited characteristics of colour, size and position. Real objects have an infinite number of features,

---

and it is simply not possible to model and comprehend all of them and the complex interactions between different objects. Alternatively, a complex object is broken down into a finite set of simplified, yet essential, characteristics. Because of the infinite number of object characteristics it is important to focus on the essential characteristics when formulating an object abstraction. Formulating an abstraction is generally not as straightforward a process as it might at first appear. Different people inherently view the same object differently. What is required is a set of characteristics which are considered essential by the majority of users for a given problem domain.

### 1.8.2 Encapsulation

While abstraction addresses the characteristics of an object from an external viewer's perspective, encapsulation focuses on the implementation of an object. The implementation encapsulates the details about an abstraction into separate elements. The implementation should be considered a secret of the abstraction and hidden from most clients. For instance, a car is composed of literally thousands of different components, but can be viewed as an engine, chassis, body, controls etc., with the specific details concerning the implementation and operation of these different abstractions hidden from the driver. The driver is supplied with *just enough* information about the different abstractions to be able to drive the car.

Encapsulation is another word for the *black box* paradigm. Given an input, the black box generates a respective output – this is all the information we know about the box. The black box can perform any number of other operations, but we are given no clues as to the nature of the internal details of the black box.

Note that the abstraction of an object should precede its implementation. The abstraction of an object should not comprise any implementation details.

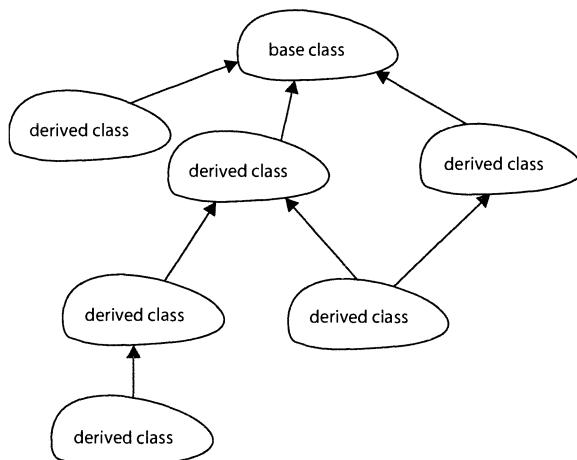
### 1.8.3 Modularity

Modularity is simply the decomposition of an abstraction into separate, discrete cells. It is an intuitive process to break a large complex system problem down into a manageable number of components. When programming in the large, organising a program into separate coherent modules is essential to the design of an application. Individual modules can be tested reasonably independently of the overall application. Modularity helps in designing an application to be both extendable and reusable.

### 1.8.4 Inheritance

Inheritance is concerned with placing separate but related abstractions into a structured hierarchy which will enable the passing of shared characteristics. Figure 1.8 illustrates a schematic **class** hierarchy in C++. A *derived class* inherits properties of a *base class*. Figure 1.8 also illustrates the concept of multiple inheritance, in which a **class** is derived from more than one base **class**. Multiple inheritance is plausible when you remember that we all have two parents.

The most important property of inheritance from a programming perspective is that it allows the reuse of program code, and consequently reduces the repetition of code.



**Fig. 1.8** Class hierarchy illustrating single and multiple inheritance.

### 1.8.5 Polymorphism

Polymorphism is the ability of a child object in an inheritance hierarchy to exhibit different behaviour based on the type of the object. This feature enables an object to respond to a common set of operations in different ways.

In addition to the above five key elements, there are some minor elements of object-oriented programming which warrant mentioning: *typing*, *automatic memory management*, *concurrency* and *persistence* and *operator overloading*.

### 1.8.6 Typing

The distinction between type and **class** is very confusing, and we will assume that the two terms are equivalent. It suffices to say that *a class implements a type*.

Programming languages are referred to as untyped, weakly typed or strongly typed. A strongly typed language requires that a strict conformance of type is enforced at all times. Operations cannot be performed on an object unless the object's **class** possesses an exact signature of the operation or member function. For strongly typed languages such problems can be dealt with during compilation. If violations of type are detected at compilation then a language is referred to as *strongly statically typed*. An untyped language, such as Smalltalk, allows an object to send any message to the object's **class** even if the **class** does not know how to respond to the message. Such errors generally go unnoticed until program execution. If violations of type are allowed to pass compilation and type checking is postponed until run-time then a language is referred to as *dynamically typed*.

Regarding classes, C++ is strongly typed. Consider the shapes program (introduced above) once more:

```

class FuzzyShape
{
// ...
};
  
```

---

```

class Rectangle : public FuzzyShape
{
protected:
    double height, width ;
    //...
public:
    //...
    double Area () ;           // compute rectangle's area
    //...
};

class Hexahedra : public FuzzyShape
{
protected:
    Rectangle* face_array ;
    //...
public:
    //...
    double SurfaceArea () ;   // compute surface area
                               // of solid
    //...
};

```

The following assignment is illegal in C++:

```

Hexahedra hex ;
double     area ;
//...
area = hex.Area () ;

```

because the member function `Area()` is not defined for the **class** or any base classes of object `hex`. However, the following assignments are legal:

```

Rectangle rect ;
Hexahedra hex ;
double     r_area, h_area ;
Position    h_pos ;
//...
r_area = rect.Area () ;
h_area = hex.SurfaceArea () ;
h_pos  = hex.GetPosition () ;

```

The more static typing performed and the more bugs found and fixed at the time of compilation the better. However, static typing alone is too constricting for object-oriented programming. The ideal solution to typing is a mixture of static typing, dynamic binding and polymorphism, which are so essential to object-oriented programming. C++ supports both strong static typing and dynamic binding, but in a controlled manner. By categorising similar abstractions into class hierarchies with a common base class and using polymorphism, a base class object is allowed to point to objects of classes within its own class hierarchy, but is excluded from pointing to objects outside of its respective hierarchy.

### 1.8.7 Automatic Memory Management

A large object-oriented application will create and destroy numerous objects. The manual task of allocating and freeing memory is a difficult task for a programmer involved in a large object-oriented application. Automatic memory management mechanisms track down memory used by objects that are no longer accessible to an application. Automatic memory management is an important support feature for object-oriented programming, which not only makes a programmer's life easier but can also reduce programmer errors associated with manual memory management.

C++ does not support automatic memory management, although optional memory management is being considered (Stroustrup, 1994). Several other object-oriented programming languages, such as Oberon-2 and Java (which is a derivative of C++), do support automatic memory management.

### 1.8.8 Concurrency

Concurrency allows objects to operate simultaneously. The Microsoft Windows 3.1 non-preemptive multi-tasking operating system is a good example of different objects being *alive* at the same time and interacting with one another.

### 1.8.9 Persistence

Objects in a program require a certain amount of space in memory. Objects also exist for a certain length of time. For example, the lifetime of temporary objects in C++ can be very short, whereas certain objects can outlive the execution lifetime of a program.

An object is referred to as persistent if it outlives its creator or program execution and/or an object survives a move from its original memory address.

### 1.8.10 Operator Overloading

Although C++ is frequently referred to as an object-oriented programming language because of its support for classes, abstraction, encapsulation, inheritance and polymorphism, an equally important feature of C++ is operator overloading. Having created classes and abstract data types, C++ allows us to overload operators to manipulate objects just as if they were objects of the fundamental data types. Operator overloading is a particularly attractive feature when performing operations on numeric objects. For example, consider objects m1, m2 and m3 of a Matrix **class** which models mathematical matrices. If we are required to add the two Matrix objects m1 and m2 and assign the result to object m3, then using **class** member functions we could write:

```
//...
Matrix m1, m2, m3 ;
//...
m3 = Add (m1, m2) ;
```

However, operator overloading allows us to overload the addition operator (+) specifically for objects of the Matrix **class**. Then it is possible to write our Matrix object addition in a more natural form:

```
//...
```

---

```
Matrix m1, m2, m3 ;
//...
m3 = m1 + m2 ;
```

## 1.9 Objects

It can be seen from the discussions in the previous sections that an object is a single entity with a well-defined structure and behaviour defined in the object's **class**. An object does things and we do things to an object by *sending messages*, via member functions. Thus an object has a state and an identity which can change with the lifetime of the object. You should be aware that an object is also referred to as an *instance*, and member functions are called *methods* or *operations* in other object-oriented programming languages.

Thinking in terms of messages, consider again an object, *c*, of the geometric **class** *Circle*, which describes a circle. To determine the area of *c* we send an area message (in the form of a call to the *Circle*::*Area* () member function) to object *c*:

```
Circle c ;
//...
double area = c.Area () ;      // compute area of object c
```

The object itself processes the area message. If we send the same area message to other objects, such as *Rectangle* and *Triangle*, then the *message-object-process* is identical and independent of the type of object. Each object processes the area message appropriately. This is contrary to procedural programming, which would place emphasis on the area procedure rather than the objects.

## 1.10 Syntax, Semantics and Pragmatics

Frequently in programming we hear the words *syntax*, *semantics* and *pragmatics*. The following three sections describe what these buzzwords mean.

### 1.10.1 Syntax

The syntax of a programming language consists of the rules for the correct use of the language. These involve the correct grammatical construction and arrangement of the language, correct spelling, hyphenation, inflection and so on. Although the syntactical rules of a programming language can be difficult at first, they are far simpler than the syntactical rules of a natural language. The syntax of a programming language has to be strictly adhered to.

### 1.10.2 Semantics

The semantics of a programming language deal with the meanings given to syntactically correct constructs of the language. Semantics also deal with the relationships between symbols and the ideas given to the symbols.

### 1.10.3 Pragmatics

The pragmatics of a programming language deal with the practical application of the language, not the theory. Computer programs are designed and written to solve problems, and their ultimate success depends on how they solve the problem to which they were designed. Large, complex programs are continually changing according to the changing demands placed upon them. Thus, pragmatics also deals with the evolution of a program.

## 1.11 Applications of C++

Although my suggestions of typical applications of C++ are biased and somewhat limited, here are a few books which illustrate the diverse fields to which C++ is now applied.

- Masters, T. (1993) *Practical Neural Network Recipes in C++*, Academic Press, New York.  
Porter, A. (1993) *C++ Programming for Windows*, Osborne-McGraw-Hill, New York.  
Rao, V.B. and Rao, H.V. (1993) *C++ Neural Networks and Fuzzy Logic*, MIS Press, New York.  
Stevens, R.T. (1992) *Fractal Programming and Ray Tracing with C++*, Prentice-Hall, Englewood Cliffs NJ.  
Wilt, N. (1994) *Object-Oriented Ray Tracing in C++*, John Wiley & Sons, New York.

## 1.12 References for C++

If you are interested in referring to alternative or additional published work on C++, then a glance through the references at the end of the book should supply you with more than enough material to get going. Of the works referenced, my favourites are:

Lafore, R. (1991) *Object-Oriented Programming in Turbo C++*, Waite Group Press, Mill Valley CA.

You would be hard pushed to find a better introductory text to C++ than this, which is now in its second edition. Robert Lafore has an excellent style of writing and presentation, typical of authors for the Waite Group Press.

Adams, J., Leestma, S. and Nyhoff, N. (1995) *C++: An Introduction to Computing*, Prentice-Hall, Englewood Cliffs NJ.

This book simultaneously introduces the reader to computing and C++. This book is an excellent text for student learning and well supported with numerous examples and programming tips.

Schildt, H. (1994) *C++ from the Ground Up* and Schildt, H. (1995) *C++: The Complete Reference*, both published by Osborne-McGraw-Hill, New York.

A list of recommended reading on C++ would not be complete without a couple of references to Schildt's many C and C++ programming books. These two, of his more recent books, are excellent first books to learn C++. When it comes to writing C++ books, Schildt is simply a magician!

---

Lippman, S.B. (1991) *C++ Primer*, Addison-Wesley, Reading MA.

For a more in depth study of C++, this book is a must.

Barton, J.J. and Nackman, L.R. (1994) *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, Reading MA.

For those of you interested in the scientific and engineering applications of C++, this well-written book is essential reading. It helps scientific and engineering programmers in the transition from Fortran and C to C++, covering a wide range of applications of C++.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, New York.

Although not specifically a text on C++ programming, this book provides an invaluable text on object-oriented programming, design and analysis. All example code in the book is implemented in C++.

If you require a standard on the C++ language:

Ellis, M.A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading MA.

The first reference manual on C++ including expected future extensions of the language.

Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edn, Addison-Wesley, Reading MA.

Although not primarily a reference manual, one is included.

The latest working paper for the ANSI/ISO C++ draft standard (Standard, 1996).

If your interests are more up to date then some key journals that feature C++ are:

*The C++ Report*

*The C++ Journal*

*The C/C++ Users' Journal*

*The Journal of Object-Oriented Programming* (JOOP)

*Object-Oriented Systems*

*Computer Language*

USENIX (Unix Users' association) conference proceedings: the first C++ Workshop conference in November 1987 marked the start of a series of conferences held by USENIX in October 1988, April 1990, April 1991, August 1992 and July 1993. Refer to USENIX in the references section for further details. Also refer to the proceedings of the OOPSLA, ECOOP and C++ At Work conferences.

C++ newsgroups: `comp.lang.c++`, `comp.std.c++` and `comp.object` on the Internet and `c.plus.plus` on BIX.

## 1.13 References for Graphics

Mortenson, M.E. (1989) *Computer Graphics: An Introduction to the Mathematics and Geometry*, Industrial Press.

This is a good first text on computer graphics. The book covers points, lines, planes, curves, surfaces, projections, coordinate systems and transformations.

I also recommend the works of Glaeser (1994), O'Rourke (1994), Preparata and Shamos (1985), Stevens (1994), Watt and Watt (1992) and Wilt (1994). More recently, Laszlo (1996) has addressed the topics of computational geometry and computer graphics in C++.

## 1.14 Notation

The notational convention used throughout this book is as follows:

Courier	User-defined identifiers, objects, classes, etc.
<i>Courier Italic</i>	User comments; C/C++ and user-defined macro and function names
<b>Courier Bold</b>	C++ keywords
CAPITALS	Disk directories and filenames; user-defined constants
File   Save	Save command from the File menu
.CPP,.H	Filename extensions for C++ implementation and header files respectively
// . . .	More program code considered not essential to the present discussion.

## 1.15 Summary

C++ is a superset of the C language. C++ not only adds object-oriented programming capabilities to C, but also adds numerous other features which make programming safer, easier and more enjoyable. Object-oriented programming helps focus a programmer's attention more on the design and organisation of a program than on the details. The fundamental concept of object-oriented programming is the object. Objects possess their own properties and operations – *objects do things*.

## Exercises

- 1.1 What do you understand by object-oriented programming? Highlight the key features which define a programming language to be object-oriented.
- 1.2 What is meant by an *object*?
- 1.3 If you do not already possess or have access to the ANSI/ISO C++ draft standard, then try getting hold of a copy – it's an impressive document! For further details of how to obtain the C++ draft standard refer to Standard (1996).
- 1.4 What are the main differences between abstraction, encapsulation and inheritance?

- 
- 1.5 Obtain a copy of and read the following paper: Parnas, D.L. (1972) On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, 15(12), 1059–62. This paper was instrumental in introducing the concept of separating a program's implementation from its interface which is at the heart of data-hiding and encapsulation.
  - 1.6 What are the essential abstractions of a triangle and a polygon? Identify those abstractions that are common to both objects.
  - 1.7 Try to find out more about other object-oriented programming languages. Is C++ the best object-oriented programming language? What are the advantages and disadvantages of C++ compared with other object and non-object-oriented programming languages?

# The Development Environment

*Your working environment is very important. If you are to do a lot of programming, optimise your program development and enjoy your programming as much as possible, then you naturally need the best environment. This chapter attempts to highlight a few key requirements and tools that you should look for in your environment by providing an overview of the Borland C++ for Windows compiler and development suite.*



## 2.1 Hardware, Software and Setup Used

All of the program code presented throughout this book was written and prepared with the help of the following hardware and software.

### 2.1.1 Hardware

An IBM-compatible desktop PC with an Intel Pentium 90 MHz processor, 16 Mbyte RAM, 1 Mbyte video RAM, 1 Gbyte hard disk and an SVGA monitor was used exclusively for all development. This specification was found to be an excellent platform for all of the programs presented in this book, with quick compile and link times.

### 2.1.2 Software

The compiler used was Borland's Borland C++ for Windows, version 5.0 (1996). Why Borland? Basically, because I have used the Borland C++ compiler since 1990, when it was (and still is, in my opinion), the leader in C++ compilers for the PC. Incidentally, Borland first released their C++ compiler in May 1990. Although it is not for me to state which is the best C++ compiler on the market for your particular applications, you should, if you can, simply choose the best compiler and environment to suit your needs and style.

No keywords, libraries, container classes, ObjectWindows Library (OWL) classes or features specific to the Borland compiler have been used in this book and all programs provided on the

supplied disk should (in theory) be portable to any preferred draft ANSI/ISO C++-compatible compiler.

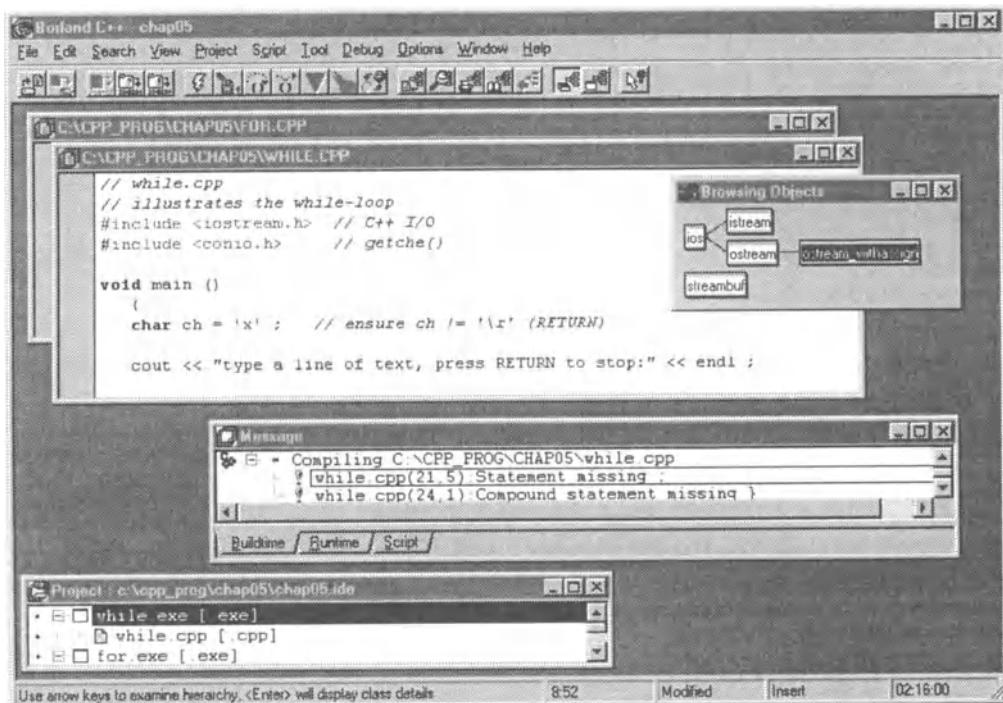
## 2.2 The Borland Integrated Development Environment

Commercial compilers such as Borland C++, Microsoft Visual C++ and Symantec C++ for Windows all support excellent development environments. This section presents a brief overview of the Borland C++ compiler and development tools.

The Borland Integrated Development Suite is an excellent environment to work in (Fig. 2.1). Apart from the Compiler, Linker and Librarian, it includes such Windows tools as a customisable Editor, SpeedBar, Browser, Debugger, WinSight and WinSpector, AppExpert, ClassExpert, Project Manager, and Resource Workshop. Command-line tools such as MAKE, TLINK and TLIB are also available. Borland C++ also fully implements the ANSI C standard with numerous additions. The list of tools and features is endless. Let us take a brief look at a few of the key features offered by the Borland C++ Integrated Development Environment (IDE).

### 2.2.1 Editor

Figure 2.1 illustrates a typical editing session. Since the client area of the IDE is a Multiple Document Interface (MDI), we can have multiple files open simultaneously. The editor is completely customisable.



**Fig. 2.1** The Borland IDE.

## 2.2.2 Text Highlighting

Syntax highlighting is a very useful facility. It allows the different elements of your code to have associated colour and font attributes. Keywords are, by default, shown in bold type, identifiers in plain type, comments in purple, strings in blue and so on. For me, syntax highlighting is not simply a nifty gimmick but greatly enhances my understanding of the code and helps to eliminate silly syntactical errors.

Syntax highlighting works by default on files with the extensions .CPP, .C, .H and .HPP. Additional file extensions can be added to the list.

## 2.2.3 Project Manager

The Project Manager manages applications that are built from many and varied components each requiring a different compilation. Figure 2.1 illustrates an example of the Project Manager window. The project tree shows the dependencies of each *node*. A project node represents the entire project, with all files required to build the project appearing under it. A target node (such as .EXE or .DLL) represents a file that is created when its dependants are built. A node (e.g. .CPP) represents a file used to build a target. A run-time node (e.g. .OBJ and .LIB) is a file used at run-time. The Project Manager relieves the programmer of having to generate a make file (.MAK). Project files (.IDE) can be converted to make files by the IDE with the 'Project | Generate makefile' option. Project Manager has an *autodependency* check which automatically translates all out-of-date nodes (.OBJ, .LIB) in the project. If a node is up-to-date then it is not re-translated.

The Borland IDE can be configured by the *ObjectScripting* feature, which uses the cScript language to make the IDE fully programmable.

## 2.2.4 Messages

Error and warning messages are displayed in a Message window when a program is compiled and linked. When a message is selected from the Message window the Editor automatically transfers to the Edit window and places the cursor on the line and column within your program code where the warning or error is most likely to have occurred. This is a useful feature that greatly speeds up the debugging phase of program development; see Fig. 2.1.

## 2.2.5 Browser

The Browser allows you to view graphically the **class** hierarchy, **virtual** and member functions, variables, constants, types, classes and objects of a program. Figure 2.1 illustrates a browsing window. The *Browsing Objects* window shows the **class** hierarchy of the *ios* C++ standard library **class**.

## 2.2.6 AppExpert

AppExpert lets you generate a program shell for Windows. The program is automatically generated and includes such features as MDI windows, a menu, a status line, a button bar, drag and drop support, on-line help and printing support. AppExpert offers a visual approach to Windows program development and interfaces with the Resource Workshop, ObjectWindows **class** library and the Project Manager. AppExpert also caters for Java-specific applications.

## 2.2.7 ClassExpert

The Borland ClassExpert assists the development of AppExpert programs by providing three window panes: *Edit*, *Events* and *Classes*. The Edit pane lets you edit your program code. The Classes pane lists the classes that ClassExpert manages. The Events pane lists Command-Notifications (e.g. CM\_EDITCUT), Windows-Messages (e.g. WM\_PAINT) and **virtual** functions of the selected **class**'s base classes. ClassExpert interfaces with the Resource Workshop to enable you to associate resources to classes.

## 2.2.8 Debugger

No matter how good a programmer you are the programs that you develop will always have their fair share of bugs in the early stages of development. The Borland Integrated Debugger assists you in locating bugs in your program which prevent it from executing correctly. The Borland CodeGuard 32/16 assists programmers in detecting program bugs at run-time for both 16- and 32-bit Windows applications. CodeGuard 32/16 validates both pointers and pointer arithmetic at the point of execution and is a useful feature for detecting memory and resource leaks.

## 2.2.9 Resource Workshop

Resources are the visual parts of a Windows program, such as dialog boxes, menus, bitmaps, icons, cursors, fonts and user-defined resources. Generally, resources are kept separate from a Windows program (in a resource compiler script file .RC, for example) so that program code is not altered when alterations are made to resources and also to enable resources to be used by other programs. The Borland Resource Workshop has both a text and a graphical resource editor.

## 2.2.10 Help

The Borland C++ IDE has an on-line help system that gives immediate access to detailed information about several key topics, such as C++ keywords and standard library class and function declarations. The Borland help system is subdivided into key topics such as Essentials, Tools, Tasks, Language Reference and Master Index. A search can be performed on any C++ language feature, keywords, macros, functions, classes, Windows API, Container and ObjectWindows **class** libraries and so on. The Master Index is a complete reference to language features in the Borland C++ supplied documentation. Also, examples are given for several key language features.

Figure 2.2 is an extract from Borland C++ help on the keyword **delete**. An underline indicates a jump to an additional, related topic.

## 2.2.11 Java Development Tools

The Borland C++ (version 5.0) compiler includes development tools for the Java programming language, including Sun's Java Development Kit, which is fully integrated with Borland's IDE. The Java compiler is supported with a GUI debugger.

**delete**

See Also    Example    Operators

**Syntax**

```
<:::> delete <cast-expression>
<:::> delete [ ] <cast-expression>
delete <array-name> [ ];
```

**Description**

The **delete** operator offers dynamic storage deallocation, deallocating a memory block allocated by a previous call to **new**. It is similar but superior to the standard library function **free**. You can also use the **delete** operator to remove arrays that you no longer need.

**Fig. 2.2** An extract from Borland C++ help on the keyword **delete**.

## 2.3 Container Class and ObjectWindows Libraries

Borland C++ contains an extensive library of container classes (also known as the Borland International Data Structures (BIDS)) and ObjectWindows classes. Using such classes saves you the massive task of developing such classes yourself and thus prevents you from reinventing the wheel. In addition, these classes have been developed over a number of years by professional programmers and are, as a result, efficient and robust.

Containers are objects that implement common data structures. Borland containers are implemented using *templates*, allowing you to pass into the template any C++ or user-defined type or **class**, while ensuring correct data-hiding and providing the appropriate access to the container **class**'s data members via member functions. The Borland container **class** library is composed of two categories: Fundamental Data Structures (FDS) and Abstract Data Structures (ADS):

### Fundamental Data Structures

- Binary tree
- Hash table
- Linked list
- Double-linked list
- Vector

### Abstract Data Structures

- Array
- Association
- Dequeue
- Dictionary
- Queue
- Set
- Stack

The ObjectWindows library classes (version 5.0) are classes specifically developed to assist a programmer in developing applications for Windows 3.1 and 3.11, Win32s, Windows 95 and Windows NT more easily and quickly, and in an object-oriented programming fashion. The library is extensive and I cannot do it justice here, but a few ObjectWindows classes are listed below:

TApplication	Application <b>class</b> supplying the basic behaviour of a Windows program.
TDC	Windows GDI device context <b>class</b> performing graphical operations.
TDialog	Modal and modeless dialog box <b>class</b> .
TRect	Encapsulates a two-dimensional rectangle.
TPoint	Encapsulates a two-dimensional point ( $x, y$ ).
TWindow	Provides a generic window that can be manipulated as required. TWindow encapsulates many of the Windows API functions.

## 2.4 Summary

Your programming environment is very important. Spend some time getting to know it. Understanding your environment will make you a better C++ programmer and a happier one!

An environment such as the Borland C++ Integrated Development Environment for Windows is a world of features, tools and libraries that can reduce your development time. Becoming too dependent, however, on a particular programming environment can make the transition from one environment to another very difficult – don't get hooked.

## Exercises

- 2.1 Get to know your compiler and your compiler's development environment.
- 2.2 Examine your compiler's documentation and on-line information regarding draft ANSI/ISO C++ compatibility and portability with respect to function and class libraries.
- 2.3 Investigate which libraries your compiler is shipped with. Look for both function and **class** (and container classes) libraries for developing command-line and GUI programs.
- 2.4 See whether your compiler has several tutorial programs to help you become familiar with the processes of compiling and linking a typical program.
- 2.5 What program debugging tools does your compiler support?
- 2.6 If your compiler supports an on-line help facility, try to obtain some information about the *main()* function.
- 2.7 In the next chapter we shall examine a program file called MAIN.CPP. Create either a project or make file for this single program file.

# Getting Started

This chapter introduces us to programming in C++. As with any language (not just computer languages) there are several fundamentals that you have to acquire before you can do even the simplest of things. With the aid of the simplest of C++ programs, this chapter illustrates several of the key elements of program construction, such as language keywords, the main() function, program statements and directing program output to the display screen. The program also introduces the processes of compiling and linking a program, user comments, use of the preprocessor and header files and stresses the need for developing a consistent programming style. A good programming style will make your programs more readable and less prone to syntactical errors, and will help you to communicate better with fellow C++ programmers.



## 3.1 The Simplest of Programs

The simplest of C++ programs, which produces an output of *hello, object*, is:

```
// main.cpp
#include <iostream.h>

void main ()
{
    cout << "hello, object" << endl ;
}
```

This program follows the now traditional manner of beginning introductory books on C and C++ programming, after the book *The C Programming Language* by Kernighan and Ritchie (1978). See also the second edition (Kernighan and Ritchie, 1988).

Note that an even simpler (actually the minimal C/C++ program) is:

```
main ()
{
}
```

So, what does the above program mean and what does it actually do?

You can either type the program into your computer yourself or use the supplied file MAIN.CPP (C:\CPP\_PROG\CHAP03\MAIN.CPP, assuming that the file resides on the C drive). At this stage it is recommended that you type the program yourself to gain confidence with your editing environment and compiler, but ensure that you pay particular attention to the case of the characters in the code. The filename extension .CPP (C Plus Plus) is used by Borland C++ compilers. The extensions .CC, .CP and .C++ are frequently used by other C++ compilers.

Let us now examine each line of the program separately in slightly more detail. If certain material in what follows appears confusing, 'don't panic' – it will become clearer (believe me) as we gradually work through the chapters in order. Also, when you have successfully typed in, compiled and linked the program, don't be afraid to experiment with the program, particularly with the comment and the output message.

## 3.2 The Comment Line

The first line of the program is:

```
// main.cpp
```

The double forward slash comment characters (//) inform the compiler that what is to the right-hand side of // is a *comment* inserted within the program body and is to be ignored during compilation. The comment extends to the end of the line (which will depend on the editor that you are using) and no closing comment characters are required.

The original C style of explicitly terminated comments involves the use of the characters /\* at the beginning of the comment followed by the characters \*/ at the end of the comment:

```
/* main.cpp */
```

No spaces are allowed between the characters of //, /\* and \*/; i.e. / / or / \* are illegal. For single-line comments, as is the case here, the (//) style is generally preferred, whereas the /\* . . . \*/ style is preferred for multi-line comments:

```
/*
main.cpp
The simplest of programs

John Smith, 1st. September '94
(c) Smith & Jones Ltd.
All rights reserved
*/
```

In addition to the name of the program (MAIN.CPP), a title, name of programmer, date and copyright have been added as comments.

Comments are not just reserved for an entire line or a single block as above. The beauty of commenting is that it can be integrated neatly into the program code:

```
// main.cpp
#include <iostream.h> // header file for C++ I/O
```

---

```
void main ()
{
    cout << "hello, object" << endl ;
} // end of main()
```

As you become more experienced as a programmer and the problems that you tackle naturally become more complex then the need for good, concise commenting becomes more important. As with programming style, to be discussed shortly, commenting a program as you develop the program is a simple skill well worth mastering for two main reasons. Firstly, the process of commenting helps us to concentrate and ask ourselves ‘What am I actually trying to do here?’, just as writing down ideas on paper by hand assists our concentration. Secondly, life is an awful lot easier for yourself, and particularly others, when reading your code on returning to a piece of code some time after it was originally written. Generally, if you have a good understanding of the problem at hand then commenting is straightforward.

A few problems can arise with commenting that you should be aware of. The `//` style of comment finishes at the end of a line, with the `/*`, `*/` and `//` characters being treated just like any other character within a `//` comment:

```
// main.cpp /* not main.c */ // or main.c++
```

Since the `//` style comment extends to the end of the line, be careful of comments of the form:

```
void function ()
{ // a function without a body }
```

in which case the closing brace, `}`, is just another comment character and not the end of the function body.

Nested comments are, strictly speaking, not allowed in C and C++. An example of nested comments is:

```
/*
main.cpp
/*
The simplest of programs
*/
*/
```

Pre-ANSI C compilers allowed nested comments, and as a result several commercial compilers possess an option of allowing programs to contain nested comments (e.g. Borland’s Borland C++ for Windows (version 5.0): Project Options | Compiler | Source *Nested comments* checkbox). However, try not to use nested comments, because they are generally too confusing.

It is worth noting that comments anywhere within a program have no effect on the executable code size or on the speed of execution of a program. Comments are removed by the compiler before any program code is generated by the compiler. This can be tested by adding the comment line `// end of program` to the end of the program and comparing the execution size (.EXE) of this program with that of the original program. When I performed this comparison the execution size of both programs was 79,240 bytes. Your execution size may vary from this value depending upon differences in code generation for different compilers or different versions of the same compiler. So, if comments are free then make good use of them.

### 3.3 Preprocessor Directives

The second line of the program:

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

contains a *preprocessor directive* that informs the compiler to read a file named IOSTREAM.H into the program. A preprocessor directive is a command to the compiler's preprocessor that deals with instructions before compilation begins. A preprocessor directive begins with the number (or hash) symbol (#).

There are several preprocessor directives, such as #define, #if and #endif. The #include directive instructs the compiler to insert the *header* or *include* file IOSTREAM.H into the program at the point where the directive occurs. For example, suppose you have a file called INFO.H which contains relevant information:

```
/*
The simplest of programs
```

```
John Smith, 1st. September '94
(c) Smith & Jones Ltd.
All rights reserved
*/
```

If we rewrite the MAIN.CPP program as:

```
#include "info.h"
#include <iostream.h>

void main ()
{
    cout << "hello, object" << endl ;
```

then after the preprocessor pass the program will be:

```
/*
The simplest of programs

John Smith, 1st. September '94
(c) Smith & Jones Ltd.
All rights reserved
*/
#include <iostream.h>

void main ()
{
    cout << "hello, object" << endl ;
```

The #include preprocessor directive has inserted the header file INFO.H into the program.

## 3.4 What is IOSTREAM.H?

IOSTREAM.H contains **class** declarations that are essential for programs that want to perform input and output; namely `ios`, `istream`, `ostream` and `iostream`. The file derives its name from input/output (io) stream. A *stream* is in fact an abstraction of data flow. In order to use the `cout` identifier and `<<` operator (which we will examine shortly) the compiler needs to have their declarations by including the appropriate header file.

Header files should, in general, not contain any executable program code, and a glance through IOSTREAM.H confirms that the file consists entirely of **class** declarations for input and output operations. The implementation of these **class** member functions is restricted to an implementation file (.CPP). If you do intend to look through IOSTREAM.H then make sure that you *do not* alter the file in any way. Such standard C++ header files are essential to the correct operation of your programs.

## 3.5 " " or <>

You may have noticed above that the file INFO.H is enclosed between double quotes (" "), whereas IOSTREAM.H is enclosed between angle brackets (<>). The `#include` preprocessor directive may take one of two forms for header files:

```
#include <header_filename>
#include "header_filename"
```

`header_filename` must be a valid filename with an extension traditionally of the form .H for header. The pathname and delimiters are optional. For instance we could use the exact path for INFO.H:

```
#include "c:\cpp_prog\chap03\info.h"
```

Note that a single backslash character (\) is sufficient to indicate subdirectories for a preprocessor directive.

(<>) indicates a *standard C++* include file and the preprocessor looks for the file in the Include directories in the order that they are defined. For the Borland C++ compiler (see Options | Project | Directories) the Include directories resident on the C drive are:

```
C:\BC5\INCLUDE;C:\BC5\INCLUDE\OWL;C:\BC5\INCLUDE\CLASSLIB
```

(" ") indicates a user-defined include file, and the preprocessor first looks in the current directory for the header file. The current directory is usually the directory where the source file being compiled resides. If the file included between double quotes is not found, then the include directories are searched, as in the (<>) case.

Having these two different `#includes` for header files allows the standard C++ header files provided with your compiler to be kept separate from user-defined header files. Also, making the correct choice between (" ") and (<>) will speed up compilation.

### 3.5.1 Other Header Files

If you browse through the subdirectories of your compiler you will probably find a directory called INCLUDE or HEADERS. For the Borland C++ (version 5.0) compiler the directory name

is C:\BC5\INCLUDE. This directory contains several header files, such as \_DEFS.H, IOS-TREAM.H and WSNLINK.H. We shall discuss header files in more detail at a later stage when we discuss libraries, but for now it is enough to know that each header file is an interface to a particular library.

### 3.5.2 .H or .HPP

You may be asking yourself, ‘If .CPP denotes an implementation file for a C++ program, then why don’t I similarly use .HPP to denote a C++ header file?’. I agree that .HPP clearly distinguishes a C++ header file from the traditional C header file, .H. However, unlike .CPP, which is a one-to-one abbreviation for C++ (C Plus Plus), .HPP (Header Plus Plus) appears meaningless. Hence I prefer .H (Header). Also, the C++ standard library header files (such as IOSTREAM.H) use the extension .H instead of .HPP, although the new style of naming header files proposed in the draft ANSI/ISO C++ standard is by name only, e.g. IOSTREAM.

## 3.6 Keywords

A keyword, or reserved word, such as **void**, is a feature of the C++ language which has associated with it some special meaning. Other keywords include **char**, **int** and **class**.

A keyword cannot be used as an identifier, object or function name:

```
//...
void void ()    // error
{
//...
int char ;    // error
//...
}
```

Appendix A lists all the keywords of the C++ programming language.

## 3.7 The *main()* Function

Every C and C++ console-based program must call the *main()* function at program-startup:

```
void main ()
{
//...
}
```

It is possible not to use *main()* and define an alternative function name:

```
void my_main ()
//...
```

and still have successful program compilation. However, if there is no `main()` function in your program the linker will produce an error depending on the memory model used.

Inter-word spaces are not allowed for keywords, function names, `class` names etc. Thus `vo id` and `my main()` are illegal and hence the reason for using the underscore: `my_main()`.

The function name `main` is followed by a pair of parentheses (opening and closing). Since `main` is not a *keyword* in C++, the parentheses after the function name indicate to the compiler that `main()` is a function.

Note that `main()` is not a user-defined function. The convention used throughout this book is that user-defined function names begin with an uppercase letter. Thus, the equivalent user-defined `main()` function would be `Main()`. These two functions, `main()` and `Main()`, are two different functions because C++ is a case-sensitive language. Let us just emphasise this important point:

*C++ is a case-sensitive language*

The following program illustrates this:

```
// capitalm.cpp
// illustrates that C++ is a case-sensitive language
#include <iostream.h>

void main ()
{
    cout << "hello, object" << endl ;
}

void Main () // O.K.- Main() is different from main()
{
    cout << "Hello, Object" << endl ;
}
```

If the uppercase letter M in `Main()` is replaced with a lowercase m you will get a compilation error of the form ‘function `main()` already defined’. Therefore `main()`, `Main()` and `MAIN()` all have different meanings in C++. Make sure that the `main()` you use is always in lowercase.

Functions, C++ and user-defined, will be discussed in more detail at a later stage, but for now the essential components of a function are:

```
return_type FunctionName (parameter_list)
{ // opening brace
/*
function body

*/
} // closing brace

return_type:      The type of the value, or void, returned by the function.
FunctionName:     Name of the function.
parameter_list:   List of parameters, or void, passed to the function. Also known as
                  function arguments or formal arguments.
```

**function body:** The opening brace, {, indicates the start of the function, while the closing brace, }, indicates the end of the function body. Any code that appears between the pair of braces { and } is the function body.

Notice that `main()` has no parameter list. If a function does not take any function parameters you can use the **void** keyword to indicate this explicitly:

```
void main (void)
{
//...
}
```

Generally, `main()` and `main(void)` are equivalent, but there are some C++ implementations where this is not the case. Therefore, if you use **void** for the parameter list be sure that the function has no parameters.

**void** as a function return type indicates that the function does not return a value. Although **int** is the default return type of `main()`, some compilers insist that `main()` returns an **int**. The **int** keyword (or *type specifier* to be more specific) will be discussed later, but if you are getting any warning messages concerning the return type of `main()` then use the following format:

```
int main ()
{
//...
return 0 ; // main() returns 0 to the operating system
}
```

placing the **return 0** just before the end brace of `main()`. The return value of the `main()` function indicates to the operating system the program's exit status. Generally, a return value of 0 indicates that the program terminated normally, whereas a return value of -1 indicates abnormal termination.

## 3.8 Program Statements

Program statements are the fundamental units of C++ programming. The program statement:

```
cout << "hello, object" << endl ;
```

instructs the computer to display the string constant *hello, object* followed by a new line. Try recompiling the program leaving out the semicolon (;) – this is an error. The semicolon indicates the end of the statement. The compiler ensures that a statement is completed before the next statement is executed. The semicolon is easily forgotten, so get into the habit early of ending with a semicolon. Fortunately, compilers are very good at detecting missing semicolons, so this is not a big deal.

The semicolon can work to your advantage sometimes by offering a more succinct form of assignment:

```
int x, y, z ;
```

---

```
//...
x = 1; y = 2; z = 3 ;
```

rather than:

```
//...
x = 1 ;
y = 2 ;
z = 3 ;
```

The minimal statement possible in C++ is the *null statement*, which simply consists of a semicolon:

```
//...
;
```

### 3.8.1 The cout Identifier/Object

`cout` (common output) is a standard output stream identifier or object. Output, by default, is directed to the screen, but can be redirected to alternative devices.

### 3.8.2 The << Insertion Operator

`<<` is the *insertion* or ‘put-to’ operator. It simply directs *what’s on the right of <<* to *what’s on the left of <<*. In the present example `<<` directs the string *hello, object* to `cout` for displaying.

The `<<` operator can be *cascaded* or *chained*. Cascading is the multiple use of `<<` within a single statement. For instance:

```
cout << "hello, object" << endl ;
```

first of all sends the string *hello, object* to `cout` and then sends `endl` to `cout`.

If you are familiar with C, the equivalent would be:

```
printf ("hello, object\n") ; // need to: #include <stdio.h>
```

which is function-based and messy.

### 3.8.3 String Constants

The text *hello, object* between the double quotes is a string constant. Being a constant, the string does not alter throughout the entire lifetime of the program.

### 3.8.4 The endl Manipulator

The `endl` manipulator inserts a line feed into the stream. This ensures that any following output is on a new line and not a continuation of the present line. You are completely free to choose where you place `endl` and how many new lines you require; for example:

```
cout << endl << "hello, object" << endl << endl ;
```

which inserts a new line before the output and two new lines after the string. Note that an insertion operator is required for each `endl`. The following is illegal:

```
cout << "hello, object" << endl endl ; // error
```

`endl` is equivalent to the C ‘\n’ *escape sequence* – just more elegant.

## 3.9 Compiling and Linking

The program source code in MAIN.CPP is a text file. To generate an executable program there are two steps that need to be performed: *compiling* and *linking*.

### 3.9.1 Compiling

First, your source code is compiled into an object file (.OBJ), which is not to be confused with an object in object-oriented programming. An object file contains machine-language instructions which can be executed by a computer. Standard C++ and user-defined header files are #included in your source code by the preprocessor. The header files will contain information necessary for compilation, such as IOSTREAM.H for the `cout` identifier/object and `<<` operator in the example above. Depending on the memory model used, the input/output functions, classes and objects are in a library routine (CWL.LIB for a large memory model).

The Borland C++ IDE lets you select either the menu option ‘Project | Compile’ or a button on the SpeedBar to compile a program. The resulting dialog box provides statistics on the number of warnings and errors and the number of lines generated.

### 3.9.2 Linking

Invariably, a program consists of a number of different object and library files. Linking simply combines the separate object and library files into a single executable program. Provided that the necessary object and library files exist and are correctly accessed, the linker will extract the appropriate information from the files and generate the executable program.

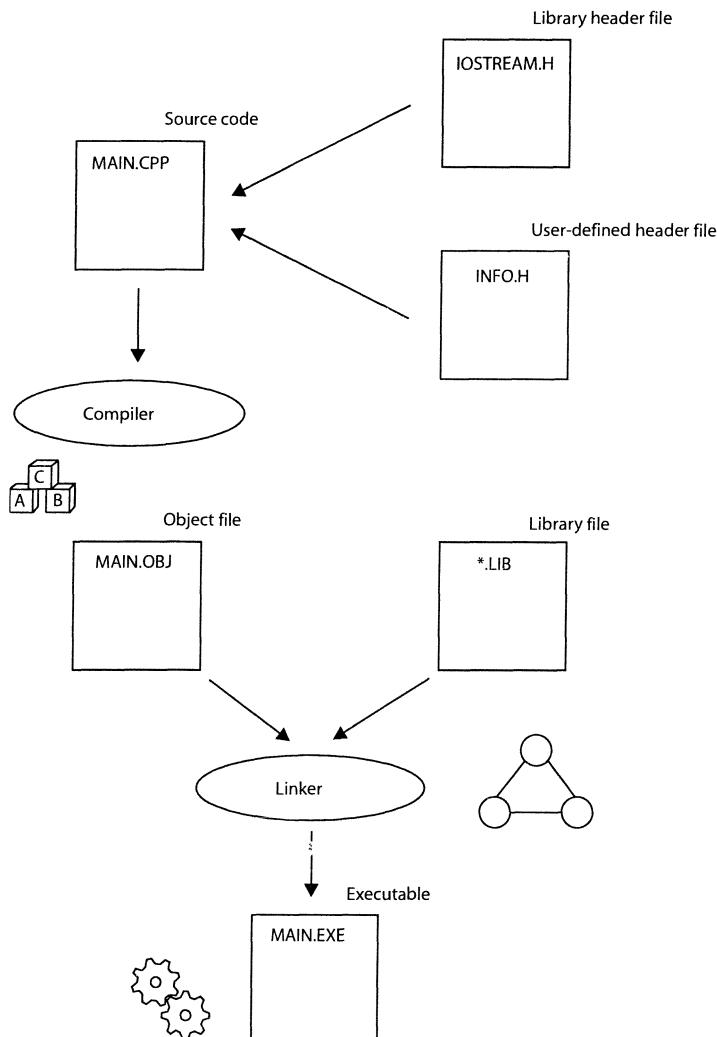
As with compiling, the Borland C++ IDE allows you to link a program either by selecting ‘Project | Make all’ or by the compile and link button on the SpeedBar. Similarly, linking statistics are generated.

Figure 3.1 schematically illustrates the general process of compile and link.

## 3.10 A Note on Programming Style and Notation

C++ is a *free-format* language. Provided the code is syntactically correct, the compiler almost completely ignores *white space*. White space is defined as spaces, tabs, new lines, line feeds, form feeds and carriage returns. Thus, there are no restrictions on the style to adopt in formatting your program code. There are several styles that can be adopted when programming. Three frequently used alternative formats for the `main()` function body are:

```
void main ()
```

**Fig. 3.1** Compilation and linking.

```

{
cout << "hello, object" << endl ;
}

void main () {
...
}

void main () { cout << "hello, object" << endl ; }

```

Clearly, the possibilities are endless. It is not my intention to enforce any particular style on the reader, but it is necessary to be aware of the style adopted in this book and to bear in mind the variety of styles around.

A couple of exceptions that prevent C++ from being a completely free-format language are lines that begin with the preprocessor directive # and continuation lines. For example, the following is illegal:

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

A preprocessing directive and any other line may be extended on to the next line by the insertion of a backslash character, \, at the end of the line to be continued:

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

You may be wondering why you would want to do this, but there are occasions when an expression or a string of text is too long to fit on one line:

```
cout << "this is a very long string of text \  
 that extends beyond the width of my screen" << endl ;
```

You may have noticed that I place spaces between function names and parentheses and between the end of statement semicolon and the last character on the line. This is a style that I picked up from Charles Petzold's classic book *Programming Windows 3.1* (Petzold, 1992). The following lines are equivalent:

```
//...  
cout<<"Petzold's book: "<<book.Petzold()<<endl;  
  
cout << "Petzold's book: " << book.Petzold () << endl ;
```

In my opinion the latter adds clarity to the program code.

Also, I adopt a style of aligning types, function declarations and assignments whenever possible (don't worry about the specifics of the code for the moment):

```
class Person  
{  
protected:  
    char* name ;  
    int age_in_years ;  
    double height_in_metres ;  
//...  
public:  
    Person () ;  
    void OutputStatistics () ;  
    virtual void Occupation () ;  
//...  
};
```

rather than:

```
class Person  
{
```

---

```

protected:
    char* name ;
    int age_in_years ;
    double height_in_metres ;
    //...
public:
    Person () ;
    void OutputStatistics () ;
    virtual void Occupation () ;
    //...
};

```

The above example also illustrates the convention adopted for naming classes and variables/objects. A **class** name always begins with an uppercase letter, such as **Person**. This is to distinguish **class** names from variables and objects, which are in lowercase, e.g. **name**. In the case of multi-word **class** and function names, uppercase letters are used (**EmployedPerson**, **OutputStatistics()**), while an underscore is used for variables/objects (**age\_in\_years**). These conventions are clearly more readable than, say, **employedperson** or **ageinyears**. More importantly, such a convention helps to distinguish between classes, functions, identifiers, objects etc. in a complex program listing. Borland C++ adopts the convention of preceding a **class** name by the letter T (T for type), e.g. **TPerson** or **TPoint**.

Constants, things that don't change throughout the entire program once defined and initialised, are defined entirely in uppercase:

```

const int I_CONST = 0 ;
const double FEIGENBAUM = 4.6692016 ;
//...

```

C++ recognises the first 32 characters of an identifier's name. More than 32 can be used, but the extra characters are not significant.

Some of the above styles may appear to be fairly trivial and subtle mannerisms, but they are nevertheless worth thinking about. Choose a formatting style that best suits you and enhances your programming style. However, think of other programmers who will have to read your program code, particularly if you work in a programming group.

### 3.10.1 Be Consistent!

The style that you adopt will continue to change as you gradually become a more experienced programmer, but try to consciously develop your style and be consistent. A consistent style of writing code will help your own readability of a program.

Consider the following cases:

```

// main.cpp
#include <iostream.h>
void main () { cout << "hello, object" << endl ; }

```

or:

```

//...
void main () {

```

```
cout <<
"hello, object" << endl ;
}
```

which are (to me) a bit confusing.

The style that I use will become clearer as we proceed through the book.

### 3.11 Summary

This chapter has introduced us to a few basics of a C++ program. We have seen two types of user *comment*, namely `//` and `/* . . . */`. The *preprocessor directive* `#include` informs the compiler to insert a *header file* at the point specified. Every console-based program must have a *main() function* which marks the start and end points of a C++ program. Every function is delimited by a pair of braces or curly brackets: `{` and `}`. We noted that C++ is a case-sensitive language, so that the functions `main()` and `Main()` have two different meanings. Output in C++ is commonly performed using the `cout` object/identifier and the `<<` *insertion operator*. The `endl` manipulator inserts a line feed into the stream. `endl` is equivalent to the C ‘\n’ *escape sequence*. A string of text is placed within double quotes (“ . . . ”).

## Exercises

- 3.1 To help you get started with the C++ programming language and your compiler, write a program that outputs your name, address and personal details (whatever they might be!).
- 3.2 Place two `endl` manipulators before the *hello, object* string of MAIN.CPP.
- 3.3 Debug the following program:

```
// 3_3.cpp
a C++ program with a few errors

#include <iostream.h> // C++ I/O

void Main ()
{
    cout << "The C++ Programming Language < endl ;
// } main()
```

- 3.4 Locate and open the IOSTREAM.H header file and find the declarations for the `cout` stream object and the `endl` stream manipulator.
- 3.5 Include a header file in MAIN.CPP which includes details about yourself, program information, date of editing and a program copyright.
- 3.6 Try changing the angle brackets with double quotes for including IOSTREAM.H in MAIN.CPP. Include the exact path of IOSTREAM.H.
- 3.7 Modify the `main()` function in MAIN.CPP so that it returns type `int`.

# Fundamental Data Types, Declarations, Definitions and Expressions

*live with the language*  
g. w.

*This chapter begins by describing the different kinds of Fundamental Data Type (FDT) in C++. The chapter then introduces identifiers (constant and variable) and the declaration and definition of identifiers. This is followed by discussing assignment; initialisation of identifiers, expressions and statements; operators; and type conversions. Basic input from the keyboard and output to the screen is covered, along with the use of manipulators for formatting input and output. The topic of casting (the process of converting an object of one data type to an object of another data type) is discussed. The chapter concludes by examining the auto, extern, register and static storage class specifiers for specifying exactly how a variable is to be stored, and the asm declaration for directly integrating assembly language code into C++ code.*



## 4.1 Fundamental Data Types

There are four *fundamental, basic or primitive* data types in C++:

<b>char</b>	Character: letter, symbol, punctuation and digit
<b>int</b>	Integer: whole number
<b>float</b>	Floating: point number
<b>double</b>	Floating: point number

**char, int, float** and **double** are keywords or, more specifically, *type specifiers*. Each data type has its own *attribute list* which defines the characteristics of the type and may vary from one machine system to another. The types **char**, **int** and **double** each have variations, or *data type modifiers*, such as **short**, **long**, **signed** and **unsigned**, to enable more efficient use of the data types.

An additional character type is **wchar\_t**:

<b>wchar_t</b>	Wide character constant
----------------	-------------------------

There are also two additional types in C++ that I feel warrant mention at this point, namely **enum** and **bool**:

<b>enum</b>	Enumeration constant
<b>bool</b>	False-True enumeration

Compared with the fundamental types, **enum** and **bool** may appear to be oddballs, but just bear them in mind at this stage until we discuss them in more detail at a later date.

The characteristics of each data type will now be discussed separately.

#### 4.1.1 The **char** Data Type

The **char** data type is just large enough to store one character of your machine's character set. Variables of type **char** can be used to store numbers (-128 to 127 for a signed **char** and 0 to 255 for an unsigned **char**), but are generally used to hold the ASCII characters. Most C++ compilers use the ASCII character set, which has defined characters from 0 to 126. However, most compilers extend this range from 0 to 255 so as to accommodate the extended IBM character set. Appendix B contains a complete listing of the ASCII character set. For example, the decimal integers 0 to 9 are stored as decimal constants 48 to 57 and the lowercase letters a, b, ... z are stored as decimal constants 97 to 122. Symbols such as +, - and # all have an equivalent ASCII character constant. At first it can be a little confusing to find out that characters are stored as integers which in turn are stored as binaries. Appendix B also lists the binary equivalents of the character set. Note that characters in the ASCII character set from 0 to 127, inclusive, could be represented as seven data bits instead of eight bits. However, since for most computers the byte (8 bits) is the primary unit, a variable of type **char** occupies one byte of memory.

For MS-DOS, characters 0–31 are reserved for control characters, 32–126 represent keys on the keyboard and 127–255 are the IBM extended characters. It is worth noting that the IBM extended character set is not part of the ASCII character set and may not be fully portable. MS-DOS also includes character sets or code pages. For example, the English and multilingual character sets are designated by the numbers 437 and 850 respectively. Note also that the Windows character sets differ from the MS-DOS character sets.

##### **Character Constants**

Character constants are written in C++ as a single character enclosed in apostrophes (or single quotes): 'C', 'p', '+'. Character constants are different from string constants enclosed within double quotes:

```
//...
char char_const = 'C' ; // character constant
//...
cout << "this is a string constant" ; // string constant
//...
```

When a character constant is encountered, the compiler converts it into its respective numeric code. The following example program illustrates character constants and variables:

```
// char.cpp
// illustrates the char data type
#include <iostream.h> // C++ I/O
```

```

void main ()
{
    cout << "'A': " << 'A' << endl << endl ; // character
                                                // constant

    char c_var1 ;           // definition of character variable
    c_var1 = 'C' ;          // assignment
    cout << "c_var1: " << c_var1 << endl ;

    //char c_var1 ;          // error: multiple definition for
                           // 'c_var1'

    char c_var2 = 'p' ; // define & initialise thro'
                        // assignment
    cout << "c_var2: " << c_var2 << endl ;
    c_var2 = 43 ;          // re-assignment
    cout << "c_var2: " << c_var2 << endl ;

    cout << << c_var1 << c_var2 << c_var2 << endl ;

    const char C_CONST = 'C' ;
    //const char C_CONST ; // error: must initialise a
                          // constant
    //C_CONST = 'p' ;      // error: can't modify a constant
}

```

which generates the following output:

```

'A': A

c_var1: C
c_var2: p
c_var2: +
C++

```

Note that the more explicit definition of **signed char** could have been used in place of **char** in the above example program, since the two are equivalent.

The program first of all outputs the character constant 'A'. Next a variable, **c\_var1**, of type **char** is defined. This is directly followed by assigning the character 'C' to the variable **c\_var1**, which is then displayed. Then a single line has been commented out which would cause a compilation error. The line defines **c\_var1** a second time, and multiple definitions of a variable are illegal. In C++ all variables must be defined before they are used. The next line illustrates that C++ allows you to define a variable or object wherever it is required. The variable **c\_var2** is defined and initialised (by assignment) in a single statement. You may be thinking 'big deal', but the traditional C style is to define all variables and objects at the top of a block before the first executable statement:

```

// C style:
char c_var1 ;
char c_var2 ;
//...

```

```
c_var2 = 112 ; // assignment
cout << "c_var2: " << c_var2 << endl ;
//...
```

In a small example like this, the C++ *define-anywhere* style may not appear any better than the C *define-at-top* style, but when program code becomes lengthy and complex you'll be grateful for the C++ style. The C++ style is also less error-prone.

After the value of `c_var2` is displayed, the next line assigns a new character to `c_var2` by assigning the decimal equivalent, 43, of the ASCII character '+'. Although it is possible to assign a character to a `char` variable or constant by assigning the character's decimal equivalent, it is good programming practice to assign the actual character constant, if possible. There are occasions, however, such as when displaying non-keyboard characters (e.g. the → character, with decimal equivalent 26), when the decimal equivalent of a ASCII character has to be used.

In the above program `c_var2` was declared as a variable, and variables can have variable values. You may be thinking, 'How do I, or the compiler, know that `c_var2` is a variable when I didn't explicitly define it as a variable?'. Basically, if an identifier is not defined as a constant it is assumed to be a variable. The program example above defines and initialises a `char` constant, `C_CONST`, to the character 'C'. Two commented lines are included within the program to indicate two key rules for constants: (1) a constant object must be initialised when defined and (2) a constant cannot be modified after initialisation. These rules apply to all `const` objects of an integral or user-defined data type and are not just restricted to objects of type `char`. The modifier `const` is discussed later.

Furthermore, some languages (such as Pascal) require that objects are explicitly defined as either constants or variables:

```
// Pascal:
const
    PI      = 3.14159 ;
    BLACK  = 0 ; WHITE = 1 ;
var
    x, y, z: real ;
    quit:   boolean ;
//...
```

but C++ adopts a more minimal style than this.

If we required several variables we could define each one on a separate line:

```
char c_var1 ;
char c_var2 ;
//...
char c_varn ;
```

An alternative approach is to use a comma ( , ) as a separator:

```
char c_var1, c_var2, c_var3 ;
```

and if we require the variables to be initialised then we can take either of the following two approaches:

```
char c_v1 = 'a' ; char c_v2 = 'b' ; char c_v3 = 'c' ;
char c_v1 = 'a', c_v2 = 'b', c_v3 = 'c' ;
```

The *comma* operator allows us to separate expressions but not separate statements. Use the comma operator when you have expressions with the same attributes; e.g. `c_v1`, `c_v2` and `c_v3` are all of type `char`.

### **Character String Constants**

We saw above that character constants are different from string constants. Basically, a string constant consists of a series or *string* of characters. For example:

```
"C++ is a programming language"
```

### **Escape Sequences**

In C++ there are certain character constants that have a special meaning. For instance, you may have seen the new line escape sequence '`\n`' used:

```
cout << "M. C. Escher\n" ;
```

instead of:

```
cout << "M. C. Escher" << endl ;
```

The backslash character (`\`) indicates a special sequence to the compiler. In the previous example '`\n`' is interpreted as a new line and not as the constant character '`n`'. Table 4.1 lists the escape sequences available in C++.

Note that the escape sequences are viewed by the compiler as a single character and thus occupy one byte and not two, as may have been expected.

Any character in the ASCII set can be alternatively described in terms of escape sequences by using the character's octal or hexadecimal code. For example '`\60`' and '`x30`' and '`48`' represent the digit 0 in octal, hexadecimal and decimal ASCII character constants respectively.

Escape sequences can be useful for string constants containing double quotes. For example, imagine you wanted to display the string *the man said "thank you."*:

```
cout << "the man said \"thankyou\"." << endl ; // error
```

**Table 4.1** Escape sequences.

<i>Escape sequence</i>	<i>Character</i>
<code>\a</code>	Alert. Audible (bell) or visible alert.
<code>\b</code>	Backspace. Move to previous position.
<code>\f</code>	Form feed. Advance to next page.
<code>\n</code>	New line (line feed). Advance to next line.
<code>\r</code>	Carriage return. Current position becomes 1st. position on next line.
<code>\t</code>	Horizontal tab. Advance to next horizontal tab position.
<code>\v</code>	Vertical tab. Advance to next vertical tab position.
<code>\\\</code>	Backslash. Display backslash.
<code>\'</code>	Single quote (apostrophe). Display single quote.
<code>\"</code>	Double quote. Display double quote.
<code>\?</code>	Question mark. Display question mark.
<code>\0</code>	Null termination character.
<code>\o</code>	Octal code. $o =$ string of 3 digits maximum.
<code>\xh</code>	Hexadecimal code. $h =$ string of hexadecimal digits.

**Table 4.2** Trigraphs.

#	[	]	{	}	\	^		~
?=?	??(	??)	??<	??>	??/	??'	??!	??-

```
cout << "the man said \"thankyou\"." << endl ; // O.K. with
// '\"' e.s.
```

The first statement is an error because the compiler interprets the character string as "the man said ".

### Trigraphs and Digraphs

For several European computers the ASCII special characters [, ], {, }, | and \ occupy alphabetical positions designated by the ISO standard. Also, certain European keyboards are without the number sign (#). The C *trigraphs* offer a solution to this problem; see Table 4.2. Thus, the following two program examples are equivalent:

```
//...
void main ()
{
//...
}
```

```
//...
void main ()
??<
//...
??>
```

Trigraphs are not very readable. An alternative to trigraphs is *digraphs*<sup>1</sup>; see Table 4.3. Note that there is not a complete overlap between trigraphs and digraphs. In terms of digraphs the above example is now:

```
//...
void main ()
<%
//...
%>
```

Note also the keywords for the bitwise operators (&, |, ^, ~ and ^=), relational operators (! and !=) and logical operators (&&, ||, &= and |=). Such operators will be discussed later.

### 4.1.2 The **int** Data Type

The **int** data type is used to store integer or whole numbers such as 0 or 21, but not numbers with a fractional part. The size of an **int** depends on the largest data register (16 or 32 bits) of

<sup>1</sup> A digraph is a token consisting of two characters.

**Table 4.3** Digraphs and keywords for C++ special characters.

<i>Digraphs</i>	<i>Keywords</i>
[ < :	& <b>bitand</b>
] :>	<b>bitor</b>
{ <%	^ <b>xor</b>
} %>	~ <b>compl</b>
# %:	^= <b>xor_eq</b>
## %:%:	! <b>not</b>
	!= <b>not_eq</b>
	&& <b>and</b>
	<b>or</b>
	&= <b>and_eq</b>
	= <b>or_eq</b>

the computer. For a 16-bit machine the high bit is used for the sign of the integer, so the range of values is from  $-32\,768 (-2^{15})$  to  $32\,767 (2^{15}-1)$ . For a 32-bit machine an **int** has the range  $-2\,147\,483\,648 (-2^{31})$  to  $2\,147\,483\,647 (2^{31}-1)$ , with the high bit similarly reserved for the sign of the number. To combat the portability problem of transferring a program from one machine to another, C++ allows the **int** data types to be modified through the use of the **short** and **long** data modifiers, to be discussed later.

The following example illustrates a few experiments with the **int** type.

```
// int.cpp
// illustrates the int data type
#include <iostream.h> // C++ I/O

void main ()
{
    int i ; // definition of i

    i = 4 ; // assignment
    cout << "i: " << i << endl ;

    const int CI = 90 ;
    // CI = 11 ; // error: can't modify a constant
    cout << "CI: " << CI << endl ;

    i = i + CI ; // addition and assignment
    cout << "i + CI: " << i << endl ;

    int j = 1900 ; // definition and assignment of j

    cout << "i + j: " << (i + j) << endl ;

    int const CJ = 91 ; // 'int const' instead of 'const int'
}
```

with output:

i: 4

```
CI: 90
i + CI: 94
i + j: 1994
```

Most of the statements are self-explanatory, but it is worth pointing out that, after the statement `i=i+CI;`, `i` is assigned a new value of 94. Note, as with the character example program, a `const` object cannot be altered after it has been initialised. Experiment with the program until you feel more comfortable with the `int` data type.

The library header file LIMITS.H defines the implementation-specific limits for both the `char` and `int` data types.

#### 4.1.3 The `float` Data Type

The `float` type is used to represent floating-point numbers. A floating-point number has an integer part, a decimal point and a fraction, e.g. 28.75. Type `float` stores a number within the range  $3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$ . Most compilers conform to the IEEE standard, which enables you to represent a floating-point number in terms of an exponent (e or E), e.g.  $1.25e10$  ( $=1.25 \times 10^{10}$ ).

If you require the exact specification of a floating-point number that your compiler uses then refer to the FLOAT.H header file. The minimum and maximum floating-point numbers associated with the `float` type are `FLT_MIN` and `FLT_MAX`, respectively. The number of digits of precision is set by the constant `FLT_DIG`, and for a `float` it is generally set to 7. Most compilers allocate four bytes of memory to a `float`.

The following example program (FLOAT.CPP) experiments with the `float` data type:

```
// float.cpp
// illustrates the float data type
#include <iostream.h>    // C++ I/O

void main ()
{
    float f ;    // definition

    f = 5.6 ;    // assignment
    cout << "f: " << f << endl ;

    float      fi = 0.0000001 ;
    const float CF = 1.25e10 ;

    cout << "fi: " << fi << " ; CF: " << CF << endl ;
}
```

with corresponding output:

```
f: 5.6
fi: 1e-07; CF: 1.25e+10
```

Notice that the output is automatically converted into IEEE format with explicit sign indication. Note also the chained output using the `cout` object and insertion operator `<<`.

#### 4.1.4 The **double** Data Type

The **double** data type is also used to store floating-point numbers. Most compilers allocate eight bytes of memory to store a **double**. Type **double** stores a number within the range  $1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$ . As with type **float**, if you require the exact specification of a **double** floating-point number that your compiler uses then refer to the FLOAT.H header file. The minimum and maximum floating-point numbers associated with the **double** type are DBL\_MIN and DBL\_MAX, respectively. The number of digits of precision is set by the constant DBL\_DIG and for a **double** it is generally set to 15. In addition to **float** and **double**, C++ offers us an additional floating-point type, **long double**, which has a range of  $3.4 \times 10^{-4392}$  to  $1.1 \times 10^{4932}$  and generally occupies 10 bytes (but refer to your C++ compiler's documentation for the exact size).

The following program uses **double** exclusively to prompt for the user's height in metres, convert the height to feet and output the person's height in both metres and feet:

```
// double.cpp
// illustrates the double type
#include <iostream.h> // C++ I/O

void main ()
{
    const double m_to_ft = 3.281 ;           // conversion
                                         // factor
    double      height_m = 0.0 ;           // initialise to
                                         // zero

    cout << "enter your height in metres: " ;
    cin  >> height_m ;

    double height_ft = m_to_ft * height_m ; // perform
                                              // conversion

    cout << "your height: " << height_m << "m; "
        << height_ft << "ft" << endl ;
}
```

A sample run is:

```
enter your height in metres: 1.8
your height: 1.8m; 5.9058ft
```

The above program introduces the `cin stream object`, which lets you input a value into the variable `height_m`. The user-value entered is placed in the `cin` object, and then the *extraction operator* (`>>`) transfers the value from the left-hand side of the operator to the right-hand side. The object `cin` is defined in the header file IOSTREAM.H.

Having now introduced both integer and floating-point numbers, consider the following statement:

```
//...
double x = 5 ;
```

Is `x` a **double** or an **integer**? Some old compilers would have created the constant 5 as an **int** and then cast this into `x`, making `x` an **int**. Modern compilers do not require a decimal point after a numeric constant and hence `x` is stored, as defined, as a **double**. However, I think it is clearer to define a floating-point number explicitly as a floating-point number and not as an integer:

```
//...
double x = 5.0 ;
```

### Epsilon Error

An epsilon error is a small floating-point number such that if it were added to the number 1.0 then the resulting number (number+epsilon error) would be detected by the compiler as *not* being equal to 1.0. Basically this is equivalent to the number of digits of precision offered by the compiler for each floating-point type. The exact values of the epsilon error for **float**, **double** and **long double** can be found in the FLOAT.H header file, defined as **FLT\_EPSILON**, **DBL\_EPSILON** and **LDBL\_EPSILON** respectively. For the Borland C++ (version 5.0) compiler these are:

```
#define DBL_EPSILON      2.2204460492503131E-16
#define FLT_EPSILON       1.19209290E-07F
#define LDBL_EPSILON      1.084202172485504E-19L
```

### 4.1.5 The **wchar\_t** Type

**wchar\_t** is a fundamental data type that can represent a character that will not fit into the storage space of one byte allocated by type **char**. A **wchar\_t** is defined as a **typedef** in the header files STDLIB.H and STDDEF.H as:

```
typedef unsigned short wchar_t ;
```

and thus is stored in two bytes.

A character constant that is preceded by an L is a wide character constant of type **wchar\_t**:

```
wchar_t wcc = L'12' ;
```

which is a two-character constant.

A string constant that is similarly preceded by an L is a wide character string of type **wchar\_t**:

```
wchar_t wcs[] = L"abcd" ;
```

which is a multi-byte string constant with a memory allocation of two bytes per character. In the case of a wide character string constant the string is required to be defined as an array, `wcs[]`. Arrays are covered in Chapter 7.

Note that an identifier or string of type **wchar\_t** is a constant.

It is worth mentioning that the C++ draft standard defines **wchar\_t** as a keyword and that a **wstring** **class** has been added to the standard C++ library. Classes are discussed in Chapter 9.

## 4.1.6 The **enum** Type

At first, **enums** can appear a little confusing. Basically **enum** is short for enumeration. No clearer! To enumerate means to count one by one. Thus an **enum** is a user-defined type with named constants of type integer (including negative values) and can be used in any expression where integer constants are valid.

Contrary to the previous four *integral* types, **enum** is a user-defined type.

Suppose we wanted to define three constants RED, GREEN and BLUE and assign the values 0, 1 and 2 to them respectively. One way would be:

```
const int RED    = 0 ;
const int GREEN = 1 ;
const int BLUE  = 2 ;
```

An equivalent, but more elegant, way would be to define the three constants as enumerators:

```
enum
{
    RED, GREEN, BLUE
};
```

Note the use of the parentheses and a semicolon.

By default the assigned enumerator values incrementally increase from 0 (i.e. 0, 1, 2, 3,...), and hence RED=0, GREEN=1 and BLUE=3.

Why use the enumerators RED, GREEN and BLUE? Well, most people find it a lot easier to use constants as mnemonics rather than using numbers like 0, 1 and 2. It is simply easier to work with a descriptive identifier than a number.

An enumeration can be named:

```
enum Colour
{
    RED, GREEN, BLUE
};
```

The **enum** declaration can also incorporate a variable definition, colour:

```
enum Colour
{
    RED, GREEN, BLUE
} colour ;
```

The enumerators can also be assigned explicit values during declaration:

```
enum Time
{
    SEC_PER_MIN      = 60,
    MIN_PER_HOUR     = 60,
    HOUR_PER_DAY     = 24
};
//...
cout << "minutes per hour: " << MIN_PER_HOUR << endl ;
```

which is a little strange, because assigning explicit, arbitrary (non-incrementing) values appears to contradict the definition of an enumeration. One might have expected the **enum** syntax to allow us to define an initial enumerator value explicitly, after which successive enumerators incrementally increase or decrease, but not to allow enumerators to be arbitrarily initialised. However, explicitly assigning values to enumerators *is legal*.

Note that the value of an enumerator is of type **int**, and it is possible to assign an **int** to an enumerator:

```
//...
Colour colour ;
//...
colour = GREEN ; // O.K.
colour = 1 ;       // O.K. but warning issued!
```

This is a good example of C++ being weakly typed, because a user-defined type, such as **Colour**, is indistinguishable from the fundamental type **int**. Although not strictly a compilation error, it does introduce a contradiction between **enum** and the **struct** or **class**. When **enum**, **struct** and **class** are described in more detail later, it will become evident that **enum** is similar to a **class**. With regard to classes, C++ is a strongly typed language, whereas with regard to enumerations C++ is weakly typed. Another example of weak typing in C++ is the use of **typedefs**. A **typedef** is not a new type but a synonym for an alternative type.

```
typedef int Int ;
//...
int i = 1 ;
Int j = 2 ;
//...
j = i ; // mixing of types int and Int
```

Although **enum**, **typedef**, **struct** and **class** will be discussed in Chapters 8 and 9, they have been mentioned here to highlight possible contradictions in compiler warnings that you may be experiencing. The Borland C++ (version 5.0) compiler issues a warning of the form ‘Initializing Colour with int’ for the above **enum** example, but no warning for the **typedef** example – so be careful!

#### 4.1.7 The **bool** Type

**bool** is an integral type in C++ with literals **true** (1) and **false** (0). Frequently in mathematical expressions we require a *boolean* data type which consists entirely of the two values *logical-true* (1) and *logical-false* (0). In C++ the value 1 is used to represent logical-true and 0 to represent logical-false. In fact, any non-zero value can be used to represent logical-true, but the constant 1 is generally used.

Before the **bool** type was introduced to the C++ language, we frequently saw a declaration of the following form in C++ programs:

```
enum Boolean { FALSE, TRUE };
```

which makes **Boolean** a distinct user-defined type with literals (constants values) **TRUE** and **FALSE**. However, C++ now has an integral type called **bool** with literals **true** and **false**.

Thus it is now possible to define an appropriately descriptive variable, `boolean`, of type `bool`, which is useful for logical tests:

```
bool boolean ;
//...
if (boolean) // condition is true
{
    // do something
}
else           // condition is false
{
    // do something else
}
//...
```

or to use `bool` as the return type of a function:

```
//...
bool Function ()
{
//...
}
//...
if (Function ()) // condition is true
{
    // do something
//...
}
```

The following program examines type `bool`:

```
// bool.cpp
// illustrates bool

#include <iostream.h> // C++ I/O

void main ()
{
    bool b1 ;           // uninitialised
    bool b2 = false ;
    bool b3 (true) ;
    int i1 = true ;   // true to int
    int i2 = false ;   // false to int
    int i3 = b3;       // bool to int
    int i4 = 10 ;
    bool b4 = i4 ;    // int to bool
    bool b5 = -i4 ;   // int to bool

    // error: can't assign to true-false literals
    true = 2 ;
    false = -1 ;
```

```
cout << "b1: " << b1 << endl ;
cout << "b2: " << b2 << endl ;
cout << "b3: " << b3 << endl ;
cout << "i1: " << i1 << endl ;
cout << "i2: " << i2 << endl ;
cout << "i3: " << i3 << endl ;
cout << "b4: " << b4 << endl ;
cout << "b5: " << b5 << endl ;
}
```

with output:

```
b1: 43
b2: 0
b3: 1
i1: 1
i2: 0
i3: 0
b4: 1
b5: 1
```

As with the other integral data types, an uninitialised **bool** variable is undefined. The program illustrates that it is possible to convert between types **int** and **bool**. A zero **int** value sets **false** to zero, whereas any non-zero **int** value sets **true** to one. Because the **true** and **false** literals are rvalues<sup>2</sup> they cannot have their values altered by assignment.

Don't worry if the above examples appear confusing at this point – all will become clear later. For the moment, just note that the use of type **bool** is more descriptive than a similar use of type **int** for an identifier or function that we want to hold or return (respectively) one of two logical values.

## 4.2 The **short**, **long**, **unsigned**, **signed**, **const** and **volatile** Data Type Modifiers

C++ would be somewhat limited if we only had at our disposal the four fundamental data types. Therefore, C++ allows us to effectively ‘create’ new data types by the use of *modifiers* to alter the attributes of a given data type. The six modifiers are **short**, **long**, **unsigned**, **signed**, **const** and **volatile**. It is worth noting that the order of modifier and type is irrelevant. Thus **signed int** or **int signed** are equivalent, although it is conventional to precede the data type by the modifier(s), e.g. **signed int**, **signed short int** or **const int**. As mentioned above, the **short** and **long** data modifiers also tackle the problem of portability between 16-bit and 32-bit computers.

---

<sup>2</sup> The *rvalue* (right value) of a variable is the actual value of the variable which is stored at the variable's *lvalue* (left value) memory address.

### 4.2.1 The **short** Modifier

The **short** modifier modifies the **int** data type only. A **short int** variable is guaranteed to have the range -32 768 ( $-2^{15}$ ) to 32 767 ( $2^{15}-1$ ), independent of the machine.

### 4.2.2 The **long** Modifier

The **long** modifier modifies the **int** and **double** types only. A long variable is guaranteed to have the range -2 147 483 648 ( $-2^{31}$ ) to 2 147 483 647 ( $2^{31}-1$ ). The suffix L or l attached to any constant forces that constant to be represented as a **long**, e.g. 101L.

### 4.2.3 The **unsigned** Modifier

Some numbers don't need a negative value: for example, speed or a millionaire's bank balance. Thus, to avoid wasting the signed bit of such numbers C++ allows a number to have no sign or be unsigned. Thus, a 16-bit **unsigned int** or **unsigned short int** ranges from 0 to 65 535 ( $2^{16}-1$ ), whereas a 32-bit **unsigned long int** assumes a maximum value of 4 294 967 295 ( $2^{32}-1$ ). The **unsigned** keyword cannot be used with the **float** and **double** types. The suffix U or u attached to any constant forces that constant to be represented as an **unsigned**, e.g. 101U.

### 4.2.4 The **signed** Modifier

The **signed** modifier allows a programmer to indicate explicitly that a data type uses a sign bit. A **signed int** variable has the identical negative to positive range of values as the **int** type.

### 4.2.5 The **const** Modifier

There is another modifier that warrants mention here: **const**. This modifier (frequently called an *access* modifier) prevents any assignments to an object after the object has been initialised. Also, a **const** object cannot be incremented (++) or decremented (--). The following are example uses of **const**:

```
const char    C_CONST          = 'C' ;
const          C               = 0 ;           // default
                                         // int
const int     I_CONST          = 9 ;
const float   TWENTY_TWO_OVER_SEVEN = 3.1428571 ;
const double  PI              = 3.1415927 ;
```

After these definitions the following are illegal:

```
//...
C_CONST = 'A' ;
C = C + 1 ;
C++ ;
```

**Table 4.4** Range and size of data types.

Type specifier	Range	Size (bytes)
<b>char</b>	-128:127	1
<b>int</b>	-32 768:32 767	2
<b>float</b>	$\pm(3.4e-38:3.4e+38)$	4
<b>double</b>	$\pm(1.7e-308:1.7e+308)$	8
<b>wchar_t</b>	same as <b>unsigned int</b>	2
<b>enum</b>	-32 768:32 767	2
<b>bool</b>	true (1) or false (0)	2
<b>short</b>	same as <b>int</b>	2
<b>short int</b>	same as <b>int</b>	2
<b>long</b>	-2 147 483 648:2 147 483 647	4
<b>long int</b>	same as <b>long</b>	4
<b>long double</b>	$\pm(3.4e-4932:1.1e+4932)$	10
<b>unsigned</b>	same as <b>unsigned int</b>	2
<b>unsigned char</b>	0:255	1
<b>unsigned int</b>	0:65 535	2
<b>unsigned short</b>	same as <b>unsigned int</b>	2
<b>unsigned short int</b>	same as <b>unsigned int</b>	2
<b>unsigned long</b>	0:4 294 967 295	4
<b>unsigned long int</b>	same as <b>unsigned long</b>	4
<b>signed</b>	same as <b>int</b>	2
<b>signed char</b>	same as <b>char</b>	1
<b>signed int</b>	same as <b>int</b>	2
<b>signed short</b>	same as <b>int</b>	2
<b>signed short int</b>	same as <b>int</b>	2
<b>signed long</b>	same as <b>long</b>	4
<b>signed long int</b>	same as <b>long</b>	4

// ...

Since the notation adopted is that all constant identifiers are in uppercase, we know that (C=C+1) is illegal before compilation.

Table 4.4 lists the valid combinations of types. Combinations such as **short char**, **long char**, **signed double**, **unsigned double** and **signed float** are illegal. Note that the ranges and sizes given in the table are for a 16-bit system. If you are using 32-bit architecture then check your compiler's documentation. For example, **long double** is implemented as a 80-bit value (10 bytes) for the Borland C++ compiler (version 5.0) but can be a 96-bit or 128-bit value on some systems. The C++ implementation of each of the data types can be found in the header files FLOAT.H and LIMITS.H. FLOAT.H deals with floating-point types. The following is an excerpt from FLOAT.H for the Borland C++ compiler:

```
//...
#define DBL_DIG 15
#define FLT_DIG 7
#define LDBL_DIG 19
```

---

```
//...
```

which defines the number of digits of precision for **double**, **float** and **long double** respectively. The header file LIMITS.H contains implementation details for the limits of data type values, such as

```
//...
#define LONG_MAX 2147483647L
#define ULONG_MAX 4294967295UL
//...
```

for the maximum **signed** and **unsigned long** data types. Try running the program files LIMITS.CPP and FLOATS.CPP to test the implementation-specific details of the character, integer and floating-point data types for your compiler.

#### 4.2.6 The **volatile** Modifier

The **volatile** modifier, also known as an *access* modifier, is a peculiar thing and rarely used. **volatile** is necessary because the majority of commercial compilers will generally try to optimise an expression which contains a variable whose value appears to remain unaltered from one statement to the next. The compiler optimisation may take the form of using a register copy of a variable rather than reloading the variable from memory. Basically, **volatile** informs the compiler that a variable's value may be altered by mechanisms unknown to the compiler and therefore the compiler must not perform any optimisations on a **volatile** variable.

As an example of **volatile**, you may define a variable, **interrupt**, which receives data from an external device such as a serial port:

```
volatile int interrupt ;
```

The keyword **volatile** in the above statement informs the compiler that expressions involving the variable **interrupt** should not be optimised. Note that a variable can be both **const** and **volatile**.

The data type or modified data type that you choose will undoubtedly depend on the application and the precision required. Although C++ gives a programmer a large degree of flexibility in choosing a date type, always try to select the smallest data type for the job at hand. It is seen from Table 4.4 that the respective sizes in bytes of **float**, **double** and **long double** are 4, 8 and 10 respectively. As a consequence, **double** and **long double** not only consume more memory than **float**, but are also slower in arithmetic operations. However, there are occasions when the exact range of values that an identifier or object will hold are not known by the programmer during development.

Another consideration, particularly when deciding between **float** and **double**, is the order of precision. The types **float**, **double** and **long double** have 7-, 15- and 19-digit precision, respectively. If you are performing accurate numerical computational work then **double** is the best compromise between size and precision.

The following sections continue our present discussion of the data types and help recap on the material covered above.

## 4.3 Constants

C++ provides four different types of constant (also referred to as *literals*): integer, floating point, character and string, and enumeration.

### 4.3.1 Integer Constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16) in C++. The suffixes (L, l) or (U, u) can be assigned to a constant to force it to be represented as a **long** or an **unsigned**, respectively; e.g. 1U or 2l. Combinations of L, l, U and u can be used on a given constant in any one of the eight permutations: ul, uL, Ul, UL, lu, lU, Lu or LU; e.g. 3Ul, 4UL or 5lu.

#### *Decimal Constants*

A digit sequence of base 10 is a decimal. Decimal constants can vary from 0 to  $4\ 294\ 967\ 295$  ( $2^{31}-1$ ) for an **unsigned long int** data type. A decimal constant cannot begin with a zero in C++ or it will be assumed to be an octal constant. An example of a decimal constant is  $17_{10}=1\times10^1+7\times10^0$ .

#### *Octal Constants*

A digit sequence of base 8 is an octal. Octal constants in C++ begin with a zero. Note that an octal constant cannot contain the digits 8 or 9. An example of an octal constant is  $17_8=1\times8^1+7\times8^0=15_{10}$ .

#### *Hexadecimal Constants*

A digit sequence of base 16 is a hexadecimal. Hexadecimal constants in C++ begin with 0x or 0X. An example of a hexadecimal constant is  $17_{16}=1\times16^1+7\times16^0=23_{10}$ .

Table 4.5 illustrates decimal, octal and hexadecimal constants for the decimal digits 0 to 9.

### 4.3.2 Floating-Point Constants

We generally think of a floating-point number as, say, 343.6 – that is, a number with a decimal integer, a decimal point and a decimal fraction. However, in C++ we can also represent

**Table 4.5** Decimal, octal and hexadecimal constants.

Decimal	Octal	Hexadecimal
0	0	0x0
1	01	0x1
2	02	0x2
3	03	0x3
4	04	0x4
5	05	0x5
6	06	0x6
7	07	0x7
8	010	0x8
9	011	0x9

floating-point numbers in terms of a signed integer exponent or a type suffix or both; e.g. 2.998e08 which has the value  $2.998 \times 10^8$ . Thus a floating-point constant can consist of:

- decimal integer
- decimal point
- decimal fraction
- signed integer exponent (e or E)
- suffix ( F, f, L or l )

The suffix F or f forces a constant to be of type **float**, whereas the suffix L or l forces a constant to be of type **long double**, e.g. 2.51f and 2.52L. Note that combinations of both suffixes is not possible.

### 4.3.3 Character and String Constants

A character constant can be either one or two characters depending on whether the constant is defined as type **char** or **wchar\_t**. A character string constant is simply a string of characters. There are ‘special’ character constants, *escape sequences*, that are preceded by a backslash (\) such as ‘\n’ for inserting a new line.

### 4.3.4 Enumeration Constants

Enumeration constants are identifiers defined in an **enum** type declaration. For example:

```
enum TelephoneDigits
{
    ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE
};
```

which defines the identifiers ZERO, ONE, TWO, ..., NINE with the integer constants 0, 1, 2, ..., 9 respectively. There are several alternative formats of the **enum** declaration which will not be discussed here but left for further discussion until Chapter 8.

## 4.4 const or #define?

We saw above that the keyword **const** ensures that an identifier cannot be inadvertently altered throughout a program:

```
const int I_CONST = 5 ;
//...
I_CONST = 6 ; // error: attempt to modify a const
```

Constants can also be specified in C++ with the **#define** preprocessor directive:

```
#define I_DEFINE 1
//...
```

Although it is feasible to use the `#define` directive, I don't recommend it. The use of `#define` stemmed from C before `const` was introduced into the C++ language. There are several problems with `#define`, but in the definition of symbolic constants the main problem is that the type of the constant is not specified when defined. Also, with no type defined no type checking can be performed by the compiler.

## 4.5 The `sizeof` Operator

In Table 4.4 we saw the size, in bytes, of all of the available integral data types in C++. If you would like to determine the size, in bytes, of a type or an identifier, use the `sizeof` unary operator. Applied to an identifier the general form is:

`sizeof` operand

where the operand can be either a constant or a variable. Alternatively, if you would like to apply the `sizeof` operator to a type (integral or user-defined) then parentheses must be used:

`sizeof` (typeSpecifier)

For example, `sizeof (int)` would return the value 2. Note that the type specifier cannot be `enum`, `struct` or `class`.

## 4.6 Keywords

Keywords are words that form the language of C++ and have a special meaning to the compiler, such as `char`, `int` and `enum`. As a consequence, a keyword cannot be used as the name of an identifier. Some compilers, such as Borland C++, also have additional non-ANSI keywords, which are generally underscored, e.g. `_asm` and `__asm`. However, if you require ANSI-compatibility and portability then always use ANSI-specified keywords.

It is fascinating to compare the C++ language, which comprises approximately 50 keywords, with, say, the English language, which is composed of literally millions of words. Computer languages have a long way to go before they reach the sophisticated level of abstraction of traditional spoken-written languages<sup>3</sup>.

All of the keywords in C++ are listed in Appendix A.

## 4.7 Declaration or Definition?

The difference between *declaration* and *definition* is a subtle one. A declaration introduces a variable's or object's name and associates a type with the variable/object. A definition is a declaration that simultaneously allocates memory for the variable/object.

---

<sup>3</sup> The traditional written and spoken languages are often referred to as *natural languages* so as to distinguish them from *artificial* programming languages.

The following statement is in fact a definition, because it not only declares the variable's name, `x`, and type, `double`, but also allocates 8 bytes of memory for `x`:

```
double x ;
```

To help emphasise the distinction between declaration and definition, consider the `enum` `Colour` once more. The following is an `enum` declaration:

```
enum Colour
{
    RED,
    GREEN,
    BLUE
};
```

whereas the statement:

```
Colour colour ;
```

defines a variable, `colour`, of type `Colour`.

If, throughout the text, I forget the difference between declaration and definition, then please forgive me!

In the above example programs a variety of constant and variable names have been chosen. So what constitutes a valid and sensible name? Basically, C++ allows you complete control to dream up whatever names you like, except for three main exceptions:

- **keywords:** You cannot use a C++ keyword as an identifier name.
- **digits:** You cannot use a digit as the first character of an identifier name, e.g. `2i`.
- **operators:** You cannot use an operator as part of an identifier name, e.g. `c+p`.

C++ allows you up to 32 characters to define an identifier, but don't carried away dreaming up too-descriptive identifiers, which can be tedious to work with. In the following, an example of 'Hungarian notation' (`hPrevInstance`) is used to shorten an identifier and make the identifier more informative (Petzold, 1992). For example, all of the following are legal:

```
int      i2 ;
const int I_CONST = 1 ;
int      handle_to_previous_instance ; // O. K. but lengthy
                                         // name!
int      hPrevInstance ;             // manageable
                                         // identifier
```

whereas the following are illegal:

```
int      2i ;
float   f+ ;
double  at@ ;
```

For a more detailed discussion of the style that I adopt throughout this book, refer to Section 3.8.

## 4.8 Assignment

Statements such as:

```
int i ;           // definition
//...
i = 99 ;         // assignment
//...
int j = 0 ;      // definition and initialisation thro'
                  // assignment
```

assign values to variables. Assignment is performed with the aid of the assignment operator `=`.

## 4.9 Initialisation

We have seen above that variables *can be* initialised when they are defined, whereas constants *must be* initialised when they are defined:

```
char      c_var ;
const char C_CONST = 'c' ;
//...
```

It is worth noting that in C++ a variable remains undefined until its value has been explicitly set. Consider the code:

```
int i ; // defined but uninitialized!
cout << "i: " << i ;
```

When I ran this program code the integer 14 919 was output. Be careful of uninitialized variables – don't assume that they are equal to zero. If you want a variable to be initialised to zero then you will have to do so explicitly:

```
int i = 0 ; // defined and initialised to zero
//...
```

A neater and more formally correct form of initialisation, which will become clearer when we discuss classes in Chapter 9, is to use the variable's type/**class** constructor:

```
int i (0) ; // also defined and initialised to zero
//...
```

## 4.10 Expressions

An *expression* is a collection of operators and identifiers (or operands) used in a specified order to evaluate a computation. An example expression is:

---

```
z = x + y ;
```

Note the distinction between an expression and a statement. The above is in fact a statement which assigns the addition of *x* and *y* to *z*. The semicolon informs the compiler that the expression is complete and that the code to evaluate the expression can be generated. Without the semicolon the compiler would assume that the expression was incomplete.

## 4.11 Operators

C++ contains a variety of operators; see Appendix C. Operators are generally categorised in terms of *unary*, *binary* and *ternary* (the *arity* of the operator). A unary operator can be applied to a single operand; e.g. -45. A binary operator has two operands (e.g. 40 + 45), and a ternary operator has three operands. ‘A ternary operator?’, I hear you ask. C++ supports only one ternary operator (? :, the conditional operator), which is used in **if-else** statements and is examined in the next chapter.

An *operand* is an identifier or object which an operator acts on. In the expression:

```
a = b ;
```

the operator is the assignment sign and the operands are *a* and *b*. There are two operands because = is a binary operator.

### 4.11.1 Arithmetic Operators

The addition, subtraction, multiplication and division arithmetic binary operators (+, -, \* and /) form the basis of mathematical computations. There is another arithmetic binary operator, % (the *modulus* operator), which is a bit different from the others. For example,  $4 \% 3 = 1$ ,  $4 \% 4 = 0$  and  $4 \% 5 = 4$ . The modulus operator gives the remainder of operand1 divided by operand2:

```
operand1 % operand2
```

Naturally, the modulus operator is restricted to integer operands.

### 4.11.2 Chaining Operators

Operators can be *chained* or *cascaded*. Chaining is simply using an operator repeatedly in a single statement. The following example code illustrates chaining with the insertion, addition and assignment operators:

```
cout << "x = " << x << " ; y= " << y << " ; z = " << z ;
i = 2 + 3 + 4 ;
x0 = y0 = z0 = 0.0 ;
// ...
```

The last statement is equivalent to the following three statements:

```
x0 = 0.0 ;
y0 = 0.0 ;
```

```
z0 = 0.0 ;
```

Chaining can also be performed on the extraction operator (`>>`):

```
//...
cout << "enter two positive or negative numbers: " ;
cin >> number1 >> number2 ;
```

The extraction operator is chained to allow the user to enter two numbers separated by a space.

### 4.11.3 Operator Shortcuts

In computing, the following statement is frequently seen:

```
sum = sum + value ;
```

which at first appears slightly confusing and contradictory. Since the precedence of `+` is greater than `=`, the expression `sum+value` is first evaluated and then the result assigned to `sum`, thus overwriting `sum`'s previous value. Because such statements are so popular in C++, an alternative notation is offered: the *arithmetic assignment operators* (`+=`, `-=`, `*=`, `/=` and `%=`). Thus, the above can be written more concisely as:

```
sum += value ;
```

Appendix C also shows other bitwise and bit-shift assignment operators: `&=`, `|=`, `^=`, `<<=` and `>>=`, which will be covered in Chapter 5.

### 4.11.4 The Increment (++) and Decrement (--) Operators

In computing, we frequently need to increment a variable by the constant 1:

```
int count = 0 ;
//...
count = count + 1 ;
```

With our ‘new’ arithmetic assignment operator (`+=`) we could rewrite this as:

```
//...
count += 1 ;
```

But there exists a more concise representation in C++ with the aid of the *increment operator* (`++`):

```
count++ ;
```

Actually, there are two versions of the increment operator: *prefix* and *postfix*. The following is prefix:

```
++count ;
```

while

```
count++ ;
```

is postfix.

To help explain the difference between prefix and postfix let us consider the two separately, prefix first:

```
int c = 1 ;
//...
value = ++c ; // increment before assignment
//...
// value = 2, c = 2
```

where `c` is incremented before assignment. In the case of the postfix operator:

```
int c = 1 ;
//...
value = c++ ; // increment after assignment
//...
// value = 1, c = 2
```

where `c` is *incremented after assignment*.

You may be thinking that this is a subtle difference, and you'd be right. However, there are occasions where the distinction between prefix and postfix can be useful.

Similarly, we have the *decrement* operator (`--`). Therefore, for the prefix decrement operator:

```
int c = 1 ;
//...
value = --c ; // decrement before assignment
//...
// value = 0, c = 0
```

while the postfix decrement operator case gives:

```
int c = 1 ;
//...
value = c-- ; // decrement after assignment
//...
// value = 1, c = 0
```

#### 4.11.5 Operator Precedence, Associativity and Arity

Consider the following expression:

```
10 - 4 * 3 ; // = -2
```

Is this expression to be interpreted as  $(10-4)*3$  or  $10-(4*3)$ ? Who knows? However, the computer has to apply some kind of logic to a situation like this, and the system that it adopts is to associate a precedence level with each of C++'s operators. Precedence, in the present context, means that an order of processing is associated for each operator. *Sub-expressions* are

evaluated according to the following rule: *highest precedence wins*. Therefore, in the expression above the sub-expression  $(4 * 3)$  is evaluated first, followed by the sub-expression  $(10 - 12)$ . This is because the precedence level of multiplication ( $*$ ) is 13 and that of subtraction ( $-$ ) is 12, so multiplication wins. Hence, the above expression is evaluated to  $-2$ .

If you think that the above expression should equal 18, then the way around this problem is to use parentheses, ( and ), which have a higher precedence level (16) than the multiplication sign:

```
(10 - 4) * 3 ; // = 18
```

What if an expression contains operators with the same precedence? For instance, the multiplicative ( $*$ ,  $/$  and  $\%$ ) operators all have the same precedence, so how is the following expression evaluated?

```
12 / 2 * 3 ; // = 18
```

The answer lies in the *associativity* of the operators. An expression involving operators of the same precedence is evaluated according to the associativity of the operators. All of the multiplicative operators are left-associative, so that the above expression is evaluated from left to right:  $(12 / 2) * 3$ . If you are ever in doubt about the exact order of evaluation of an expression, use parentheses. Parentheses help to clarify the intended order of evaluation in an expression.

Associativity is also helpful in clarifying the order of assignment. Consider the following expression, which was introduced above:

```
a = b ;
```

Is  $b$  assigned to  $a$  or  $a$  assigned to  $b$ ? To us the answer is obvious, but the compiler has to apply the associativity rule to the assignment operator. From Appendix C we see that the assignment operator is right-associative, and therefore the right-hand operand,  $b$ , is assigned to the left-hand operand,  $a$ .

Consider the following use of the arithmetic (+) operator with the insertion ( $<<$ ) operator:

```
cout << 1 + 2 ;
```

The addition operator is evaluated first because it has a higher precedence than the insertion operator. Thus the output is 3. However, certain compilers may issue a warning in such a situation, indicating that the operators need parentheses.

The arity of an operator is simply the operator's number of operands. For example, the insertion and addition operators are both binary because they operate on two operands. C++ supports the unary, binary and ternary arities.

We shall return to precedence, associativity and arity in Chapter 10, when we discuss operator overloading, but for now Appendix C lists all of the operators in the C++ language with their associated level of precedence, associativity and arity.

## 4.12 Manipulators

Manipulators are function-like operators that enable us to change the format of a stream. Table 4.6 lists the available stream manipulators.

**Table 4.6** C++ stream manipulators.

<i>Manipulator</i>	<i>Description</i>
dec	Set decimal conversion base format flag
endl	Insert a new line and flush stream
ends	Insert a null termination in a string
flush	Flush an ostream
hex	Set hexadecimal conversion format flag
oct	Set octal conversion base format flag
<i>resetiosflags (long f)</i>	Clear the format bits specified by f
<i>setfill (int c)</i>	Set the fill character to c
<i>setiosflags (long f)</i>	Set the format bits specified by f
<i>setprecision (int n)</i>	Set the floating-point precision to n
<i>setw (int n)</i>	Set the field width to n
ws	Extract the white space characters

To be able to use the *setw()* and *setprecision()* manipulators you will have to include the IOMANIP.H header file.

The use of manipulators and other functions or objects for formatting both program input and output is essential to good programming. Good formatting of program output can make a program more readable. To illustrate this, consider the formatted stream output (with the aid of the *setw()* and *setprecision()* manipulators) for the **float**, **double** and **long double** floating-point data types:

```
// manip.cpp
// illustrates the stream manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // C++ manipulators: setw() &
setprecision()

void main ()
{
    // exponential
    const float      F_E   = 2.7182818 ;           // 7 d.p.
    const double     D_E   = 2.718281828459045 ;      // 15 d.p.
    const long double LD_E = 2.7182818284590452354 ; // 19 d.p.

    // 'straight' output
    cout << "float e: "       << F_E   << endl ;
    cout << "double e: "      << D_E   << endl ;
    cout << "long double e: " << LD_E << endl ;
    cout << endl ;

    // setw() 'manipulated' output
    cout << setw(15) << "float e: "
        << setw(7) << F_E   << endl
        << setw(15) << "double e: "
        << setw(7) << D_E   << endl
        << "long double e: "
        << setw(7) << LD_E << endl ;
    cout << endl ;
```

```
// setw(), setprecision() 'manipulated' output
cout << setw(15) << "float e: "
    << setw(21) << setprecision(8) << F_E << endl
    << setw(15) << "double e: "
    << setw(21) << setprecision(16) << D_E << endl
    << "long double e: "
    << setw(21) << setprecision(20) << LD_E << endl ;
}
```

which produces the following output:

```
float e: 2.71828
double e: 2.71828
long double e: 2.71828

        float e: 2.71828
        double e: 2.71828
long double e: 2.71828

        float e:           2.7182817
        double e:         2.718281828459045
long double e: 2.71828182845904509
```

The output is divided into three parts to illustrate the `setw()` and `setprecision()` manipulators. The first part is simply the default output provided by the output stream object `cout`. Observe that the output is unformatted and the number of decimal places is five. The second part uses `setw()` to give the text strings and the three constant numbers a field width. Finally, we use the `setprecision()` manipulator, passing the number of decimal places, to output the three constant numbers to the required levels of precision. Note that the `float` and `long double` constants are not displayed as accurately as expected!

The following sections describe several manipulators in more detail.

### 4.12.1 The `endl` Manipulator

The program above and many of the previous programs have used the `endl` manipulator with the `cout` stream object. The `endl` manipulator inserts a new line into the stream output and then *flushes* the stream.

### 4.12.2 The `flush` Manipulator

If you need to flush an output stream then use the `flush` manipulator, e.g.

```
cout << flush ;
```

### 4.12.3 The `setw()` Manipulator

Each output value has its own field width. The default field is just wide enough to hold the value or string being displayed. For instance, the string "float e:" has a field width of 8. The `setw()` manipulator allows you to specify your own field width. Clearly, `setw()` has to occur before the string.

#### 4.12.4 The *setprecision()* Manipulator

*setprecision()* allows you to specify the number of decimal places of a displayed value.

#### 4.12.5 The *setiosflags()* Manipulator

The C++ `ios` **class** provides operations that are common to both input and output. The `ios` **class** is a base **class** for the classes `istream`, `ostream`, `fstreambase` and `strstreambase`. For the moment you don't need to know the details of the `ios` **class**, only that it contains format flags such as `dec`, `oct`, `hex`, `showpoint` and `fixed`. For example, with the aid of the *scope resolution* operator (`::`) we are able to set the `showpoint` and `fixed` flags so that the output will always contain a decimal point and use a fixed (rather than exponential) decimal point for floating-point numbers:

```
cout << setiosflags (ios::showpoint)
    << setiosflags (ios::fixed)
    << setprecision (2)
    << 456.00f ;
```

Refer to your compiler's documentation for a complete list of the format flags available with the `ios` **class**.

#### 4.12.6 The *resetiosflags()* Manipulator

If any format flags have been set using the *setiosflags()* manipulator they can be unset using the *resetiosflags()* manipulator.

#### 4.12.7 The `dec`, `hex` and `oct` Manipulators

To display a number as a decimal, hexadecimal or octal number use the `dec`, `hex` and `oct` manipulators respectively. For example:

```
int i = 9 ;
//...
cout << hex << i ; // o/p integer variable i as a
                      // hexadecimal number
```

Note that the decimal base is the default.

The `dec`, `hex` and `oct` manipulators can also be used to manipulate input:

```
int number = 0 ;
//...
cout << "enter a hexadecimal integer: " ;
cin >> hex >> number ;
```

The `hex` manipulator is used to configure the input to accept a hexadecimal integer.

## 4.13 Basic File Input and Output

In the previous section we examined output and the use of manipulators to the standard output device (the display screen) using the `cout` object and the insertion operator, `<<`. Although we will examine file input and output in detail in Chapter 17, let's take a brief look at basic file streams. Consider the following program, which writes some strings of text to a file called `STRINGS.TXT`<sup>4</sup>:

```
// ofile.cpp
// output of strings to a file

#include <fstream.h> // C++ file I/O

void main ()
{
    // file object and open file
    ofstream file_out ("STRINGS.TXT") ;

    file_out << "Jack's mother said, 'We're stony broke!" << endl ;
    file_out << "'Go out and find some wealthy bloke" << endl ;
    file_out << "Who'll buy our cow. Just say she's sound" << endl ;
    file_out << "'And worth at least a hundred pound. " << endl ;
    file_out << "But don't you dare to let him know" << endl ;
    file_out << "'That she's as old as billy-o'" << endl ;

    file_out.close () ; // close file
}
```

Rather than outputting the strings of text to the display screen using `cout`, an output file stream object, `file_out`, is defined of the C++ library **class** `ofstream` (*output file stream*) and initialised to the text file `STRINGS.TXT`. Upon defining `file_out`, the file `STRINGS.TXT` is created if it does not already exist; it will be overwritten if it does already exist. Apart from operating on `file_out` rather than `cout` there is syntactically no difference in writing to a file rather than to the display screen. When we are finished writing to the output file, the file is closed by a call to the `close()` function. The peculiar looking `file_out` definition and initialisation and `close()` function call are in fact calls to **class** `ofstream`'s *constructor* and *member function* `close()` respectively. In order to define an object of **class** `ofstream`, the header `FSTREAM.H` must be `#include`d.

Consider now writing integer numbers to a file:

```
// ofile1.cpp
// output of integers to a file

#include <fstream.h> // C++ file I/O

void main ()
{
    // file object and open file
```

---

<sup>4</sup> *Jack and the Beanstalk*, Roald Dahl's Revolting Rhymes, Picture Puffins, London.

---

```

ofstream file_out ("INTS.DAT") ;

file_out << 0 << endl ;
file_out << 1 << endl ;
file_out << 2 << endl ;

file_out.close () ; // close file
}

```

After executing OFILE1.EXE the contents of the file INTS.DAT are:

```

0
1
2

```

The similarity of programs OFILE.CPP and OFILE1.CPP indicates that the process of writing to a file using an `ofstream` object and the insertion operator is independent of the type of data written to the file.

So how about reading data from a file? Let's take a look at the following program, which reads the integers in file INTS.DAT generated by OFILE1.CPP and then proceeds to display the numbers to the display screen:

```

// ifile.cpp
// get integers from a file

#include <iostream.h> // C++ file I/O

void main ()
{
    // in-file object and open file
    ifstream file_in ("INTS.DAT") ;

    int i, j, k ;

    // get from file
    file_in >> i ;
    file_in >> j ;
    file_in >> k ;

    // O/P to screen
    cout << i << endl
        << j << endl
        << k << endl ;

    file_in.close () ; // close file
}

```

with output to the display screen:

```

0
1

```

To read a file we define a file object, `file_in`, of the C++ library `class ifstream` and initialise `file_in` to the file INTS.DAT. Then, instead of the standard input stream object `cin` and the extraction operator `>>`, the input file stream object `file_in` and `>>` are used to extract the three integers from file INTS.DAT. After displaying the integers on the screen (using `cout` and `<<`), the file INTS.DAT is closed.

If the above discussion appears slightly confusing, all will become clear in Chapter 17 when we discuss file streams in more detail.

## 4.14 Type Conversion and Casting

Before discussing the storage class specifiers in the next section, let us consider automatic and user-specified type conversions. Consider what happens in the program:

```
int working_day = 8 ;
float hourly_rate = 7.70 ;

double days_pay = working_day * hourly_rate ;
cout << "days pay: f" << days_pay ;
```

An `int` is multiplied by a `float` and assigned to a `double`! This is legal, but bad practice, and a compiler will perform the type conversion automatically and not issue a warning message.

Occasionally you need to be able to perform your own type conversions. C++ offers the programmer two alternative cast styles:

```
int i = 1 ;
//...
long j = (long) i ; // C style
long k = long (i) ; // C++ style
```

which both cast, or coerce, integer `i` to type `long`. The first of the two forms, `(long)`, is the C syntax, whereas `long ()` is the generally adopted syntax in C++. If you are familiar with functions, then the syntax `long ()` may at first appear to indicate that `long` is a function. However, as you become more familiar with programming in C++ the reasons for adopting the *functional notation* `long ()` will make more sense.

Anyone who has programmed for Windows will be used to type casting to avoid error messages during compilation. For example, the Windows API (Application Program Interface) function `SendMessage ()` has the following signature:

```
LRESULT SendMessage (HWND hwnd, UINT uMsg,
                      WPARAM wParam, LPARAM lParam) ;
```

The `SendMessage ()` function sends a specified message (`uMsg`) to the given window (`hwnd`) procedure function and does not return until that window procedure has processed the message. The relevant defined types in WINDOWS.H are:

```
typedef unsigned int WORD ;
```

---

```
typedef unsigned int UINT ;
typedef signed long LONG ;
typedef UNIT WPARAM ;
typedef LONG LPARAM ;
typedef LONG LRESULT ;
//...
```

A typical use of *SendMessage()* could be:

```
//...
char szBuffer[MAX] ;
WORD n ;
//...
n = WORD (SendMessage (hwndList, LB_GETTEXT, n,
                      LONG (LPSTR(szBuffer)))) ;
//...
```

which casts the string *szBuffer* first to type LPSTR (long pointer to string) and then to a LONG and finally casts the *SendMessage()* function return type to WORD – nasty!

## 4.15 The Storage Class Specifiers **auto**, **extern**, **register** and **static**

C++ supports four<sup>5</sup> storage class specifiers, called **auto**, **extern**, **register** and **static** (each a keyword), which inform the compiler of exactly how a variable is to be stored. The following subsections discuss each specifier separately.

### 4.15.1 The **auto** Specifier

The **auto** storage class specifier explicitly declares a variable as local or *automatic*. Since local variables are declared **auto** by default, **auto** is essentially redundant and hardly ever used:

```
// auto.cpp
// illustrates the auto storage class specifier
#include <iostream.h> // C++ I/O

void main ()
{
    int i = 0 ;
    auto int ai = 0 ;

    cout << i << endl ;
    cout << ai << endl ;
}
```

---

<sup>5</sup> Discounting **mutable** which will be discussed in Chapter 9.

with output:

```
0  
0
```

#### 4.15.2 The **extern** Specifier

Computer programs generally consist of several separate different source files or modules. Consider that you are working with a given file, FILE.CPP, in which you declare a number of variables that you would like to access from another file. Declaring the variables with the **extern** specifier gives the variables *external linkage* and can be accessed from another file. Here is FILE.CPP:

```
// file.cpp  
// some variable definitions  
  
char ch = 'a' ;  
int i = 1 ;
```

If the file FILE.CPP is linked with another file, say EXTERN.CPP, then EXTERN.CPP has access to the variables *ch* and *i* if the **extern** specifier is used for each variable:

```
// extern.cpp  
// illustrates the extern storage class specifier  
#include <iostream.h> // C++ I/O  
  
extern char ch ;  
extern int i ;  
  
void main ()  
{  
    cout << "ch: " << ch << "; i: " << i << endl ;  
}
```

with output:

```
ch: a; i: 1
```

Although the variables *ch* and *i* were defined and initialised in FILE.CPP they are accessible in EXTERN.CPP with the help of the **extern** *linkage specification* declaration. It is worth noting that **extern** is a declaration and not a definition, and therefore does not allocate memory; see the previous discussion on declarations and definitions. Try removing the **extern** keyword from EXTERN.CPP and relink:

```
// extern.cpp  
//...  
char ch ;  
int i ;  
  
void main ()  
{
```

---

```
cout << "ch: " << ch << "; i: " << i << endl ;
}
```

A linker error of the form ‘variables `_ch` and `_i` defined in module FILE.CPP are duplicated in module EXTERN.CPP’ will be produced. This error occurs because in a C++ program a variable can be defined only once but may be declared any number of times:

```
int i = 1 ;
//...
int i ; // error: multiple definition for i
```

Alternatively, if variables `ch` or `i` had been declared **extern** but simultaneously initialised, then the **extern** specifier would be ignored because an initialised declaration is a definition:

```
// extern.cpp
//...
extern char ch = 'b' ;
extern int i = 2 ;

void main ()
{
    cout << "ch: " << ch << "; i: " << i << endl ;
}
```

This will again result in a linker error because of the multiple definitions of `ch` and `i`.

The **extern** specifier is also applicable to functions, which are discussed in Chapter 6.

### 4.15.3 The **register** Specifier

The **register** specifier is a request to the compiler that a variable be allocated a *register* of the processor or cache memory. Thus, **register** variables are accessed more quickly than non-register variables. The keyword **register** is only a request, and the compiler has the last say on whether or not a variable is allocated a register. If a register is not available, a local **auto** variable is allocated. The **register** specifier can be applied to any variable type.

The following program illustrates **register**:

```
// register.cpp
// illustrates the register storage class specifier
#include <iostream.h> // C++ I/O

void main ()
{
    register char ch = 'a' ;
    register int i = 0 ;
    register double d = 0.0 ;

    cout << "ch: " << ch
        << "; i: " << i
        << "; d: " << d << endl ;
}
```

with output:

```
ch: a; i: 0; d: 0
```

The **register** specifier is best reserved for variables that are used frequently. A good case for using a **register** variable is as the control variable in a loop:

```
for (register int i=0; i<MAX; i++)
{
//...
}
```

The **for**-loop is discussed in the next chapter.

#### 4.15.4 The **static** Specifier

The **extern** specifier, discussed above, illustrated external linkage. The **static** specifier declares a variable with *internal linkage*; that is, local to a source file. To illustrate **static**, let us modify the file FILE.CPP, presented above, to include **static** definitions of variables ch and i:

```
// file1.cpp
// some static variable definitions

static char ch = 'a' ;
static int i = 1 ;
```

noting that **static** variables can be simultaneously defined and initialised, just as non-**static** variables can. A second file, STATIC.CPP, also contains **static** definitions of **char** and **int** variables with the same names ch and i:

```
// static.cpp
// illustrates the static storage class specifier
#include <iostream.h> // C++ I/O

static char ch = 'b' ;
static int i = 2 ;

void main ()
{
cout << "ch: " << ch << "; i: " << i << endl ;
```

On linking and executing the two files we have the following output:

```
ch: b; i: 2
```

Although both files define ch and i the program will link correctly because the **static** specifier ensures that each file has its own ch and i, therefore eliminating name clashes and multiple definitions.

The **static** specifier in connection with functions will be discussed in Chapter 6.

## 4.16 The **asm** Declaration

The C++ language allows assembly language instructions to be placed in a C++ program rather than being assembled in separate assembly modules. The keyword **asm** is used to inform the compiler of assembly instructions. Although **asm** is implementation-dependent, compilers generally support the following **asm** syntax:

```
asm instruction instruction_operands ;

asm instruction instruction_operands newline

asm
{
    // multiple instructions
}
```

The first **asm** statement contains an assembly language instruction followed by possible operand(s) for the instruction and terminated with a semicolon. The second form is identical to the first except that the end of an **asm** statement is terminated by a new line rather than a semicolon. The final **asm** declaration includes multiple **asm** instructions between the opening, {, and closing, }, braces.

A discussion of assembly language programming is beyond the scope of this book, but to illustrate the three types of **asm** declaration discussed above consider the following program:

```
// asm.cpp
// illustrates the asm keyword
#include <iostream.h> // C++ I/O

void main ()
{
    int i = 0 ;

    // call to interrupt procedure
    asm INT 0x05 ;

    // move registers
    asm MOV i, 1

    // push and pop
    asm
    {
        PUSH AX ;
        PUSH CX ;
        //...
        POP AX ;
        POP CX ;
    }
}
```

## 4.17 Summary

It is important to have a very good understanding of the fundamental data types, simply because they are so fundamental to C++ programming. This chapter has introduced you to the four primitive types: **char**, **int**, **float** and **double**. Variations of these exist through the use of the modifiers **short**, **long**, **unsigned**, **signed**, **const** and **volatile**. Also, C++ possesses three other data types that have been introduced: **wchar\_t**, **enum** and **bool**.

Standard input and output has been introduced through the use of the stream objects **cin** and **cout** and the operators **>>** and **<<**, and input received from the keyboard and output sent to the display screen. The manipulation of files has been briefly introduced by defining file stream objects of the C++ library classes **ofstream** and **ifstream**. Input and output can be formatted by the use of *manipulators*, which operate on the stream objects.

The C and C++ syntax styles of *casting* have been discussed for converting an object of one data type to an object of another data type. Casting will be discussed in much more detail in Chapter 16 when the C++ *run-time type information* language feature is covered.

The four storage class specifiers **auto**, **extern**, **register** and **static** were introduced. The **auto** specifier explicitly declares a variable as local, but is rarely used because variables are local by default. The **extern** specifier attaches external linkage to a variable, whereas **static** attaches internal linkage to a variable. A register can be allocated to a variable using the **register** specifier so that a frequently used variable can be accessed more quickly.

The **asm** keyword allows assembly language code to be integrated directly into C++ code.

We have also been introduced to the basics of programming in C++ and program construction. Different types of operators and expressions have been examined for the four primitive types. If there are any points that you don't fully understand in this chapter, give them a second visit.

## Exercises

- 4.1 Run the LIMITS.CPP and FLOATS.CPP program files to test your compiler's implementation of the integral type ranges.
- 4.2 Write a program that gets four characters from a user via the keyboard using the **cin** input stream object and the extraction operator, **>>**.
- 4.3 Write a program that reads in values entered by a user from the keyboard for the coefficients  $a$ ,  $b$  and  $c$  of the quadratic equation  $ax^2+bx+c=0$  and then proceeds to find the two roots of this general equation, which are given by:

$$x = \frac{-b \pm (b^2 - 4ac)^{1/2}}{2a}$$

Test your program with several values for  $a$ ,  $b$  and  $c$  for which the discriminant ( $b^2 - 4ac$ ) remains positive.

**Hint:** Include the header file MATH.H in your program to enable you to use the square root function:

```
double sqrt (double) ; // #include <math.h>
```

- 4.4 Display the following numbers to the display screen ensuring that the formatting and precision are preserved:

```
3.14159  
2.71828  
1.41421  
1.293  
9.998e+02  
11370.00
```

Use the `setw()`, `setprecision()` and `setiosflags()` manipulators.

**4.5** What are the results of the following expressions?

```
12 * 4 + 3 * 2  
6 * 4 / 2 % 2
```

**4.6** A file called EXP.DAT contains three integer numbers:

```
142  
857  
285
```

Write a program which reads in the above three numbers and displays the average to the display screen.

**4.7** Cast a floating-point number to an integer number.

# Making Decisions and Repetition

*Second Thoughts. Don't have them. Switch over.*  
Gerard Gilbert, TV Guide 23/9/94, *The Independent*

*Programming is mainly concerned with getting a computer program to demonstrate a little intelligence or to make decisions. The basis of a decision in computing lies ultimately in testing whether an expression is true or false. This chapter introduces the C++ tools available for making decisions: the if, if-else and switch statements and the conditional operator, ?:, which is a specialised form of the if-else statement.*

*The great power of computers is their ability to perform a task over and over again (namely repetition) with phenomenal speed, accuracy and reliability. Repetition is something that we humans find very boring and difficult to do. In this chapter we introduce the three facilities offered by C++ for performing repetition: the for loop, the while loop and the do-while loop.*

*The chapter concludes by examining the bitwise and shift operators.*



## 5.1 Decisions

A program frequently requires the capability to make a decision based on some *known*<sup>1</sup> information. For example, if a car's speed is below the speed limit then accelerate; otherwise, if the car's speed is above the speed limit then decelerate. Given the speed of the car and the speed limit we can make this kind of accelerate or decelerate decision. Decisions occur in a variety of disguises: *on-off, yes-no, true-false* etc.

C++ offers us two key ways of making decisions: the **if-else** and **switch** statements. We shall begin by examining the **if** statement alone, followed by the **if-else** decision statement, after which we shall discuss the **switch** statement, finishing by examining the *conditional operator* (`? :`), which is a specialised form of the **if-else** statement.

---

<sup>1</sup> Note the emphasis placed on known information. We can always make a purely random or a *blind* decision, but we shall not be concerned with such decisions.

### 5.1.1 The **if** Statement

Consider the program:

```
// if.cpp
// illustrates the if statement
#include <iostream.h> // C++ I/O

void main ()
{
    float number ;

    // get user-entered number
    cout << "enter a positive or negative number: " ;
    cin  >> number ;

    // compare number to zero
    if (number > 0)
        cout << number << " is greater than zero" << endl ;
}
```

The following shows a typical program–user interaction:

```
enter a positive or negative number: 10.12
10.12 is greater than zero
```

Upon execution the program extracts a number from the user and then proceeds to test whether the number is greater than zero. The keyword **if** directly followed by the required expression in parentheses (`number>0`) performs the comparison. The general syntax of the **if** statement is:

```
if (test expression)
    statement ; // single statement if-body
```

Figure 5.1 illustrates the **if** statement schematically. Think of the **if** statement as a box in which we have only an input and an output. If the input ‘passes the test’ then the box *fires* and the box produces an action. Alternatively, if the box ‘fails the test’ then the box produces no output.

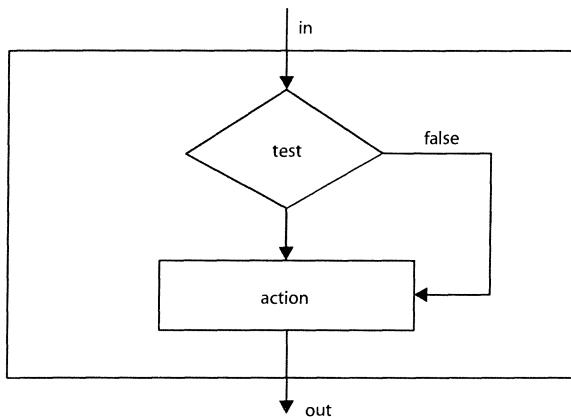
Try running the above program but enter a negative number. What happens? Nothing. The program is so simple that it can only test whether the number entered is greater than zero.

OK: let’s make the above program a bit more intelligent:

```
// ifif.cpp
// illustrates the if statement
#include <iostream.h> // C++ I/O

void main ()
{
    float number ;

    // get user-entered number
```

**Fig. 5.1** **if** statement.

```

cout << "enter a positive or negative number: " ;
cin  >> number ;

// compare number to zero
if (number > 0)
    cout << number << " is greater than zero" << endl ;

if (number < 0)
    cout << number << " is less than zero" << endl ;

if (number == 0)
    cout << number << " is equal to zero" << endl ;
}
  
```

This program simply adds another **if** statement which tests whether the number entered is less than zero. Actually, a third **if** statement is also added, which tests whether the number is in fact equal to zero<sup>2</sup>. This has been performed to give the program *closure*. A program without closure (in the present case, the possibility of a user entering zero) is prone to bugs. Thinking of special closure cases such as these to make a program more watertight is the bane of programming.

Using multiple **ifs** works fine, but there is a more elegant way: the **if-else** statement.

### 5.1.2 The **if-else** Statement

If we examine each of the above three test expressions in the **if** statements, we observe that in each case the test expression has to be true for the single statement in the **if** body to be executed. However, in making decisions we are frequently interested in performing a true-false comparison rather than a series of true conditions. The **if-else** statement is designed for just such a comparison. The following program illustrates the **if-else** version:

```
// if_else.cpp
```

<sup>2</sup> Is zero positive or negative or both?

```
// illustrates the if-else statement
#include <iostream.h> // C++ I/O

void main ()
{
    float number ;

    // get user-entered number
    cout << "enter a positive or negative number: " ;
    cin >> number ;

    // compare number to zero
    if (number > 0)
        cout << number << " is greater than zero" << endl ;

    else if (number < 0)
        cout << number << " is less than zero" << endl ;

    else
        cout << number << " is equal to zero" << endl ;
}
```

Note that if the program did not have to test whether the entered number is equal to zero we could simply use an **if-else** statement rather than an **if-else if-else**:

```
//...
if (number > 0) // test if true
    cout << number << " is greater than zero" << endl ;
else           // else false
    cout << number << " is less than zero" << endl ;
}
```

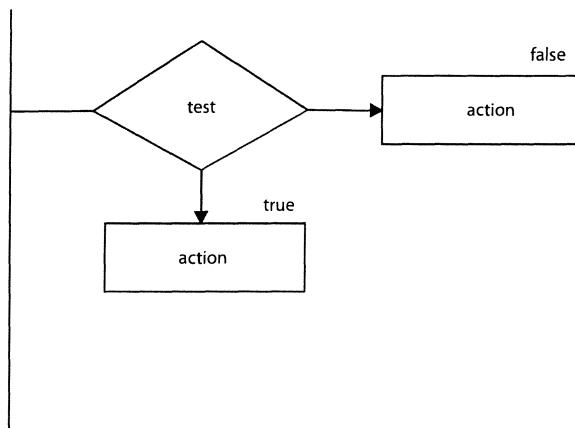
The general syntax of the **if-else** selection statement is:

```
if (test expression)
    statement ;
else if (test expression)
    statement ;
else           // none of the above
    statement ;
```

Figure 5.2 illustrates the operation of the **if-else** statement. If the test is either true or false then a respective action is generated. The flow through a multiple **if-else** statement is shown in Figure 5.3.

To date, we have only used **if** and **if-else** statements with a single statement. If we require *multiple* or *compound* statements within an **if** or **if-else** block then we have to enclose (or delimit) the block of statements by a pair of braces, { and }:

```
// if_comp.cpp
// illustrates the if-else compound statements
#include <iostream.h> // C++ I/O
```

**Fig.5.2 if-else statement.**

```

void main ()
{
float number ;

// get user-entered number
cout << "enter a positive or negative number: " ;
cin >> number ;

// compare number to zero
if (number > 0)
{
    cout << number << " is greater than zero" << endl ;
    cout << "try again entering a negative number" << endl ;
}
else if (number < 0)
{
    cout << number << " is less than zero" << endl ;
    cout << "try again entering a positive number" << endl ;
}
else
{
    cout << number << " is equal to zero" << endl ;
    cout << "why don't you enter a nonzero number?" << endl ;
}
}

```

The general syntax of the **if-else** statement with compound body statements is:

```

if (test expression)
{
    statement1 ; // compound statements if-body
    statement2 ;
}

```

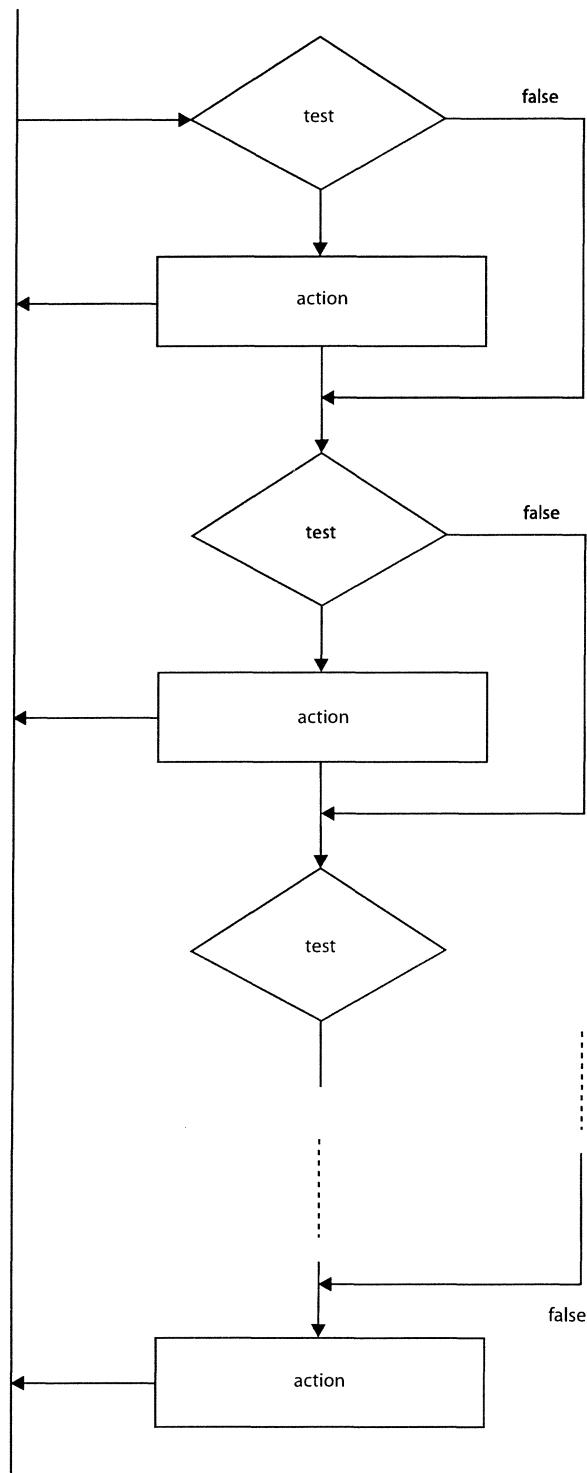


Fig. 5.3 Multiple `if-else` selection.

```

//...
statementn ;
}
else if (test expression)
{
statement1 ;
statement2 ;
//...
}
else
{
statement1 ;
statement2 ;
//...
}

```

Note the style adopted of indenting the compound statement delimiters by one tab stop. This style makes it easier to read the compound statements. Several programmers adopt the following style:

```

if (test expression) {
    statements ;
    //...
}

```

which, although it occupies one line fewer than the previous style, in my view does not help to clarify the **if** statement body.

A more useful illustration of the compound **if-else** statement is the following program:

```

// if_manip.cpp
// illustrates the if-else statement and the
// dec, hex and oct stream manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // dec, hex, oct

void main ()
{
    char ch      = 'x' ;
    int  number = 0 ;

    cout << "enter a format: 'd' decimal,
            'h' hexadecimal, 'o' octal: " ;
    cin  >> ch ;

    if (ch == 'd')
    {
        cout << "enter a decimal integer: " ;
        cin  >> dec >> number ;
        cout << "hexadecimal: " << hex << number << endl
            << "octal       : " << oct << number << endl ;
    }
}

```

```

else if (ch == 'h')
{
    cout << "enter a hexadecimal integer: " ;
    cin >> hex >> number ;
    cout << "decimal: " << dec << number << endl
        << "octal : " << oct << number << endl ;
}
else
{
    cout << "enter an octal integer: " ;
    cin >> oct >> number ;
    cout << "decimal : " << dec << number << endl
        << "hexadecimal: " << hex << number << endl ;
}
}

```

The above program prompts the user to enter one of three characters: d, h or o, which respectively inform the program whether the user is entering an integer in decimal, hexadecimal or octal base. The **if-else** statement then directs program flow to prompt the user to enter an integer in the requested base, and then displays the integer entered in two alternative bases. Note that both the input and output are manipulated using the `dec`, `hex` and `oct` stream manipulators; refer to Chapter 4. An example of some user interaction is:

```

enter a format: 'd' decimal, 'h' hexadecimal, 'o' octal: h
enter a hexadecimal integer: ffa
decimal: 4090
octal : 7772

```

### 5.1.3 Nested ifs

Nested **ifs** are **if** statements within **if** statements within.... The following program illustrates nested **ifs**:

```

// if_nest.cpp
// illustrates nested ifs
#include <iostream.h> // C++ I/O

void main ()
{
    float number ;

    // get user-entered number
    cout << "enter a positive or negative number: " ;
    cin >> number ;

    // compare number to zero
    if (number > 0)
    {
        cout << number << " is greater than zero" << endl ;
        if (number < 100) // nested if
        {

```

```

        cout << "0 < " << number << " < 100" ;
    }
}
else
{
    cout << number << " is less than zero" << endl ;
    if (number > -100) // nested if
    {
        cout << "-100 < " << number << " < 0" << endl ;
    }
}
}
}

```

You must be getting familiar with this number program by now! This version uses nested **if**s to test whether the number entered is within the range ( $0 < \text{number} < 100$ ) if the number is positive or in the range ( $-100 < \text{number} < 0$ ) if the entered number is negative. Note that it wasn't actually necessary to use the delimiters { and } for the nested **if**s, since both **if**-bodies are single statements, but the delimiters increase the clarity of the program.

Similarly, nested **if-else** statements are allowed:

```

if (number > 0)
{
//...
}
else
{
if /*...*/
{
//...
}
else
{
if /*...*/
{
//...
}
//...
}
//...
}

```

### *Precedence of Scope*

Before we examine relational operators, let us take a quick look at the *scope* or *visibility* of variables defined within different levels of **if** statements:

```

void main ()
{
int i = 10 ;
//...
if /*...*/ // outer if

```

```

{
int i = 20 ;
//...
if /*...*/ // nested if
{
    int i = 30 ;
    //...
}
//...
}
//...
}

```

The scope of a variable within an **if** statement is local to the statement block in which it is defined. In other words, the variable dies or is not visible from outside the statement block. The above program illustrates three variables defined with the same name, *i*. In each **if** statement block the variable with the most current scope takes precedence.

Chapter 6 examines *local* and *global* scope more closely for function definitions.

### 5.1.4 Relational Operators

Relational operators are used to test the *relationship* between two expressions. In the programs above we have seen the three relational operators less than (<), greater than (>) and equal to (==). Table 5.1 lists the relational operators in C++.

Comparing these operators with the operators in Appendix C we observe that the relational operators are all binary operators and left-associative. Strictly speaking, the equality and inequality operators are referred to as *equality* operators, but for most purposes it suffices to consider them as relational operators. In addition, we see from Appendix C that the precedence of the == and != operators is one less than the other four. A general expression that uses the relational operators is of the form:

expression1 relational\_operator expression2

The result of the comparison will be either logical-true or logical-false. In C++ logical-false is indicated by 0, whereas logical-true can be any non-zero value (e.g. 1, 21 or -11). The following **if** statement is therefore always true:

```

if (-11)
    cout << "this if-body statement will always be executed!" ;
//...

```

**Table 5.1** Relational operators.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Although any non-zero value indicates a logical-true, it is general practice to use the constant 1.

In the example program IFCPP above we saw the following use of the greater than (>) relational operator:

```
if (number > 0)
// ...
```

Also note that the equal to (==) operator uses two = signs. Don't make the mistake of using a single = sign (assignment operator) in a relational operation:

```
if (number = 0) // don't!
// ...
```

This expression means that number is assigned the value zero and therefore the conditional test will always equate to logical false. This is easy to do and is not a compilation error, although the Borland C++ (version 5.0) compiler issues a 'possibly incorrect assignment' warning message.

There is a strange way around this problem in C++, although I don't recommend using it. Simply flip round the expression:

```
if (0 == number) // O.K. but weird!
```

This works – try it. If you now omit one of the = signs then a compilation error is generated:

```
if (0 = number) // error: lvalue required
```

The reason why a compilation error is issued is because a constant does not have an *lvalue* or a memory address, so the compiler has nowhere it can store the variable number.

### 5.1.5 Logical Operators

Logical operators allow us to combine boolean (true–false) values logically. There are three logical operators provided by C++, which are listed in Table 5.2.

A general expression that uses the binary logical operators && and || is of the form:

```
expression1 logical-operator expression2
```

The logical operators && and || are both binary operators and left-associative. The logical operator ! is a unary operator and right-associative and has the general form:

```
! expression
```

Let's now examine each of the three logical operators in turn.

**Table 5.2** Logical operators.

Operator	Description
&&	logical AND
	logical OR
!	logical NOT

### **Logical AND Operator (&&)**

The logical AND operator (`&&`) is used to test whether two expressions are logically true or false. Consider the following program that prompts a user for two numbers (yes two!):

```
// log_and.cpp
// illustrates the logical AND operator (&&)
#include <iostream.h> // C++ I/O

void main ()
{
    float number1, number2 ;

    // get user-entered numbers
    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    // compare both numbers to zero using logical AND (&&)
    if (number1 > 0 && number2 > 0)
    {
        cout << number1 << " and " << number2
            << " are greater than zero" << endl ;
    }
    else
    {
        cout << number1 << " and " << number2
            << " are not greater than zero" << endl ;
    }
}
```

The **if** statement is now of the form:

```
if (number1 > 0 && number2 > 0)
```

which will be logically true only if *both* `number1` and `number2` are greater than zero. Only if both expressions for the logical AND operator are true will the **if**-body statement be displayed. Since we are now testing two expressions, each of which can be true or false, we have four possible permutations. These four permutations are usually presented in the form of a *truth table*; see Table 5.3.

Table 5.3 illustrates that only when both expressions are logical-true is the logical result true. The other three permutations each produce a logical-false result.

You may be wondering why the **if** statement wasn't written as:

**Table 5.3** Logical AND truth table.

<i>expression1</i>	<i>expression2</i>	<i>Logical result</i>
false	false	false
false	true	false
true	false	false
true	true	true

---

```
if (number1 && number2 > 0)
```

Try it and see what happens. If you enter the two numbers 10 and 11 then the string ‘10 and 11 are greater than zero’ is correctly displayed. However, if you enter the two numbers -12 and 13 then the string ‘-12 and 13 are greater than zero’ is incorrectly displayed. This form of condition test is not testing number1 but only testing number2. In addition, the **if**-body statement is executed because in C++ any non-zero (positive or negative) number is logical-true. So, be careful of this common error!

Also, why don’t we need parentheses around the two separate expressions to distinguish between the logical (**&&**) and relational (**>**) operators? For example:

```
if ((number1 > 0) && (number2 > 0))
```

You’ve probably guessed that the answer lies with the precedence of the operators. A quick check with Appendix C shows that the precedence of the relational operator (**>**) is greater than the precedence of the logical operator (**&&**). Thus the relational comparisons are performed before the logical comparison. Although in the present example the use of the parentheses is redundant, do use parentheses if you feel that they add to the clarity of a program.

### *Logical OR Operator (| |)*

The logical OR operator (**| |**) is used to test whether *either* of two expressions is logically true; if so, the result is logical-true. Logical OR is a binary operator and is left-associative. The truth table for the logical OR operator is shown in Table 5.4.

Note that there are three permutations of logical-true for the logical OR operator. If either or both of the two expressions are true then the result is true.

Consider the numbers program again:

```
// log_or.cpp
// illustrates the logical OR operator (| |)
#include <iostream.h> // C++ I/O

void main ()
{
    float number1, number2 ;

    // get user-entered numbers
    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    // compare both numbers to zero using logical OR (| |)
    if (number1 > 0 | | number2 > 0)
    {
```

**Table 5.4** Logical OR truth table.

expression1	expression2	Logical result
false	false	false
false	true	true
true	false	true
true	true	true

```

        cout << number1 << " or " << number2
        << " are greater than zero" << endl ;
    }
else // note: else-executed only if both numbers are
      // negative!
{
    cout << number1 << " and " << number2
    << " are not greater than zero" << endl ;
}
}

```

Since the **if**-body statement will be executed if either or both of the two numbers entered is positive, we know that if the **else**-body statement is executed then both numbers are negative – that's logic for you.

### *Logical NOT Operator (!)*

The logical NOT operator (!) reverses the logical value of its operand. Thus a true expression is reversed to false and a false expression is reversed to true. The logical NOT operator is a unary operator and is right-associative. Consider the program:

```

// log_not.cpp
// illustrates the logical NOT operator (!)
#include <iostream.h> // C++ I/O

void main ()
{
    float number ;

    // get user-entered number
    cout << "enter zero or a positive/negative number: " ;
    cin >> number ;

    if (!number) // if number=0 (false) then !number !=0 (true)
    {
        cout << number << " is zero" << endl ;
    }
    else
    {
        cout << number << " is nonzero" << endl ;
    }
}

```

Admittedly, we could simply use **if**(number) rather than **if**(!number):

```

//...
if (number)
{
    cout << number << " is nonzero" << endl ;
//...
}

```

### 5.1.6 The **switch** Statement

The **switch** statement is a convenient way of dealing with a large, complex series of similar decisions in a program.

The general syntax of the **switch** statement is:

```
switch (test expression)
{
    case constant_expression1:
        // statements ;
        // ...
    case constant_expressionn:
        // statements ;
}
```

The expression is typically of type **int**, **char**, **long** or **unsigned**, and cannot be of a floating-point type such as **float** or **double**. The body of the **switch** statement is delimited by a pair of braces. Within the **switch** statement flow is controlled by the keyword **case**. The keyword **case** is directly followed by a constant expression, which is in turn followed by a colon (:). Let us consider an example which prompts a user to enter an integer number between 0 and 2, inclusive, and displays a message according to the number entered:

```
// switch.cpp
// illustrates the switch statement
#include <iostream.h> // C++ I/O

void main ()
{
    int number ;

    cout << "enter an integer number in the range (0:2): " ;
    cin >> number ;

    switch (number)
    {
        case 0:
            cout << "number entered: zero" << endl ;
        case 1:
            cout << "number entered: one" << endl ;
        case 2:
            cout << "number entered: two" << endl ;
    }
}
```

Some typical user interaction, entering the number 2, is:

```
enter an integer number in the range (0:2): 2
number entered: two
```

That works fine. Now try entering the number 0:

```
enter an integer number in the range (0:2): 0
number entered: zero
number entered: one
number entered: two
```

When the number 0 is entered we see that the program *falls through* the remaining **case** statements. Thus, we need a way out of the **switch** statement when we have finished executing the statement(s) associated with a particular **case**. Fortunately, C++ provides the keyword **break** to solve this problem. The **break** keyword terminates a **case** statement(s) block after the last statement. Upon exiting the **switch** statement control is directed to the first statement after the closing brace of the **switch** statement. Consider the same program as above, but now modified by the keyword **break**:

```
// b_switch.cpp
// illustrates the switch statement and the keyword break
#include <iostream.h> // C++ I/O

void main ()
{
    int number ;

    cout << "enter an integer number in the range (0:2): " ;
    cin  >> number ;

    switch (number)
    {
        case 0:
            cout << "number entered: zero"   << endl ;
            break ;
        case 1:
            cout << "number entered: one"    << endl ;
            break ;
        case 2:
            cout << "number entered: two"    << endl ;
            // note: no break required here!
    }
}
```

Note that no **break** for **case** 2 is necessary since **case** 2 is the last **case** statement of the **switch** statement.

There are occasions when you may require your program to fall through the remaining **case** statements, but generally programs **break** out after the **case** statement.

What happens in the above program if a user enters a number other than 0, 1 or 2? Nothing. The **switch** statement at present does not cater for entered numbers that are outside the range (0:2). C++ provides another keyword, **default**, that will catch all of the unexpected user input. The following program uses **default** to catch exceptions:

```
// default.cpp
// illustrates the switch statement, break and default
```

```
#include <iostream.h> // C++ I/O

void main ()
{
    int number ;

    cout << "enter an integer number in the range (0:2): " ;
    cin >> number ;

    switch (number)
    {
        case 0:
            cout << "number entered: zero" << endl ;
            break ;
        case 1:
            cout << "number entered: one" << endl ;
            break ;
        case 2:
            cout << "number entered: two" << endl ;
            break ;
        default:
            cout << "enter either 0, 1 or 2: " << endl ;
    }
}
```

There are a few points to note about **default**. Notice that no expression or value is attributed to **default**. The **default** keyword can appear anywhere within the **switch** statement but can appear only once. Naturally, **default** is usually placed at the bottom of the **switch** statement. A good reason for placing **default** at the bottom of the **switch** statement is that no **break** statement is required. If you place **default** elsewhere, then a **break** will be required to prevent fall-through.

If it is required that a variable is defined local to a **case** statement block then { and } braces are required:

```
switch (number)
{
//...
case 2:
{
    int i = 0 ; // variable local to case
//...
}
//...
}
```

Finally, if you have programmed for Windows in C, using a non-object-oriented approach, then you will probably have noticed that the majority of Windows programs use a **switch-case** statement construction to process the window procedure messages:

```
long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
/*...*/)
```

```
{  
//...  
switch (message)  
{  
    case WM_CREATE:  
        // process WM_CREATE message  
        return 0 ;  
    case WM_PAINT:  
        // process WM_PAINT message  
        return 0 ;  
    case WM_KEYDOWN:  
        switch (wParam)  
        {  
            case VK_HOME:  
                // process VK_HOME key  
                break ;  
            //...  
        }  
    //...  
    case WM_DESTROY:  
        // process WM_DESTROY message  
        return 0 ;  
    }  
    return DefWindowProc (hwnd, message, //...  
}
```

Notice the **switch** within a **switch** or *nested switch* statements. Windows provides over 200 messages, and the **switch** statement for a large Windows application can be unmanageable.

### 5.1.7 The Conditional Operator (? :)

The following typical **if-else** statement:

```
//...  
if (a > b)  
    // statements ;  
else  
    // other statements ;
```

is so popular in C++ that a *conditional* operator is available which is more concise. The conditional operator is a ternary operator and is left-associative. The conditional operator is of the general syntax:

```
expression1 ? expression2 : expression3
```

The second expression (expression2) is evaluated if expression1 is logical-true. Otherwise, if expression1 is logical-false then the third expression (expression3) is evaluated.

The following program demonstrates the conditional operator:

```

// cond.cpp
// illustrates the conditional operator (? :)
#include <iostream.h> // C++ I/O

void main ()
{
    float n1, n2 ;

    cout << "enter two positive or negative numbers: " ;
    cin >> n1 >> n2 ;

    // if-else
    cout << endl << "if-else: " ;
    if (n1 > n2)
        cout << n1 << ">" << n2 ;
    else
        cout << n1 << "<" << n2 ;

    // conditional operator
    cout << endl << "conditional: " ;
    n1 > n2 ? cout << n1 << ">" << n2
              : cout << n1 << "<" << n2 ;
}

```

## 5.2 Repetition

The general technique for performing repetition in C++ is through the use of a loop. C++ offers us three alternatives for constructing a loop: **for**, **while** and **do-while**. We shall consider each of these separately:

### 5.2.1 The **for**-Loop

Of the three alternative forms of loop, the **for**-loop is by far the most frequently used. The general syntax of the **for**-loop is:

```

for (expression1; expression2; expression3)
{
    // statement body
}

```

There are three parts to a **for**-loop:

- initialisation      initialises a variable that controls the **for**-loop
- test                determines whether another iteration should be performed
- control             controls the **for**-loop variable

Each of the three separate parts is contained within the pair of parentheses ( and ). The **for**-loop body begins with the opening brace, {, and ends with the closing brace, }. Let's consider an example:

```

// for.cpp
// illustrates the for-loop
#include <iostream.h> // C++ I/O

void main ()
{
    double a, x_old, x_new ;

    cout << "enter const. (0<a<=4) and initial (0<x<1) values: " ;
    cin >> a >> x_old ;

    cout << endl << x_old << endl ;

    for (int i=0; i<10; i++)
    {
        x_new = a * x_old * (1.0 - x_old) ;
        x_old = x_new ;
        cout << x_new << endl ;
    }
}

```

This program prompts a user for a constant  $a$  and an initial value  $x_0$  of a variable  $x$ . The initial  $x$  value is first displayed and then the **for**-loop of the program computes the following iteration 10 times:

$$x_{n+1} = ax_n(1-x_n)$$

Each new iteration ( $x_{\text{new}}$ ) uses the previous  $x$  value ( $x_{\text{old}}$ ) determined by the formula. The following shows the output for  $a=2.65$  and  $x_0=0.8$ :

```

enter const. (0<a<=4) and initial (0<x<1) values: 2.65 0.8

0.8
0.424
0.647194
0.605085
0.633236
0.615457
0.627174
0.619641
0.624568
0.621379
0.623458

```

and if enough iterations were performed then  $x$  would converge to  $0.622\ 642^3$ .

---

<sup>3</sup>  $x_{n+1} = ax_n(1-x_n)$  is often referred to as the *quadratic* or *logistic* law and models a simple feedback system that can have very complicated behaviour. Try entering the values  $a=4.0$  and  $x_0=0.8$ . The behaviour of  $x$  becomes *chaotic* and convergence will not occur. For an excellent discussion of this logistic law and chaos in general refer to Peitgen *et al.* (1992).

The above program illustrates that the **for**-loop performs a fixed number of iterations determined by the test expression:

```
for (int i=0; i<10; i++)
{
//...
}
```

Note that I don't adopt the end of statement style of placing a space between the first two expressions and the semicolons and a space between the operators and operands:

```
for (int i = 0 ; i < 10 ; i++)
{
//...
}
```

The former style is adopted purely for reasons of compactness. Choose a style that suits you.

Let us examine each of the three parts of the **for**-loop separately

### **for** Loop Initialisation

The first expression statement in the **for**-loop initialises the **for**-loop variable *i*:

```
int i=0;
```

The variable is initialised to zero when the **for**-loop is entered. Note that in C++ we are able to define a variable when we require it, so we don't have to use the following inelegant C-style **for**-loop:

```
int i ;
for (i=0; i<10; i++)
{
//...
}
```

The variable *i* can be initialised to any value that you require and is initialised only once, when the **for**-loop begins.

Although the definition and initialisation of *i* is within the **for**-loop, the scope of *i* is not limited to the **for**-loop. These means that the following is an error:

```
for (int i=0; i<10; i++)
{
//...
}
//...
for (int i=0; i<20; i++) // error: multiple definition for i!
{
//...
}
```

because the variable *i* has been defined more than once within the same block. The solution to this problem is simply to call the second **for**-loop variable something different, e.g. *j*, *k* or

1. In a future version of C++ it is expected that the scope of the **for**-loop variable will be restricted to the scope of the **for**-loop statement block.

Generally, the **for**-loop has only one variable, but it is possible to have multiple initialisations:

```
for (int i=0, int j=9; i<10; i++)
{
//...
}
//...
```

Each variable (*i* and *j*) is separated by the comma operator (, ).

### **for-Loop Test**

The second expression statement in the **for**-loop is the test expression:

```
i<10;
```

This expression determines the number of times the **for**-loop statement body will be executed. From the previous section, *i* is initialised to 0, and as long as *i* remains less than 10 the test expression is true and the **for**-loop statement body will continue to be processed. In the present case the test expression involves the less than relational operator (<). Relational operators are generally used for the test expression. When *i* equals 10 the **for**-loop terminates because the test expression is now false. Since the first and last values of *i* are 0 and 9 respectively, the **for**-loop body is executed 10 times.

Always ensure that the test expression can be logical-false; otherwise the **for**-loop will iterate endlessly. For example, a common error is to use the assignment operator:

```
for (int i=0; i=10; i++)
{
//...
}
```

When the value of *i* is incremented to 10, the test expression remains logical true and the **for**-loop will iterate forever. We shall take a look at *forever-loops* later.

### **for Loop Control**

The third and last expression in the **for**-loop is:

```
i++
```

This expression is used to increment the variable *i*. Remember that the C++ increment operator (++) offers a more concise syntax of incrementing a variable *i* by the constant 1 than *i=i+1* or *i+=1*. It was noted above that multiple variable initialisations are allowed in a **for**-loop. Similarly, multiple control expressions are allowed:

```
for (int i=0, int j=0; i<10; i++, j++)
{
//...
```

---

```
}
```

Again, each control expression is separated by the comma operator. While multiple expressions are syntactically correct and increase conciseness, they simultaneously decrease readability.

The test expression is not limited to increments of 1. C++ allows us to use any incrementing or decrementing step size. The following illustrates two other possible **for**-loops:

```
for (int i=0; i<10; i=i+2) // incrementing step size of 2
{
//...
}
for (int j=10; j>0; j--) // decrementing step size of 1
{
//...
}
```

Also, it is worth noting that function return values can be integrated with **for**-loop initialisation and test expressions. If two functions exist, called *StartValue()* and *NumberOfIterations()*, which return the initialisation value and the maximum number of iterations, respectively, then the following **for**-loop construction is possible:

```
int StartValue ()
{
//...
return start_value ;
}

int NumberOfIterations ()
{
//...
return number_of_iterations ;
}
//...
for (int i=StartValue (); i<NumberOfIterations (); i++)
{
//...
}
```

## 5.2.2 The **while**-Loop

The general syntax of the **while**-loop is:

```
while (test expression)
{
// statement body
}
```

Provided the test expression is logical-true then the loop will continue to iterate. The difference between the **while**-loop and the **for**-loop is that the **while**-loop can be used when you do not know in advance how many iterations of the loop are required. The following program illustrates this point by prompting a user to type in some text:

```

// while.cpp
// illustrates the while-loop
#include <iostream.h> // C++ I/O
#include <conio.h> // getche()

void main ()
{
    char ch = 'x' ; // ensure ch != '\r' (RETURN)

    cout << "type a line of text, press RETURN to
stop:" << endl ;

    int count = 0 ;
    while (ch != '\r')
    {
        ch = getche () ; // get and echo character

        if (ch == 'a' || ch == 'e' || ch == 'i'
            || ch == 'o' || ch == 'u')
        {
            count++ ;
        }
    }
    cout << endl << endl << "no. of vowels entered: " << count ;
}

```

At the time of writing the program we do not know how much or how little text a user will type in. The **while**-loop is ideal for this situation if we terminate entering the text by a carriage return. Each character that is typed in is tested against the carriage return escape sequence character '\r'. If the user has pressed carriage return then the program terminates; otherwise, the program will continue indefinitely. Note that although there is no initialisation within the loop statement, as with the **for**-loop, initialisation still has to be performed, and before the loop begins. Thus, the **while**-loop is not as concise as the **for**-loop, and this is its main disadvantage.

The **char** variable ch is initialised to an arbitrary character (in the present case 'x') to ensure that ch is not equal to carriage return at the start of program execution. Within the **while**-loop a character is obtained from the user and echoed to the screen by the function **getche()**. Each character is tested against the five vowels, and if a vowel is entered by the user a vowel count is incremented. Finally, the total number of vowels entered is displayed. For example:

```

type in a line of text, press RETURN to stop:
triangles and tetrahedra are tricky to teach

no. of vowels entered: 14

```

The **getche()** function gets a character read from the keyboard and echoes it to the text screen using the BIOS or direct video. The header file CONIO.H must be included to use **getche()**. The signature of **getche()** is:

```
int getche () ;
```

### 5.2.3 The **do-while** Loop

The general syntax of the **do-while** loop is:

```
do
{
    // statement body
} while (test expression) ;
```

Note the semicolon following the **do-while** loop test expression – this is easily forgotten. Essentially, the difference between the **while** and **do-while** loops is that the **do-while** loop executes the loop body statements before the test expression is evaluated. If the test expression is logical-true, then the statement body is executed again, and so on until a logical-false value is evaluated by the test expression. This is contrary to the **while**-loop, which evaluates the test expression before the statement body. The **do-while** loop is particularly useful when you need to execute a set of statements at least once.

The following program modifies the FOR.CPP program presented above. The **do-while** loop enables the **for**-loop iteration to be performed at least once even if a user enters values for *a* and *x* that are outside the specified range:

```
// do_while.cpp
// illustrates the do-while loop
#include <iostream.h> // C++ I/O

void main ()
{
    double a, x_old, x_new, xo ;

    do
    {
        cout << "enter const. (0<a<=4) and initial \
                  (0<x<1) values " << endl ;
        cout << "program terminates if (a) or (x) \
                  outside range: " ;
        cin >> a >> xo ;
        x_old = xo ;

        cout << endl << x_old << endl ;
        // iterate for 10 iterations
        for (int i=0; i<10; i++)
        {
            x_new = a * x_old * (1.0 - x_old) ;
            x_old = x_new ;
            cout << x_new << endl ;
        }
    } while (a>0 && a<=4 && xo>0 && xo<1) ;
}
```

This program is far from ideal, since it continues to perform the **for**-loop iteration even if a user enters values for *a* and *x* that are outside the specified range. To modify the above program

to prevent this would require the use of an **if** statement and the *exit()* or *abort()* C++ library functions, which are discussed later.

## 5.3 Forever Loops

An example was presented above where a **for**-loop could iterate indefinitely. This was accidentally achieved by incorrectly using the assignment operator in the test expression instead of a relational operator. A better and more intentional method of achieving a *forever loop* is to omit the three expressions altogether:

```
for ( ; ; )      // forever for-loop
{
    //... // for-loop body statements executed indefinitely
}
```

The null program statements are legal in C++.

To achieve similar infinite **while** and **do-while** loops we simply use a non-zero test condition:

```
//...
while (1) // infinite while-loop
{
    //...
}
//...
do          // infinite do-while-loop
{
    //...
} while (1)
```

## 5.4 Nested Loops

A nested loop is a loop within a loop. The following is an example of a **for**-loop within a **for**-loop:

```
for (int i=0; i<MAX_I; i++)
{
    //...
    for (int j=0; j<MAX_J; j++) // nested for-loop
    {
        //...
    }
}
```

Variable **i** controls the outer **for**-loop, while variable **j** controls the inner **for**-loop. Nested **for**-loops are mostly used with arrays, to be discussed in Chapter 7.

## 5.5 The **break** Statement

When we examined the **switch** statement we also saw the **break** statement, which allowed us to exit from a **case** statement (preventing fall-through) and ultimately to exit from the **switch** statement. The **break** statement can also be used to exit from a loop to give premature termination; that is before the loop test condition is logical-false. Consider the program:

```
// b_for.cpp
// illustrates the for-loop and break statement
#include <iostream.h> // C++ I/O
#include <stdlib.h> // rand(), srand(), exit()
#include <time.h> // time(), time_t

void main ()
{
    int n_iterations = 0 ;
    cout << "enter the number of iterations (1:100): " ;
    cin >> n_iterations ;

    // test number of iterations entered
    if (n_iterations <= 0 || n_iterations > 100)
    {
        cout << "a non-valid number of iterations!" ;
        exit (EXIT_SUCCESS) ; // quit execution
    }

    // initialise random number generator
    time_t t ;
    srand (unsigned (time(&t))) ;

    // for-loop to display random numbers
    for (int i=0; i<n_iterations; i++)
    {
        int random_number = rand () ;
        cout << random_number << endl ;

        if (random_number > 25000)
        {
            cout << "stop: last random number > 25,000" ;
            break ;
        }
    }
}
```

Some user interaction is:

```
enter the number of iterations (1:100): 25
291
5446
```

```
22902  
7252  
19967  
7115  
31344  
stop: last random number > 25,000
```

This program generates random numbers and displays them on the screen. If a random number is greater than 25 000 the program terminates as a result of the **break** statement. The **break** statement directs program control to the first statement immediately *outside the loop* that contains the **break** statement. Note that control is not directly sent outside the **if** statement, as you may have expected, but outside the loop. This is an important distinction and is in contradiction to using **break** and the **switch** statement that we saw previously. If you recall, a **break** within a **switch** statement sends control to the first statement directly after the **switch** statement's closing brace. Thus, in the **for**-loop above the **break** statement sends control to the closing brace of the **for**-loop, upon which we have reached the end of the program.

The random numbers are generated using two functions:

```
int rand () ;  
void srand (unsigned seed) ;
```

which require the inclusion of the STDLIB.H header file.

*rand()* generates a pseudo-random number in the range 0 to RAND\_MAX (defined in STDLIB.H) and returns the number as type **int**. *srand()* initialises the random number generator. The random number generator is reinitialised by passing as an argument to *srand()* the function *time()*:

```
time_t time (time_t *timer) ;
```

*time()* returns the time of day in seconds elapsed since 00:00:00 Greenwich Mean Time (GMT) on 1 January 1970. The TIME.H header file has to be #included in order to use *time()*.

If we don't reinitialise the random number generator each time the program is executed, the same series of random numbers will be generated.

Note also the test performed on *n\_iterations* in the program to ensure that the number of iterations entered by the user is in the specified range 1 to 100. This is necessary to prevent a user from entering a negative number, zero or a large number that will take all day to iterate through:

```
//...  
// test number of iterations entered  
if (n_iterations <= 0 || n_iterations > 100)  
{  
    cout << "a non-valid number of iterations!" ;  
    exit (EXIT_SUCCESS) ; // quit execution  
}  
//...
```

Note the use of the relational operators ( $\leq$  and  $>$ ) and the logical OR operator. The function *exit()* terminates program execution immediately. For console-based programs *exit()* is

the recommended way of initiating program shutdown. The header file STDLIB.H must be #included for `exit()`, whose signature is:

```
void exit (int status) ;
```

Before termination all open files are closed and any buffered output is written. The program above passes the EXIT\_SUCCESS status identifier, which indicates normal program termination, to the operating system. Abnormal program termination is signalled by the status identifier EXIT\_FAILURE. Refer to your compiler's documentation for a more detailed description of the `exit()` and `atexit()` functions.

There is another termination function, called `abort()`, which is similar to `exit()`:

```
void abort () ;
```

The signature indicates that `abort()` is not passed any arguments and does not return a status value to the operating system. The function `abort()` does not close open files; it simply terminates program execution. `abort()` requires the inclusion of the STDLIB.H header file.

The **break** statement can be used with **for**, **while** or **do-while** loops. For example:

```
//...
count = 0 ;
while (count < MAX_COUNT)
{
//...
if (number > MAX_NUMBER)
{
//...
break ; // break while loop
}
//...
count++ ;}
```

## 5.6 The **continue** Statement

We noted above that the **break** statement, when used in conjunction with a loop, directs control to the first statement directly after the closing brace of the loop. Sometimes this is not what we want. C++ gives us the keyword **continue**, which allows you to continue with the next iteration in the loop but to jump over any remaining code in the loop. Control is transferred to the control expression in a **for**-loop and to the test condition in **while** and **do-while** loops. An example program that illustrates the **continue** statement in a **for**-loop is:

```
// continue.cpp
// illustrates the continue statement
#include <iostream.h> // C++ I/O
#include <math.h> // sqrt()

void main ()
{
```

```
double number = 0.0 ;
for (int i=0; i<3; i++)
{
    cout << "enter a number: " ;
    cin  >> number ;

    if (number < 0)
    {
        cout << "number < 0" << endl << endl ;
        continue ;
    }
    cout << "sqrt(" << number << ") : "
        << sqrt (number) << endl << endl ;
}
}
```

A typical output is:

```
enter a number: 2
sqrt(2): 1.41421

enter a number: -1
number < 0

enter a number: 10
sqrt(10): 3.16228
```

This program prompts the user to enter a number three times. If the user enters zero or a positive number the program simply displays the square root of the number entered. If the number entered is negative the **if** condition is true and a message ‘number < 0’ is displayed and the **continue** statement is executed. The **continue** statement directs control to the third expression (*i*++) in the **for**-loop rather than displaying the **if**-body statement and then continuing to evaluate the square root. Thus, the **continue** statement offers a way of jumping over the following loop body statements but continuing with the next loop iteration. Try commenting out the **continue** statement and entering a negative number. A run-time error of the form ‘sqrt: domain error’ or ‘floating-point error’ will be issued.

## 5.7 The **goto** Statement

There is another method of creating a loop that I didn’t bother to mention above: the **goto** statement. This statement is so confusing (because of the potential it generates for writing code that jumps from one place to another indiscriminately) that if you can forget I ever mentioned it then please do. If not, then read on. The general syntax of the **goto** statement is:

```
goto label ;
//...
label:
```

The **goto** statement consists of the keyword **goto** followed by a label. The label can occur before or after the **goto** statement. The label name is followed by a colon. For example:

```
//...
start:
cout << "enter a positive number: " ;

float n ;
cin >> n ;
if (n < 0)
{
goto start ;
}
```

This example is analogous to a **while**-loop, in that if a user persists in entering a negative number the **if**-condition will be logical-true and the **goto** statement will transfer control back to the **goto** label (`start`) until the **if**-condition is logical-false.

## 5.8 Loop Body Visibility

Before we bring this chapter to a close by discussing the bitwise and shift operators, let us discuss the visibility of objects within a loop body. Constants, variables or objects that are defined within the loop body (between the opening, `{`, and closing, `}`, braces) have *local scope*, which limits their availability to the loop body in which they are defined. In other words, such objects are not *visible* from outside the loop body. For example:

```
for (int i=0; i<10; i++)
{
    double mean ;
    //...
}
//...
cout << "mean of numbers: " << mean ; // error: 'mean'
                                            // undefined
```

The notion of local scope is intuitive – a program would be chaos without it!

Scope and visibility will be discussed further when we introduce functions, structures and classes.

## 5.9 The Bitwise and Shift Operators

In addition to the operators discussed above, C++ also supports a set of operators called the *bitwise* and *shift* operators, which allow a programmer to manipulate the bits of a variable; see Table 5.5. These operators were not discussed above because the bitwise AND and OR operators are similar to the logical AND and OR operators, and if you are new to C++'s operators then the logical and bitwise operators are best discussed separately.

**Table 5.5** The bitwise and shift operators.

Operator	Description
<b>Bitwise</b>	
&	AND
	OR
^	Exclusive OR (XOR)
~	One's complement
<b>Shift</b>	
<<	Shift left
>>	Shift right

Note that the bitwise and shift operators can only be used on C++ integral data types.

Before we discuss the bitwise operators, let us first have a brief reminder of binary numbers. Decimal numbers are in base 10, meaning that the digits 0 to 9 are used to represent a decimal number. Binary numbers are in base 2, which means that only the digits 0 and 1 are used to represent a binary number.

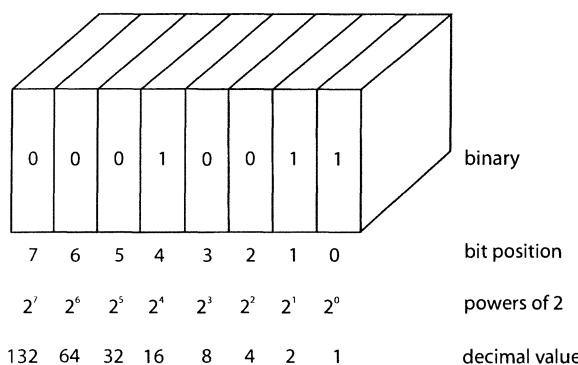
Figure 5.4 illustrates the eight-bit binary representation of the decimal number 19. Shown also in Fig. 5.4 are the bit positions ranging from 0 to 7, with their corresponding powers of two and decimal values. The high bit is position 7 and the low bit is position 0. A value of 1 turns on a bit, whereas a value of 0 turns off a bit. The decimal equivalent of a bit string is the sum of the decimal values of all turned-on bits. As shown in Fig. 5.4, the decimal equivalent of 00010011 is  $2^4 + 2^1 + 2^0$ , or 19.

### 5.9.1 Bitwise AND Operator (&)

The bitwise AND operator (&) is a binary operator and has the following general syntax:

```
operand1 & operand2
```

If the same bit position for *both* operands is on, the resultant bit is on, else if either or both of the same bit positions for both operands is off the resultant bit is off. Therefore, the truth table for bitwise AND is identical to that of logical AND (Table 5.3), except that the bitwise AND operator operates on bits rather than expressions. The bitwise AND truth table is shown in Table 5.6.



**Fig. 5.4** Eight-bit representation of a decimal number. The decimal equivalent of 00010011 =  $2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19$ .

**Table 5.6** Bitwise AND truth table.

<i>operand1</i>	<i>operand2</i>	<i>Bitwise result</i>
0	0	0
0	1	0
1	0	0
1	1	1

As an example, consider the bitwise AND operator operating on the decimal numbers 19 and 37:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	1	0	0	1	1	19
0	0	1	0	0	1	0	1	37
0	0	0	0	0	0	0	1	19&37

Only bit position 0 is turned on as a result of the bitwise AND operator, since that is the only bit position in which both operand bits are turned on. The result of 19&37 is therefore equal to the decimal value of 1. This example is illustrated in the following program:

```
// bit_and.cpp
// illustrates the bitwise AND operator (&)
#include <iostream.h> // C++ I/O

void main ()
{
    int i (19), j (37) ;

    cout << "i&j: " << (i & j) << endl ;
}
```

with output:

i&j: 1

A popular illustration of the bitwise AND operator is determining whether an integer number is odd or even. If a number is odd the zero position bit is turned on, else if a number is even the zero position bit is turned off. By combining this property of odd and even numbers with a number which is known to be odd, say 1, we have a method of determining whether a number is odd or even:

```
// even_odd.cpp
// illustrates the bitwise AND operator (&)
// by determining whether a number is odd or even
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0) ;
```

```

cout << "enter an integer number: " ;
cin  >> number ;
cout << number << " is " ;

number & 1 ? cout << "odd" : cout << "even" ;

cout << endl ;
}

```

With some user interaction:

```

enter an integer number: 2
2 is even

```

Because the logical AND (`&&`) and the bitwise AND (`&`) differ only by one `&`, always ensure that you are using the correct operator.

### 5.9.2 Bitwise OR Operator (`|`)

The bitwise OR operator (`|`) is a binary operator and has the following general syntax:

```
operand1 | operand2
```

If the same bit position for *either* operands is on the resultant bit is on, else if both of the same bit positions for both operands are off the resultant bit is off. Therefore, the truth table for bitwise OR is identical to that of logical OR (Table 5.4), except that the bitwise OR operator operates on bits rather than expressions. The bitwise OR truth table is shown in Table 5.7.

As before, consider the bitwise OR operator operating on the decimal numbers 19 and 37:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	1	0	0	1	1	19
0	0	1	0	0	1	0	1	37
0	0	1	1	0	1	1	1	19 37

The result of 19|37 is therefore equal to the decimal value of 55. This example is illustrated in the following program:

```

// bit_or.cpp
// illustrates the bitwise OR operator ()
#include <iostream.h> // C++ I/O

```

**Table 5.7** Bitwise OR truth table.

<i>operand1</i>	<i>operand2</i>	<i>Bitwise result</i>
0	0	0
0	1	1
1	0	1
1	1	1

```
void main ()
{
    int i (19), j (37) ;

    cout << "i|j: " << (i | j) << endl ;
}
```

with output:

```
i|j: 55
```

### 5.9.3 Bitwise Exclusive OR Operator (^)

The bitwise exclusive OR (XOR) operator (^) is a binary operator and has the following general syntax:

```
operand1 ^ operand2
```

The operation of XOR is similar to that of OR, in that if the same bit position for either operands is on the resultant bit is on, except that if both bit positions are turned on the resultant bit is off. The bitwise XOR truth table is shown in Table 5.8.

Sticking with the numbers 19 and 37, we have for XOR:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	1	0	0	1	1	19
0	0	1	0	0	1	0	1	37
0	0	1	1	0	1	1	0	$19 \wedge 37$

The result of  $19 \wedge 37$  is therefore equal to the decimal value of 54. This example is illustrated in the following program:

```
// bit_xor.cpp
// illustrates the bitwise XOR operator (^)
#include <iostream.h> // C++ I/O

void main ()
{
    int i (19), j (37) ;

    cout << "i^j: " << (i ^ j) << endl ;
}
```

**Table 5.8** Bitwise XOR truth table.

operand1	operand2	Bitwise result
0	0	0
0	1	1
1	0	1
1	1	0

with output:

```
i^j: 54
```

#### 5.9.4 Bitwise One's Complement Operator (~)

The bitwise one's complement operator (~) is a unary operator and has the following general syntax:

```
~ operand
```

The one's complement operator reverses the bits of its operand. Thus, turned-on bits are turned off and vice versa. Consider the one's complement operator operating on the 8-bit number 19:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	1	0	0	1	1	19
1	1	1	0	1	1	0	0	$\sim 19$

The result of  $\sim 19$  is therefore equal to the decimal value of 236.

To illustrate the one's complement operator let's take a look at the two's complement of a number, which simply adds 1 to the one's complement of a number. The two's complement is frequently used to obtain the negative of a number:

```
// bit_one.cpp
// illustrates the bitwise one's complement operator (~)
// by finding the two's complement of a number
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0) ;

    cout << "enter an integer number: " ;
    cin  >> number ;

    cout << "two's complement of number: "
        << (~number + 1) << endl ;
}
```

With some user interaction:

```
enter an integer number: 19
two's complement of number: -19
```

#### 5.9.5 The Shift Operators

The left (<<) and right (>>) shift operators are binary operators and have the following general syntax:

```
operand << number_of_bits_to_shift
```

---

```
operand >> number_of_bits_to_shift
```

operand is the variable to be shifted number\_of\_bits\_to\_shift bit positions to the left or right, respectively.

To illustrate the left and right shift operators, let them operate on the number 19 with one bit shift position:

$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	1	0	0	1	1	19
0	0	1	0	0	1	1	0	$19 \ll 1$
0	0	0	0	1	0	0	1	$19 \gg 1$

As a bit is shifted off either the high or low bit positions, a 0 is automatically placed on the opposite end of the bit string. This example is illustrated in the following program:

```
// shift.cpp
// illustrates the left and right shift operators << and >>
#include <iostream.h> // C++ I/O

void main ()
{
    int i (19) ;

    cout << "i<<1: " << (i << 1) << endl
        << "i>>1: " << (i >> 1) << endl ;
}
```

with output:

```
i<<1: 38
i>>1: 9
```

Try not to confuse the left and right shift operators with the insertion and extraction stream operators.

## 5.9.6 The Bitwise and Shift Assignment Operators

Chapter 4 illustrated the arithmetic assignment operators for writing concise expressions. Similarly, both the bitwise and shift operators have assignment versions, i.e.  $\&=$ ,  $|=$ ,  $^=$ ,  $<<=$  and  $>>=$ . These are demonstrated in the following program:

```
// bs_assgn.cpp
// illustrates the bitwise and shift assignment operators
#include <iostream.h> // C++ I/O

void main ()
{
    int i (19), j (37), k(73), l (145), m (289) ;

    // bitwise assignment
```

```

k &= j ;
cout << "k&=i: " << k << endl ;
l |= j ;
cout << "l|=i: " << l << endl ;
m ^= j ;
cout << "m^=i: " << m << endl ;

// shift assignment
i <<= 1 ;
cout << "i<<1: " << i << endl ;
i >>= 1 ;
cout << "i>>1: " << i << endl ;
}

```

with output:

```

k&i: 1
l|i: 181
m^i: 260
i<<1: 38
i>>1: 19

```

## 5.10 Summary

The relational operators (`<`, `>`, `<=`, `>=`, `==` and `!=`) allow us to compare two expressions to see if they are equal, not equal, one is greater than the other and so on. The logical result of the relational comparison is either true or false. The logical operators AND (`&&`) and OR (`||`) operate on two expressions to test whether they are logically true or false. The logical NOT operator (`!`) reverses an expression's logical value from true to false or from false to true.

The `if` and `if-else` statements form the basis of decision-making in C++. The relational and logical operators give the `if-else` statement great flexibility. The `switch` statement provides a simple tree structure for dealing with a large number of decisions. With the help of the additional keywords `break` and `default` the `switch` statement is very powerful. The keyword `break` prevents fall-through and sends control to the statement directly after the `switch` statement's closing parenthesis, while `default` enables a `switch` statement to deal with unexpected exceptions. The conditional operator (`? :`) provides us with a concise form of the `if-else` statement.

This chapter has also introduced three loops for performing repetition: `for`-loop, `while`-loop and `do-while`-loop. The `for`-loop is undoubtedly the most frequently used of the three loops. The `for`-loop's popularity lies in the fact that the initialisation, test and control expressions are all in one place. The `while`-loop must initialise its control variable before the loop commences and the `do-while`-loop similarly initialises its control variable before the loop starts but evaluates the test expression at the end of the loop. Thus, the `while` and `do-while`-loops are used when the test condition arises within the loop. The `for`-loop executes a predetermined number of iterations, whereas the `while` and `do-while`-loops are generally used to execute an unknown number of iterations, with the `while`-loop not necessarily executing at all and the `do-while` loop always executing at least once. The use of the `break` statement within a loop directs control to the closing brace of the innermost loop,

whereas the **continue** statement sends control to the start of the loop for the next iteration. The **goto** statement redirects program control to a specified label.

The chapter concluded by discussing the bitwise and shift operators. Although not as frequently used as the relational and logical operators, the bitwise and shift operators are an essential part of the C++ language because they allow a programmer to manipulate the individual bits of a variable of an integral type. The bitwise operators are AND (&), OR (|), XOR (^) and one's complement (~). The bitwise AND (&) and OR (|) are not to be confused with the logical AND (&&) and OR (||). The left and right shift operators, << and >>, shift bits by a specified number of bit positions left and right respectively.

## Exercises

5.1 Debug the following program:

```
#include <iostream.h> // C++ I/O

void main ()
{
    for (int i=0; i<10; i++)
    {
        if (i = 3)
            cout << "i=3" << endl ;
    }
}
```

5.2 Write a program that prompts the user to enter a number which can be positive or negative and displays on the screen the absolute value of the number entered.

5.3 Using a loop of your choice, write a program that displays the first 10 terms of the Fibonacci series:

$$f(x_n, x_{n+1}) = x_n + x_{n+1}, \quad n = 1, 2, 3$$

5.4 Write a program that writes the ASCII character set to a file with the equivalent decimal, hexadecimal and octal values for each character. Refer to Chapter 4 for basic file input and output.

5.5 Using a loop of your choice, write a program that calculates  $\pi$  using Gregory's series:

$$\tan^{-1} x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots$$

If we set  $x=1$  in the series then:

$$\tan^{-1}(1) = \frac{\pi}{4}$$

Note the slow convergence of this series.

**Hint:** Use the `pow()` function, which calculates  $x$  to the power of  $y$ :

```
double pow (double x, double y) ; // #include <math.h>
```

- 5.6 Write a program (using a `switch` statement) that converts three numbers specifying an HSV (Hue–Saturation–Value) colour to three numbers specifying an RGB (Red–Green–Blue) colour.

**Hint:** The logic of the program is shown in the following pseudocode:

```
double h, s, v, r, g, b ;
int i ;
double f, p, q, t ;

if s=0
    r = v ; g = v ; b = v ;
else
    if h=1
        h = 0 ;

    h = h / 6 ;
    i = h ;
    f = h - i ;
    p = v(1-s) ;
    q = v(1-sf) ;
    t = v(1-s(1-f)) ;

case i
    case 0:
        r = v ; g = t ; b = p ;
    case 1:
        r = q ; g = v ; b = p ;
    case 2:
        r = p ; g = v ; b = t ;
    case 3:
        r = p ; g = q ; b = v ;
    case 4:
        r = t ; g = p ; b = v ;
    case 5:
        r = v ; g = p ; b = q ;
```

which assumes that both HSV and RGB are in the range [0,1].

- 5.7 Write a program that prompts the user for an integer number and then proceeds to divide and multiply the number by 2 using the right and left shift operators.

## Functions

*If you think writing software is difficult, try rewriting software.*  
B. Meyer (1995)

A function is a collection of program statements. Functions are an essential tool for creating modularity within a program by discretising a large complex program into a series of separate units. The function helps a programmer to organise a program and at the same time to reduce program size and develop reusable code. A programmer's life without the function would be unbearable.

We begin by taking a look at the characteristics of all C++ functions, such as the function name, return type, arguments and function body. Functions are called by name and can return a value or object following the execution of the function's program statements. Information is passed to a function via the function arguments and can be passed either by value or by reference (memory location). Functions define their own scope. In other words, an object defined within a function is local to that function.

The chapter spends some time considering why functions are so important to programming and how similar functions can be arranged into declaration and implementation files so that function libraries can be built and easily reused. Programming without the extensive ANSI C and C++ library functions would be very difficult indeed. An overview of the functions supplied with all ANSI-compatible compilers is presented.

C++ allows functions to be overloaded so that the same name can be given to functions which differ according to the number of objects and/or type of arguments. Function overloading eliminates the need for multiple different function names when one will suffice. Functions can be assigned default arguments so that a function can be called without explicitly passing all of the function arguments, and this is a useful feature when it is desired that a function has a default behaviour. For small functions that are frequently called, the inline feature allows a function to be expanded inline and eliminates the need to define a macro. C++ supports external linkage with non-C++ functions so that C++ and non-C++ functions can communicate.

The chapter concludes by examining the popular geometric problems of the orientation of three points and the intersection of two lines.



## 6.1 A Look at Functions

Let's begin our discussion of functions by examining an example:

```
// func.cpp
// illustrates a C++ function
#include <iostream.h> // C++ I/O

// returns the maximum of two numbers
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}

void main ()
{
    double number1, number2 ;

    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    double largest = MyMax (number1, number2) ;

    cout << "largest of the two numbers: " << largest ;
}
```

This program contains two functions: *MyMax()* and *main()*. You should be familiar with the form of the *main()* function by now; we'll examine *main()* in more detail later. The other function, *MyMax()*, returns the maximum value of two numbers that are passed to the function. Note the combined use of the relational (*>*) and conditional (*? :*) operators to test for the largest of two numbers.

Some user interaction with the above program is:

```
enter two positive or negative numbers: 1 25
largest of the two numbers: 25
```

## 6.2 The Body of a Function

A function in C++ has the following general syntax:

```
return_type FunctionName (function_argument_list)
{
    // function body
    //...
    return return_type ;
}
```

We shall examine each part of a C++ function in more detail in the following sections.

## 6.3 Function Name

Any name can be given to a function. The name can contain the letters a to z and A to Z, an underscore ‘\_’ and the digits 0 to 9. There are two restrictions on function naming: (1) the first character must be either a letter or an underscore and (2) only the first 32 characters of the name are recognised by the compiler. These rules are the same as for naming identifiers, constants, variables, objects, classes etc.

The style that I adopt for function naming was highlighted in Chapter 3 and combines lower and upper case letters rather than being entirely in lowercase and using the underscore to separate words (which is the reserved style for identifiers):

```
NumberOfWindows () ;      // yes
number_of_windows () ;   // no
NumWin () ;              // O.K.
NoW () ;                // too cryptic!
```

Try to choose function names that are as descriptive as possible rather than as cryptic as possible and known only to you.

Also, try to avoid choosing function names that are standard C++ library functions. The following example defines a function called *sqrt()* which has an identical signature to the C++ library function declared in the header file MATH.H:

```
#include <math.h> // C++ sqrt()
//...
// user-defined square root function
double sqrt (double a)
{
    double value ;
    //...
    return value ;
}

void main ()
{
    double sq_root = sqrt (2) ; // calls user-defined sqrt()
    //..
}
```

The user-defined *sqrt()* function takes precedence over the C++ library *sqrt()* function.

In fact, if you adopt the style of naming user-defined functions using upper- and lowercase letters (e.g. *Sqrt()*) you will never accidentally override a C++ library function, because they are all named in lowercase.

## 6.4 Function Body

The function body is the program code that actually performs a function’s task. In the above program, the function body of *MyMax()* is a single statement:

```
return a > b ? a : b ;
```

The function body can be as small or as large as is necessary. Within the body of a function it is perfectly legal to define new constants, variables, objects etc., and to call other C++ or user-defined functions:

```
double AnyOldFunction ()
{
//...
const double SIZE = 100 ;
int count = 0 ;
//...
double any_old_value = AnyOtherOldFunction () ;
//...
return any_old_value ;
}
```

The start of the function body begins with the opening brace: {. The end of the function definition is marked by the closing brace: }. There is no difference between the body of a user-defined function and the `main()` function.

The Fortran programming language uses a FUNCTION header statement and an END statement to indicate the start and finish, respectively, of a function body. Similarly, in Basic, Pascal and a Windows resource script file the keywords BEGIN and END are used to delimit a function body or a block of statements. C++ adopts a more minimal approach to function definition by simply requiring the delimiting braces { and }.

## 6.5 Function Return Type

The last part of a function definition is generally the function return type. The `MyMax()` function above returns type `double`. A function can return any C++ type or user-defined type, but ensure that the function returns the correct type. For example, change the return type of `MyMax()` from `double` to `int`:

```
int MyMax (double a, double b)
{ /*...*/
//...
```

A user interaction with the program will now produce the output:

```
enter two positive or negative numbers: 23.45 67.89
largest of two numbers: 67
```

`MyMax()` has automatically cast the largest of two `double` numbers in to an `int`.

### 6.5.1 void Return Type

You can define functions that do not return a value. An example is a function that outputs certain information:

```
void PrintCircleInfo ()
```

---

```
{
cout << "circle radius: " << radius << endl ;
cout << "circle centre: " << centre << endl ;
}
```

The *PrintCircleInfo()* function takes no arguments and returns no type. The keyword **void** explicitly specifies to the compiler that the function does not return a value. This function would be called purely by name and empty parentheses:

```
PrintCircleInfo () ;
//...
```

Since the return value of *PrintCircleInfo()* is not used, why be so specific and use the keyword **void**? Using the keyword **void** prevents any incorrect use of the function, and incorrect usage will be detected by static type checking:

```
int i = PrintCircleInfo () ; // error:incorrect assignment of
                           // type void
```

## 6.5.2 Default Return Type

What if you forget or want to use the default return type of the function definition? The default function return type is of type **int**. The following function definition will return type **int**:

```
MyMax (int a, int b)
{
    return a > b ? a : b ;
}
```

It works, but it's not very informative to a user. If a function returns type **int**, then explicitly indicate so.

## 6.6 The **return** Statement

We saw above that a function that does not return a value has return type **void**. If a function returns a value of a given type, the function has to instruct the compiler the type of the value that it wants to return. This is performed via the **return** statement, which returns a *single* value whose type must be compatible with the specified return type in the function declarator. For instance:

```
// mixes type of return value (int)
// with return type specifier (double)
double Number ()
{
    int i = 10 ;
    return i ;
}
```

Most commercial compilers will not issue a warning or error compilation message for the above function definition, so be careful of making such errors.

After a **return** statement has been executed, control returns to the function caller. In the case of the function *MyMax()* in the program FUNC.CPP, the return value is copied to the variable *largest*, defined in *main()*. The variable *largest* is then used to output the largest of two numbers through the use of the *cout* stream object.

The **return** statement in the *MyMax()* function combines the relational and conditional operators into a single statement:

```
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}
```

Alternatively, for increased clarity, we could have written the function as:

```
double MyMax (double a, double b)
{
    double largest = a > b ? a : b ;
    return largest ;
}
```

The above two functions are equivalent. In the first version of *MyMax()*, a temporary variable is created by the compiler to which the value of either *a* or *b* is assigned before returning the value. In the second version of *MyMax()* we have explicitly indicated this procedure by defining the additional variable *largest*. If you find the intention of the second version of *MyMax()* clearer then use this approach. More experienced programmers tend to try to write a program as concisely as possible without a significant loss of clarity, and as a result the technique of eliminating unnecessary variables is a common practice.

It is also worth mentioning that when an identifier or object is returned by a function the compiler generates a temporary object to hold a copy of the object returned by the function. The temporary object simply goes out of scope when the object has been returned to the calling function. This creation of a temporary object when a function returns will be discussed in more detail when we discuss *copy constructors* and *pointers* in Chapters 9 and 12.

## 6.7 Function Calling

The function *MyMax()* is called in the *main()* function by the statement:

```
double largest = MyMax (number1, number2) ;
```

In the present case the function call involves assignment of the value returned by *MyMax()* to the **double** variable *largest*. The function call passes the two user-entered numbers *number1* and *number2* as arguments to the function and ends with a semicolon to indicate the end of statement.

Since *MyMax()* was designed and defined to return a value of type **double**, its return value is assigned directly to a variable. Alternatively, *MyMax()* can simply be called with no assignment:

---

```
MyMax (number1, number2) ;
```

but such a call is meaningless since we are still left wondering which of the two numbers is the largest! The compiler will not issue a warning or error message for such a call. If such a function call is what you require, then a redesign of *MyMax()* would have to be made:

```
void MyMaxPrint (double number1, double number2)
{
    double largest = number1 > number2 ? number1 : number2 ;
    cout << endl << "largest: " << largest ;
}
```

Such a function definition greatly restricts the possible applications of *MyMaxPrint()*.

Notice that the function call allows a program to do things without worrying about the details of the function body. All the function caller needs to know is the *signature* of the function. The function signature provides the necessary information to call the function correctly. In the program above the function signature of *MyMax()* is:

```
double MyMax (double a, double b) ;
```

which informs a programmer of the function name, number and type of arguments, and return type. Function libraries should be accompanied by sufficiently informative documentation to enable a user to call and apply the function in the manner that the function designer intended.

## 6.8 Function Definition

The function definition of *MyMax()* is:

```
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}
```

The first line is referred to as the *declarator*; we shall see why in the next section. After the declarator we have the function body, which comprises the function program statements. That's it – no big deal.

One point to note is that function definitions cannot be nested:

```
void Function ()
{
//...
void Function ()
{
//...
}
}
```

## 6.9 Function Declaration

In the above program, FUNC.CPP, the definition of `MyMax()` preceded the `main()` function. Try flipping these two function definitions round:

```
// func_dec.cpp
// illustrates a C++ function declaration
#include <iostream.h> // C++ I/O

// prototype (function declaration)
double MyMax (double a, double b) ;

void main ()
{
    double number1, number2 ;

    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    double largest = MyMax (number1, number2) ;

    cout << "largest of the two numbers: " << largest ;
}

// returns the maximum of two numbers
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}
```

Note the inclusion of the comment line and the *function declaration* or *prototype*:

```
// prototype (function declaration)
double MyMax (double a, double b) ;
```

Without the function declaration the compiler will issue an error message of the form ‘call to an undefined function’. This error message sums it up. In the program FUNC\_DEC.CPP a call is made to `MyMax()`, but the function has not been defined at this point. The function declaration indicates to the compiler that the function has not been defined yet, but is defined elsewhere – so go away and find it, Mr Compiler. Note that the function declaration ends with a semicolon since it is a statement.

The function declaration can be of the form above, which includes in the declaration the names of the function arguments (a and b), or alternatively, the function declaration can simply include the argument types:

```
double MyMax (double, double) ;
```

Choose the type of function declaration that suits you, but remember that it is good programming practice to include the argument identifiers in the function declaration, simply because the declaration is more informative with the argument identifiers. Also, it is a simple matter to

generate the function declaration by a *copy and paste* of the function declarator to the top of the program, provided you remember to add on a semicolon.

Whichever style you adopt, the most important thing is to ensure that the function declaration agrees with the declarator. Clearly, the return type, number of arguments, and type and argument order must be identical for definition and declarator.

## 6.10 Function Scope

We noticed above that it is possible to define identifiers within a function body. Are these identifiers accessible from outside the function body? The answer is no – they are local to the function in which they are defined. Accessibility is generally discussed in terms of *scope*. Scope simply refers to the visibility, availability and lifetime of an identifier within a program.

### 6.10.1 Local Scope

An identifier (constant, variable or object) has *local scope* if it is defined within a statement block or function body. For example, consider the program:

```
// l_scope.cpp
// illustrates local scope
#include <iostream.h> // C++ I/O

void Function ()
{
    int i = 20 ;
    cout << "Function (): i= " << i << endl ;
}

void main ()
{
    int i = 10 ;
    cout << "main (): i= " << i << endl ;

    Function () ;
}
```

with output:

```
main (): i= 10
Function (): i= 20 ;
```

The output illustrates that the defined *i* variables in *main()* and *Function()* are not the same. The variable *i* defined in *main()* has local scope that is confined to the *main()* function body and is not visible outside *main()*. The variable *i* in *Function()* is local to *Function()*, is not visible from outside *Function()* and its lifetime exists between definition and the closing brace of *Function()*'s function body. Figure 6.1 shows schematically the scope of the variables *i*.

```
//...
void Function ()
{
    int i ;
}

void main ()
{
    int i ;
}
```

**Fig. 6.1** Local scope of variables *i* defined in the functions *main()* and *Function()*.

## 6.10.2 Global Scope

Let's now take a look at global scope:

```
// g_scope.cpp
// illustrates global scope
#include <iostream.h> // C++ I/O

int i = 10 ;

void Function ()
{
    cout << "Function (): i= " << i << endl ;
}

void main ()
{
    cout << "main (): i= " << i << endl ;
    Function () ;
}
```

In the above program, variables *i* have been removed from the function body of *main()* and *Function()* and placed outside of any statement block. Variable *i* now has global scope. Figure 6.2 shows the global scope of *i*, which encompasses both *main()* and *Function()*. Thus both *main()* and *Function()* now have access to *i*.

```
int i ;
void Function ()
{
    ...
}

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

**Fig. 6.2** Global scope of variable *i* defined outside the functions *main()* and *Function()*.

### 6.10.3 Precedence of Scope

Consider the program below, which defines a variable `i` within the function body of `Function()`. Although `i` is defined globally, the definition within `Function()` does not redefine `i`. This is because when two variables are defined with the same name within the same scope, the variable with the most current scope takes the highest precedence:

```
//...
int i = 10 ;

void Function ()
{
    int i = 20 ;
    cout << "Function (): i= " << i << endl ;
}
//...
```

and thus `Function()` outputs the value 20 for `i`. Precedence of levels of scope was also noted in Chapter 5 for nested `if` statements. The rules for precedence of scope are the same for `if` and `if-else` statements, loops, and functions.

### 6.10.4 Pros and Cons of Local and Global Scope

On comparing the above two programs (L\_SCOPE.CPP and G\_SCOPE.CPP), we observe that defining the variable `i` as global eliminates the necessity of passing `i` as an argument to `Function()`. However, the main disadvantage (and a serious one) of global identifiers is that they are not private. Any function has access to a global identifier. Consider the program:

```
//...
int i = 10 ;

void Function ()
{
    i = 20 ;
    cout << "Function (): i= " << i << endl ;
}

void main ()
{
    cout << "main (): i= " << i << endl ;

    Function () ;
    cout << "main (): i= " << i << endl ;
}
```

The output of this program is:

```
main (): i= 10 ,
Function (): i= 20 ;
main (): i= 20 ;
```

which illustrates that the global variable `i` has been altered by the function `Function()`. This is the danger of global identifiers. Any function can change a global identifier. It is good programming practice to make as few global identifiers as possible because of the accessibility problem. When we introduce classes in Chapter 9, the subjects of scope and data access will be re-examined, and it will be shown that classes give us greater data protection and access than do functions.

### 6.10.5 Scope Resolution Operator

What if we require access to a global identifier instead of a local identifier from within a function body? For such cases C++ provides the scope resolution operator (`::`). As the name suggests, this operator resolves the problem of an identifier's scope. The scope resolution operator is illustrated in the following program:

```
// res_op.cpp
// illustrates the scope resolution operator ::

#include <iostream.h> // C++ I/O

int i = 10 ; // global i

void main ()
{
    int i = 20 ; // local i

    cout << "main () i: " << i << endl ;
    cout << "global i: " << ::i << endl ;
}
```

Within the `main()` function the statement:

```
::i ; // global variable i
```

accesses the global variable `i` rather than the local variable.

## 6.11 Function Arguments

Generally, arguments are passed to a function so that the function can operate. The function argument list follows directly after the function name and is enclosed within the parentheses (`(` and `)`). Each argument in the list is a type specifier (e.g. `int` or `double`) followed by an identifier name. If the function argument list contains more than one argument, the comma operator is used to separate the arguments. For example:

```
double MyMax (double a, double b)
{
    // ...
}
```

Function arguments are generally referred to as the *formal arguments*.

Occasionally you may come across a style of function definition in which the argument type is placed outside the parentheses and between the function declarator and the opening brace, {, of the function body:

```
double MyMax (a, b)
double a ; double b ;
{
```

This syntax was defined by Kernighan and Ritchie (1978) and is not the ANSI C/C++ standard for function definitions. The K&R standard of function definitions is now considered obsolete by many compilers, which will issue a compilation warning of the form ‘function definition syntax is now obsolete’.

### 6.11.1 No Argument List

If the function has no argument list (i.e. the function does not require arguments to operate correctly) then there are two ways to indicate a zero arguments list. The first is the use of empty parentheses, as we have seen with `main()`, and the second is to use the keyword **void**. Thus we can use either of the following:

```
return_type Function ()
```

or

```
return_type Function (void)
```

### 6.11.2 By Value or by Reference?

We have left the discussion of function arguments until now because the passing of arguments can be broken down into two separate categories: *passing by value* and *passing by reference*.

#### *Passing by Value*

Let’s look at a program:

```
// p_value.cpp
// illustrates the passing of arguments by value
#include <iostream.h> // C++ I/O

void Function (int i)
{
    cout << endl << "Function(): rvalue of i: " << i
        << ", lvalue of i: " << &i ;
}

void main ()
{
    int i = 10 ;
```

```

cout << endl << "main()      : rvalue of i: " << i
    << ", lvalue of i: " << &i ;

Function (i) ;
}

```

Note the use of the *address of* operator (`&`). The output of the program is:

```

main()      : rvalue of i: 10, lvalue of i: 0x470721e8
Function(): rvalue of i: 10, lvalue of i: 0x470721dc

```

When you run the program, the lvalues that you will see displayed will probably be different, depending on where the variables are stored in memory. The `main()` program calls the function `Function()`, passing the `int` constant 10 as an argument. The output illustrates that although the `i` variables in `main()` and `Function()` have the same value (rvalue) the variable `i` defined in `main()` is stored at a different memory address (lvalue) than the `i` defined in `Function()`.

When `main()` calls `Function()` the compiler places the rvalue of the `i` defined in `main()` on the stack. Since the rvalue of `i` is passed to `Function()`, a copy of `i` is passed as the argument to `Function()`, and not the variable itself. Thus, the compiler has generated a new variable (which is an exact copy of the passed value) to hold the passed value to `Function()`. Passing arguments in this manner is referred to as *passing by value*, since a copy of the argument is made. An important consequence of passing by value is that there is no way that the function can alter the original variable, because a copy of the variable was sent and the function does not have access to where the variable is stored in memory. This returns us to the previous discussion of local scope. A variable `i` defined in `main()` is a different variable from a variable `i` defined in `Function()`.

Passing by value ensures that a function cannot alter the function-caller argument in any manner. For this security we have to pay the price of allocating additional memory, since a function argument is copied. We shall shortly see that `const` reference arguments give us the best of both worlds: the ability to define an identifier as constant, thus ensuring that the function-caller argument is not altered, and the use of reference arguments, which eliminate any copying.

### ***Passing by Reference***

The above section illustrated how function arguments are passed by value, which requires a copy of the argument to be made. Alternatively, *passing by reference* passes an argument to a function by reference or memory address, and does not make a copy of the function argument. Thus, passing by reference accesses the function-caller argument.

The following program illustrates passing function arguments by reference:

```

// p_ref.cpp
// illustrates the passing of arguments by reference
#include <iostream.h> // C++ I/O

void Function (int& ir)
{
    cout << endl << "Function(): rvalue of ir: " << ir
        << ", lvalue of ir: " << &ir ;
}

```

```
// change alias ir
ir = 20 ;
}

void main ()
{
    int i = 10 ;

    cout << endl << "main()      : rvalue of i : " << i
        << ", lvalue of i : " << &i ;

    Function (i) ;

    cout << endl << "main()      : rvalue of i : " << i
        << ", lvalue of i : " << &i ;
}
```

The program output is:

```
main()      : rvalue of i : 10, lvalue of i : 0x461721e8
Function(): rvalue of ir: 10, lvalue of ir: 0x461721e8
main()      : rvalue of i : 20, lvalue of i : 0x461721e8
```

This time, the lvalues of the two variables *i* and *ir* are the same. However, when the alias *ir* is changed within *Function()*, variable *i* in *main()* is correspondingly modified.

Reference arguments are indicated by an ampersand (&) after the data type (e.g. **int&** *ir*). The ampersand indicates that *ir* is an alias for the variable passed by the function caller. Let us highlight this important point:

*A reference is an alias for an identifier*

Note that there is no difference between the function call of *Function()* in the two programs P\_VALUE.CPP and P\_REF.CPP:

```
Function (i) ;
```

Thus, from a user's point of view, there is no difference in the function call.

Before we discuss the use of the **const** modifier with reference arguments, let us just mention that the ampersand can be attached to either the type specifier or the identifier:

```
Function (int& ir) // general style
```

or

```
Function (int &ir)
```

Which style you choose is completely up to you, and makes no difference to compilation, but the general convention is the former style of attaching the ampersand to the type specifier.

### 6.11.3 **const** Reference Arguments

Passing by reference eliminates the copying of argument parameters, through the use of an alias, but introduces the danger of accidentally altering the value of a function-caller variable. To prevent this from occurring we define the function reference arguments as type **const** (e.g. **const T&**). The program below illustrates this:

```
// p_const.cpp
// illustrates the use of const for passing arguments by
// reference
#include <iostream.h> // C++ I/O

void Function (const int& i)
{
    // error: can't alter i since defined constant
    i = i + 10 ;
}

void main ()
{
    int i = 10 ;

    Function (i) ;
}
```

This program will not compile because *Function()* attempts to modify the **const** variable *i* within its function body.

As a further example, the following illustrates the overloaded **ostream** function **write()**, declared in the header file **IOSTREAM.H**:

```
ostream write (const signed char*, int) ;
ostream write (const unsigned char*, int) ;
```

Note the use of **const**, although with a pointer (\*) instead of a reference, to prevent alteration of the caller's data. Pointers will be discussed in Chapter 12.

### 6.11.4 **const** Return Value

The **const** keyword can be placed after a **class** member function argument list:

```
class Circle
{
private:
    double centre, radius ;
public:
    Circle () ;
    //...
    double Radius () const ;
    double Centre () const ;
    //...
```

---

```

};

//...
double Centre () const
{
    return centre ;
}
```

The **const** keyword is restricted to **class** member functions and will therefore be discussed in more detail in Chapter 9, when we introduce classes. The **const** function declarator indicates to the compiler that the **class** member access function *Centre()* is allowed access to the data member *centre* but should not modify it.

## 6.12 Returning More than One Value

You may be wondering how we return more than one value from a function in C++. Programming languages such as Fortran, Pascal and QBasic provide the *subroutine*, *procedure* and *subprogram*, respectively, to enable multi-return values from a function. C++ adopts a more minimal approach, in that it refrains from having two function types where one will do.

The **return** statement only allows us to return a single value. However, passing by reference allows a function to return more than one value to its caller. The following program illustrates multiple return values for a *Circle()* function:

```

// return2.cpp
// illustrates returning more than 1 value
#include <iostream.h> // C++ I/O
#include <math.h> // atan()

// computes surface area and circumference of a circle
// returns 1 if O.K., 0 if not O.K.
int Circle (const double& radius, double& area, double& circ)
{
    int ok = 1 ;
    if (radius <= 0)
    {
        return !ok ; // return 0: error
    }
    else
    {
        const double PI = 4.0 * atan (1.0) ;

        area = PI * radius * radius ;
        circ = 2.0 * PI * radius ;
        return ok ; // return 1: o.k.
    }
}

void main ()
{
```

```

double circle_rad, circle_area, circle_circ = 0.0 ;

cout << "enter the radius of a circle: " ;
cin >> circle_rad ;

int test = Circle (circle_rad, circle_area, circle_circ) ;

if (test)
{
    cout << "circle area: " << circle_area << endl ;
    cout << "circle circ: " << circle_circ << endl ;
}
else
{
    cout << "illegal radius entered!" << endl ;
}
}

```

Some typical user interaction is:

```

enter the radius of a circle: 5
circle area: 78.5398
circle circ: 31.4159

```

Through the use of reference arguments we are able to return two values from *Circle()*, namely the area and circumference of a circle. This procedure is more convenient than returning single values and using two separate functions:

```

double CircleArea (const double& radius) ;
double CircleCirc (const double& radius) ;
// ...

```

Similarities will be observed between the present discussion of returning more than one value from a single function call and the previous discussions of local and global scope and returning by value or reference.

## 6.13 Return by Reference

In the sections above we have seen passing by value and reference. It is also possible to return by reference. Returning by reference really comes into its own when we examine overloading operators. For example, the following overloads the insertion (<<) operator for use with a user-defined **class**/type Point:

```

friend ostream& operator << (ostream& s, const Point& p) ;

```

For now, it suffices to say that returning by reference allows a function to be placed on the left-hand side of an assignment operator:

```
// r_ref.cpp
```

---

```
// illustrates returning by reference
#include <iostream.h> // C++ I/O

int area = 0 ; // define global variable

int& Area ()
{
    return area ;
}

void main ()
{
    Area () = 10 ;
    cout << Area() ;
}
```

The return type specifier of `Area()` is `int&`. The function `Area()` simply returns the value of the global variable `area`. Within the function body of `main()` the global variable `area` is assigned a value of 10 through use of the `Area()` function on the left-hand side of the assignment expression. After assignment, `Area()` is used to output the value of `area`. Note that `area` has to be made a global variable since you cannot return a reference to a local variable. For instance, the following is illegal:

```
int& Area ()
{
    int area = 0 ;
    return area ;
}
```

because the lifetime of the local variable `area` expires at the closing brace of `Area()` and it is therefore not possible for `Area()` to return a memory address

## 6.14 Why Use Functions?

In the program example above the single program statement within the `MyMax()` function is simple enough not to have been placed in a function body. We could have simply incorporated the statement that tests two numbers into the `main()` function:

```
//...
void main ()
{
    double number1, number2 ;

    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    double largest = number1 > number2 ? number1 : number2 ;

    cout << "largest of the two numbers: " << largest ;
```

```
}
```

This is quite legitimate, since the *MyMax()* function accomplishes a small task. However, as function size increases, this procedure of placing program statements within the *main()* function clearly becomes impractical. Functions also hide away the gory details of a function body, which are of no importance and a distraction to the function-caller. For instance, when a user calls the C++ standard library *rand()* function to return a pseudo-random number, it is not necessary for the user to know every program detail of the *rand()* function. The *rand()* user is simply interested in generating pseudo-random numbers for a particular application in mind. Documentation can enhance a user's understanding of the *rand()* function, but a more detailed knowledge of random number generation will require the user to examine in some detail random numbers and possibly to implement his or her own routine(s).

Functions are reusable, and not only by the programmer(s) who developed the functions. A spin-off from reusability is that functions dramatically reduce program size. Consider developing a library of functions which determine the maximum and minimum of two numbers, the absolute value of a positive or negative number and the square root of a positive number. We shall call these functions appropriately *MyMax()*, *MyMin()*, *MyAbs()* and *MySqrt()*. We have already developed one of these functions above, *MyMax()*, and we now need to place this function and the three additional functions in a form that makes them suitably accessible to other programs – hopefully any C++ program. How do we go about doing this? Well, one way is simply to add our new functions to the above program, FUNC.CPP:

```
// func.cpp
//...
double MyMax (double a, double b)
{ /*...*/ }
double MyMin (double a, double b)
{ /*...*/ }
double MyAbs (double n)
{ /*...*/ }
double MySqrt (double n)
{ /*...*/ }
//...
void main ()
{
//...
}
```

This is OK if the *My* functions are only to be called by *main()* in FUNC.CPP.

What we really need to do is to separate the *My* functions from the *main()* function. We shall place the *My* function *definitions* into a separate *implementation* file called MY\_LIB.CPP, and at the same time place the *My* function *declarations* in a *header* file called MY\_LIB.H.

The form of the implementation file is:

```
// my_lib.cpp
//...
double MyMax (double a, double b)
{
    return a > b ? a : b ;
```

---

```

}

double MyMin (double a, double b)
{
    return a < b ? a : b ;
}
//...

```

while the declaration header file is:

```

//...
double MyMax  (double a, double b) ;
double MyMin  (double a, double b) ;
double MyAbs  (double n) ;
double MySqrt  (double n) ;

```

include-ing the header file MY\_LIB.H in the main program enables us to use our library of functions:

```

// new_func.cpp
//...
#include "my_lib.h"

void main ()
{
    //...
}
```

With the library and main program structure out of the way, let's now examine the implementation details of the *My* functions in MY\_LIB.CPP.

The *MyMin()* function is a simple modification of the *MyMax()* function; just flip round the relational operator. The function *MyAbs()* returns the absolute value of a number. If a number is negative then the negative sign is removed by *MyAbs()*; otherwise if the number is positive the number remains the same. The trickiest function of the four is *MySqrt()*. The square root is calculated using the ancient<sup>1</sup> square root formula:

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right), \quad n = 0, 1, 2, \dots$$

This formula calculates the square root of number *a* ( $>0$ ) by an iterative process:  $x_0, x_1, x_2, \dots, x_n$ . As *n* increases the above formula can be shown to tend to the square root of the number *a*. To start, the iterative process makes an arbitrary guess,  $x_0$ , of the square root (assume for computational purposes that  $x_0=a$ ). The initial value of  $x_0$  is placed in the above formula and a value of  $x_1$  is determined. Then we pass this new value for  $x$ ,  $x_1$ , back into the formula and so on. As an example, consider determining the square root of the constant 2:

---

<sup>1</sup> When I use the word *ancient*, I mean older than I am – 4000 years old in this case.

$$x_1 = \frac{1}{2} \left( x_0 + \frac{2}{x_0} \right) = \frac{1}{2} \left( 2 + \frac{2}{2} \right) = 1.5$$

$$x_2 = \frac{1}{2} \left( x_1 + \frac{2}{x_1} \right) = \frac{1}{2} \left( 1.5 + \frac{2}{1.5} \right) = 1.4166\dots$$

$x_3 = \dots$

Convergence is very fast. The number of correct digits approximately doubles with each iteration. If convergence has occurred, to the required accuracy, before the maximum number of iterations has been reached, the function **breaks** out of a **for**-loop. The function *MySqrt ()* requires only six iterations to obtain an accuracy of 15 decimal places.

*MySqrt ()* calculates the square root of a positive number, and as a result the function has to perform an internal check to see whether the user of the function has passed a negative number. If a negative number is passed the function returns -1 to indicate an error. Here is the complete MY\_LIB.CPP file:

```
// my_lib.cpp
// implementation file for my library of functions

#include "my_lib.h" // my_lib header file

// returns the maximum of two numbers
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}

// returns the minimum of two numbers
double MyMin (double a, double b)
{
    return a < b ? a : b ;
}

// returns the absolute value of a number
double MyAbs (double n)
{
    return n < 0 ? -n : n ;
}

// returns the square root of a positive number
// return -1 if negative number is passed
double MySqrt (double n)
{
    const int MAX_NO_ITERATIONS = 50 ;
    double x_old = n, x_new = 0.0 ;
    double sqrt = 0.0 ;

    if (n < 0) // if negative number
    {
```

---

```

        sqrt = -1.0 ;
    }
else // O.K.
{
    .
// loop thro'
for (int i=0; i<MAX_NO_ITERATIONS; i++)
{
    x_new = (x_old + n / x_old) / 2.0 ;

    // test current accuracy
    if (fabs (x_new - x_old) < TOLERANCE)
    {
        sqrt = x_new ;
        break ;
    }
else
    x_old = x_new ;
}
return sqrt ;
}

```

Note the use of the C++ library function `fabs()`<sup>2</sup>, which is declared in MATH.H and returns the absolute value of a floating-point number. The use of `fabs()` is required in the above `if`-statement because  $(x_{\text{new}} - x_{\text{old}})$  could be less than zero but not necessarily less than TOLERANCE. The accompanying header file is:

```

// my_lib.h
// header file for my_lib.cpp

#ifndef _MY_LIB_H // prevent multiple includes
#define _MY_LIB_H

#include <math.h> // fabs()

const double TOLERANCE = 1e-06 ;

double MyMax (double a, double b) ;
double MyMin (double a, double b) ;
double MyAbs (double n) ;
double MySqrt (double n) ;

#endif // _MY_LIB_H

```

The header file is fairly straightforward. It contains a declaration for each of the four functions and a defined constant, TOLERANCE, which specifies the required level of accuracy of the square root value returned by `MySqrt()`.

---

<sup>2</sup> Equally, the `MyAbs()` function could have been used instead of the C++ library function `fabs()`. The `fabs()` function is used to illustrate including header files within a user-defined header file.

Note the conditional compilation statement in the header file:

```
#ifndef _MY_LIB_H
#define _MY_LIB_H // prevent multiple includes

// include any header files that header file needs
#include <math.h>
//...
// function declarations
double max (double a, double b) ;
//...
#endif // _MY_LIB_H
```

This statement uses the preprocessor directives `#ifndef` (if not defined), `#define` and `#endif` to prevent including a header file several times. As function libraries get more complex it is easy to include the same header file several times in an implementation file, because each header file generally includes other header files. Including a header file more than once is in effect equivalent to redefining a function and will generate a compilation error. One way around this problem is simply to ensure that each header file is included just the once, but this can be difficult and tedious to do. The simplest way to avoid this problem is the above use of the preprocessor directives. If the identifier `_MY_LIB_H` is already defined then the compiler will skip over the subsequent source code up to the `#endif` statement. Alternatively, if `_MY_LIB_H` is not defined then the source code is compiled. Note that the identifier `_MY_LIB_H` directly after `#endif` is commented out. Some commercial compilers accept:

```
#endif _MY_LIB_H
```

but this may not be portable. Therefore, adopt the style of commenting out the identifier.

As a further example consider the declaration of two classes `Line` and `Face`, which include a `Point` **class** declared in the header file `POINT.H`:

```
// point.h
// header file for Point class
#ifndef _POINT_H // prevent multiple includes
#define _POINT_H
//...
class Point
{
private:
    double x, y, z ;
public:
    Point ()
        { x = y = z = 0.0 ; }
    //...
};
#endif // _POINT_H
```

The `Point` **class** will be discussed in detail in Chapter 9. The `Line` and `Face` **class** declarations, in `LINE.H` and `FACE.H` respectively, could be of the form:

```
// line.h
```

```

// header file for Line class
//...
#include "point.h" // include Point class header file

class Line
{
private:
    Point p1, p2 ;
public:
    Line ()
    {
    }
//...
};

// face.h
// header file for Face class
//...
#include "point.h" // include Point class header file
#include "line.h" // include Line class header file

class TriangularFace
{
private:
    Point p1, p2, p3 ;
    Line l1, l2, l3 ;
public:
    TriangularFace ()
    {
    }
//...
};

```

The header file POINT.H is included in LINE.H and FACE.H, and if the conditional preprocessor directive was not used then **class** Point would be declared twice.

Ensure that the identifiers used for the `#ifndef` and `#define` preprocessor directives are identical and that all source code and `#include` statements are within the conditional compilation statement. I always adopt the notation of `_FILENAME_H` for naming conditional compilation identifiers, where FILENAME is the name of the header file. Preceding FILENAME with an underscore ensures that there is no confusion with uppercase `#defined` or `const` constants.

The example program below illustrates typical applications of the *My* functions:

```

// new_func.cpp
// demonstrates the use of implementation and header files
// my_lib.cpp and my_lib.h
#include <iostream.h> // C++ I/O

#include "my_lib.h" // my_lib header file

void main ()
{
    double number1, number2 ;

```

```

cout << "enter a positive and a negative number: " ;
cin >> number1 >> number2 ;

cout << "MyMax (" << number1 << ", " << number2 << "): "
    << MyMax (number1, number2) << endl ;
cout << "MyMin (" << number1 << ", " << number2 << "): "
    << MyMin (number1, number2) << endl ;
cout << "MyAbs (" << number2 << "): "
    << MyAbs (number2) << endl ;
cout << "MySqrt (" << number1 << "): "
    << MySqrt (number1) << endl ;
}

```

Note that the header file MY\_LIB.H is included within double quotes (" ") rather than angle brackets (<>) because the header file resides in the same directory as the main program and is a user-defined file. The distinction between " " and <> was discussed in Chapter 3. The following shows some user interaction:

```

enter a positive and a negative number: 30 -10.4
MyMax (30, -10.4): 30
MyMin (30, -10.4): -10.4
MyAbs (-10.4): 10.4
MySqrt (30): 5.47723

```

## 6.15 Overloaded Functions

Overloading allows a function to perform different actions depending on the data types that it is sent<sup>3</sup>. A function is said to be overloaded when different function declarations exist for the same function name in the same scope. Consider a program that requires three different functions to output a message that is dependent on the type of argument passed to the function. One way is simply to name each function differently:

```

//...
void FunctionChar (char ch)
{ /*...*/ }

void FunctionInt (int i)
{ /*...*/ }

void FunctionDouble (double d)
{ /*...*/ }
//...

```

---

<sup>3</sup> Overloaded functions are an example of compile-time polymorphism in which a single function can have multiple different operations.

This works fine, but requires a programmer to know that three differently named functions exist for types **char**, **int** and **double**. C++ allows a function name to be overloaded:

```
// over.cpp
// illustrates overloaded functions
#include <iostream.h> // C++ I/O

void Function (char ch)
{
    cout << "Function (char): " << ch << endl ;
}

void Function (int i)
{
    cout << "Function (int): " << i << endl ;
}

void Function (double d)
{
    cout << "Function (double): " << d << endl ;
}

void main ()
{
    Function ('c') ; // char
    Function (1) ; // int
    Function (1.0) ; // double
}
```

with output:

```
Function (char): c
Function (int): 1
Function (double) 1
```

The above example illustrates overloading the same function name with the same number of arguments (in this case one argument). Multiple arguments are also valid:

```
//...
void Function (int a)
{ //... }

void Function (int a, int b)
{ //... }

void Function (int a, double b, char c)
{ //... }
//...
```

How does the compiler know which function to choose? The correct function is selected by comparing the number and types of the actual function arguments with the formal arguments.

However, an overloaded function cannot only differ in its return type. For instance, consider the following program code:

```
void Function (int i) ;
int Function (int i) ;
double Function (int i) ;
//...
void main ()
{
//...
Function (0) ; // call which Function()?
//...
}
```

For the above call to *Function()* within *main()* the compiler would not know which version of *Function()* to choose and would correspondingly issue a compilation error.

Frequently in programming, particularly with classes that we shall observe in Chapter 9, we are required to *get* and *set* a data value (say *position*). One way to do this is simply to define two functions:

```
//...
double GetPosition () ; // gets position
void SetPosition (double pos) ; // sets position
```

A more concise way is to overload a more general *Position()* function that can both get and set the value of *position*:

```
// get_set.cpp
// illustrates function overloading
#include <iostream.h> // C++ I/O

double position = 10.0 ; // some global variable: position

// gets position
double Position ()
{
return position ;
}

// sets position
void Position (double pos)
{
position = pos ;
}

void main ()
{
cout << "get position: " << Position () << endl ;
Position (20) ;
cout << "get position: " << Position () << endl ;
}
```

Overloading *Position()* allows us to get and set a data value with a single function name.

Overloaded functions are clearly a very powerful feature of C++, but there are a few exceptions that you should be aware of. The first case illustrates arguments of type T and T&. Since types T and T& accept the same set of initialisers, the following two functions are indistinguishable from one another:

```
//...
void Func (int i)
{
    cout << endl << "Func (int)" ;
}

// error: Func(int&) can't be distinguished from Func(int)
void Func (int& i)
{
    cout << endl << "Func (int&)" ;
}
```

This example code is extracted from OVERLOAD.CPP. Another case, also extracted from OVERLOAD.CPP, which requires some careful consideration is the following:

```
//...
void Func (int i)
{
    cout << endl << "Func (int)" ;
}

// error: Func(float) can't be distinguished from Func(int)
void Func (float f)
{
    cout << endl << "Func (float)" ;
}

void main ()
{
    Func (1) ;
    Func (1.0) ;
}
```

The compiler is unable to distinguish between *Func(int)* and *Func(float)* in the function calls. The solution is to use the suffix f or F on the constant of type **float** that is passed as an argument:

```
//...
void main ()
{
    Func (1) ;
    Func (1.0f) ; // suffix f removes ambiguity
}
```

The following is also legal:

```
//...
void Func (int i)
{
    cout << endl << "Func (int)" ;
}

void Func (double d)
{
    cout << endl << "Func (double)" ;
}

void main ()
{
    Func (1) ;           // int
    Func (1.0) ;         // double
}
```

The following program code eliminates any ambiguity, but the function call *Func(1.0)* will call the **double** version:

```
//...
void Func (int i)
{
    cout << endl << "Func (int)" ;
}

void Func (float f)
{
    cout << endl << "Func (float)" ;
}

void Func (double d)
{
    cout << endl << "Func (double)" ;
}

void main ()
{
    Func (1) ;           // int
    Func (1.0) ;         // double
}
```

Thus, to eliminate any confusion or ambiguity in function overloading when the argument(s) are both floating-point numbers of type **float** and **double**, use the suffix **f** or **F** for arguments of type **float**.

We shall examine **typedefs** and enumerations in Chapter 8, but it is worth highlighting that a **typedef** is a synonym for an existing type and not a new type, and therefore functions with the same name and number of arguments cannot be overloaded using **typedefs**:

```
//...
void Func (int i) ;
```

---

```
//...
typedef int Integer ;

// error: re-defines Func (int)
void Func (Integer ti) ;
```

The above is an error because *Func(Integer)* redefines *Func(int)*. Enumerations, however, are distinct types, and functions involving enumerations can be overloaded:

```
//...
void Func (int i)

enum Enum
{
    a
} ;

void Func (Enum e) ;
```

There are several other subtle distinctions to be found in function overloading, so feel free to experiment with the OVERLOAD.CPP program.

## 6.16 Default Arguments

C++ allows a function to be called without specifying all of its arguments. How is this possible? If a *function declaration* supplies default values for certain or all arguments then a function can be called without values for these arguments:

```
// def_arg.cpp
// illustrates default function arguments
#include <iostream.h> // C++ I/O

// prototype with default argument values
double RectangleArea (const double& width=12.0,
                      const double& height=5.0) ;

// returns area of a rectangle
double RectangleArea (const double& width,
                      const double& height)
{
    return width * height ;
}

void main ()
{
    cout << "area (12, 5): " << RectangleArea ()      << endl ;
    cout << "area (8, 5): " << RectangleArea (8)     << endl ;
    cout << "area (2, 10): " << RectangleArea (2, 10) << endl ;
}
```

which produces the following output:

```
area (12, 5): 60
area (8, 5): 40
area (2, 10): 20
```

The first statement in *main()* uses both default argument values of *RectangleArea()*. The second statement just uses one default argument (*height=5.0*), and the last statement uses no default arguments. In the case of the second statement, how does the compiler know which of the two default arguments to choose? The system adopted by the compiler is that only trailing arguments in an argument list can be assigned default values. For instance:

```
Triangle (const Point& p1, const Point& p2, const Point& p3,
          const Surface& surface=TWOD, const Colour&
          colour=BLACK) ;
```

The first three function arguments of *Triangle()* define the vertices of a triangle and the last two arguments set the default surface and colour attributes of a triangle to two-dimensional and black, respectively.

It is important to observe that default arguments are restricted to the declaration of a function. Therefore, in the above example of *RectangleArea()* the following is an illegal use of default arguments:

```
double RectangleArea (const double& width=12.0,
                      const double& height=5.0)
{
//...
}
```

Can a default argument expression use a local function variable? In the above program, DEF\_ARG.CPP, we may alternatively require that *RectangleArea()* defaults to the area of a square if the height of a rectangle is unspecified:

```
double RectangleArea (const double& width=12.0,
                      const double& height=width) ; // error
```

Regrettably, the use of local variables in default argument expressions is illegal in C++.

## 6.17 Local static Variables

Chapter 4 examined the use of the **static** storage class specifier for global variables. Let us now examine local **static** variables. The following program compares two functions containing non-**static** and **static** local function variables:

```
// static.cpp
// illustrates local static variables
#include <iostream.h> // C++ I/O

// non-static local variable
```

---

```

int Function0 ()
{
    int ns_count = 0 ;
    return ++ns_count ;
}

// static local variable
int Function1 ()
{
    static int s_count = 0 ;
    return ++s_count ;
}

void main ()
{
    cout << "non-static local variable Function0(): " ;
    for (int i=0; i<5; i++)
        cout << Function0 () << " " ;
    cout << endl ;

    cout << "      static local variable Function1(): " ;
    for (int j=0; j<5; j++)
        cout << Function1 () << " " ;
    cout << endl ;

//ns_count = 0 ; // error: local to Function0()
//s_count = 0 ; // error: local to Function1()
}

```

with output:

```

non-static local variable Function0(): 1 1 1 1 1
      static local variable Function1(): 1 2 3 4 5

```

*Function0()* defines and initialises to zero an **int** non-**static** local variable called *ns\_count* and then returns *ns\_count*'s prefix increment value. As expected, each time *Function0()* is called from the **for**-loop in *main()* the value of 1 is returned by *Function0()* because upon returning from *Function0()* the value of *ns\_count* is lost. However, the program output illustrates that this is not the case for **static** local variables. Each time *Function1()* is called *s\_count* maintains its value from the last call to *Function1()*.

To understand why *s\_count* maintains its value between function calls, it helps to note that *s\_count* is defined and initialised to zero at the start of *Function1()* – so why isn't *s\_count* zeroed upon each call to *Function1()*? The answer lies in the fact that local **static** variables are identical to global variables in that they are defined and initialised only once and that their lifetime extends to the end of the program and not when they go out of scope when a function returns. Where local **static** variables differ from global variables is that a local **static** variable's scope is only from its point of definition to the end of the function block in which it resides. The scope of the local non-**static** and static variables *ns\_count* and *s\_count* is illustrated at the end of the *main()* function above.

## 6.18 **inline** Functions

Functions save memory space because all function calls use the same executable code. The disadvantage of function calls is that they take time. Consider the *MyMax()* function call:

```
//...
double MyMax (double a, double b)
{
    return a > b ? a : b ;
}

void main ()
{
//...
double largest = MyMax (number1, number2) ;
//...
}
```

When *MyMax()* is called in *main()* then control goes to *MyMax()* and when *MyMax()* returns control goes back to *main()*. This transfer of control can involve large memory jumps. If we simply expand the *MyMax()* function code inline in *main()*, no transfer of control is required:

```
void main ()
{
//...
double largest = a > b ? a : b ;
//...
}
```

Clearly, if the function is large this is not feasible, and at the same time we have lost a neat function *MyMax()*.

C programmers would implement the *MyMax()* function **inline** by defining a macro:

```
#define MYMAX(a, b) ( ((a) > (b)) ? (a) : (b) )
```

Macros don't generate any function overhead code. All of a macro's code is generated inline, and thus executes faster than its function counterpart because memory access is sequential. However, macros generate inline code, which means that each time you call the macro the compiler generates the macro code, whereas the compiler generates function code just once. Also, a major disadvantage of macros is that they receive no static type checking, and the use of parentheses can be confusing and the source of difficult-to-find errors.

C++ offers us the **inline** keyword to make a function inline. The **INLINE.CPP** program illustrates the use of **inline** with the *MyMax()* function:

```
// inline.cpp
// illustrates an inline function
#include <iostream.h> // C++ I/O

// returns the maximum of two numbers
```

---

```

inline double MyMax (double a, double b)
{
    return a > b ? a : b ;
}

void main ()
{
    double number1, number2 ;

    cout << "enter two positive or negative numbers: " ;
    cin >> number1 >> number2 ;

    double largest = MyMax (number1, number2) ;

    cout << "largest of the two numbers: " << largest ;
}

```

The keyword **inline** inserts the function code directly **inline** in the code of the function caller. Ensure that the function definition appears before an **inline** function call. This is simply because it is not enough for the compiler to see just the function declaration before a function call: the compiler has to insert the inline function code in the function-caller program.

So which functions do you make **inline**? Generally, reserve **inline** functions for small functions that are frequently called. Inline functions are usually defined with their declaration in their corresponding header file in order to prevent the problem mentioned above of a function call occurring before the function definition. Actually, the compiler ultimately decides what action to take with **inline** functions. The compiler can ignore the **inline** request if it considers that the function is unsuitable to be made **inline**.

## 6.19 The *main()* Function

In Chapter 3 the *main()* function was introduced, and we have seen every program so far use the *main()* function. So, before we discuss external linkage and C++ library functions let's take a closer look at the *main()* function.

Every C++ console-based program must call the *main()* startup function. The *main()* function indicates the start and end of program execution via the opening, {, and closing, }, braces of the *main()* function body. If *main()* is not defined within your program you will not get correct linkage. Where you place *main()* within your program is completely up to you. Programmers generally place *main()* either at the start or the end of a program. There appears to be no general consensus about the placement of *main()*, but I always place *main()* at the end of the program, so I know where to find it.

All of the programs that we have seen so far have defined *main()* in the general form:

```

//...
void main ()
{
    //...
}

```

which indicates that the return type of *main()* is **void** and that it does not return a value. It was mentioned above that we could explicitly indicate that *main()* has no argument list as:

```
void main (void)
{
//...
}
```

The value returned by *main()* is the status code of the program, and is in fact an integer. Thus, it is legal to define *main()*'s return type specifier as **int**:

```
int main ()
{
//...
return 0 ; // normal program termination
}
```

Type **int** is the default return type of the *main()* function for the majority of C++ compilers and hence the reason why the return type specifier of *main()* can be **void**. The value returned by *main()* is used to indicate to the operating system the program's exit status. Most operating systems indicate normal exit by a zero and abnormal exit by a non-zero value. For instance, the *exit()* function, which terminates program execution, uses two error status signals (defined in STDLIB.H): EXIT\_SUCCESS (0) and EXIT\_FAILURE (1), which indicate normal and abnormal program termination respectively.

### 6.19.1 Command Line Arguments

The *main()* function can be defined with *command line arguments*. A typical use of command line arguments is in executing Microsoft's DOS screen editor. The editor can simply be executed by typing the editor's executable program name (EDIT) followed by a carriage return and then proceeding to open a text file by way of File | Open. Alternatively, the text file to edit can be passed as a command line argument:

```
EDIT \CPP_PROG\CHAP06\CHAP06.TXT
```

which incorporates the path and filenames of the text file as a command line argument. We shall continue our discussion of command line arguments further when we discuss input and output, files, and streams in Chapter 17, but for now just bear in mind that they are possible.

### 6.19.2 WinMain()

Windows programs call the *WinMain()* function at startup instead of *main()*:

```
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                     LPSTR lpszCmdLine, int nCmdShow)
{
//...
if (!hPrevInstance) // if first instance
{
//...
}
hwnd = CreateWindow /*...*/ // create window procedure
//...
```

---

```

ShowWindow (hwnd, nCmdShow) ; // show & update
UpdateWindow (hwnd) ;
//...
while (GetMessage (&msg, NULL, 0, 0) // message loop
{
//...
}
return msg.wParam ; // return messages to operating system
}

```

If you have programmed for Windows using the Borland ObjectWindows **class** library then you may have seen the *OwlMain()* function.

## 6.20 External Linkage

Chapter 4 examined the use of the **extern** storage class specifier for global variables. We are now in a position to discuss **extern** as applied to functions. Let us first define a function, *Function()*, in FILE.CPP:

```

// file.cpp
// a function definition
#include <iostream.h> // C++ I/O

void Function ()
{
    cout << "Function() called" << endl ;
}

```

If we would like to use *Function()* in a separate file, say EXTERN.CPP, then we can use the **extern** specifier in a *linkage specification* to declare *Function()* and inform the compiler that external linkage is required to obtain the definition of *Function()*:

```

// extern.cpp
// illustrates the extern storage class specifier

extern Function () ;

void main ()
{
    Function () ;
}

```

with output:

```
Function() called
```

It is perfectly legal to link the above two files FILE.CPP and EXTERN.CPP, even though they both contain a reference to *Function()*, because EXTERN.CPP contains an **extern** declaration of *Function()* and not a definition. As highlighted in Chapter 4, a program can contain multiple variable or function declarations but only one definition of the same named variable or function.

The **extern** specifier can also be used to inform the compiler of linkage between C++ and non-C++ variables and functions. C++/non-C++ linkage is achieved by placing a string literal in the linkage specification which can have one of the two following general forms:

```
extern string_literal declaration;  
  
extern string_literal  
{  
// multiple declarations  
}
```

where *declaration* is any legal variable or function declaration. The second form of the **extern** declaration includes multiple **extern** declarations between the open, {, and close, }, braces. The *string\_literal* indicates the kind of linkage, such as C++, C or Fortran. C and C++ linkage are guaranteed by all ANSI/ISO draft standard compatible compilers, with C++ linkage being the default, whereas with other languages linkage is implementation-dependent. To illustrate the above two **extern** C and C++ linkage specifications, consider the program:

```
// extern1.cpp  
// further illustrates the extern storage class specifier  
#include <iostream.h> // C++ I/O  
  
extern double fabs (double x) ; // default C++ linkage  
  
extern "C" double sqrt (double x) ;  
extern "C++" double pow (int x, int y) ;  
  
extern "C"  
{  
double sin (double x) ;  
double cos (double x) ;  
double tan (double x) ;  
}  
  
void main ()  
{  
cout << "tan(1): " << tan (1.0) << endl ;
```

with output:

```
tan(1): 1.55741
```

Note that the C++ standard library header file MATH.H is not included in the above program.

## 6.21 Library Functions

Aside from learning the syntax, semantics and pragmatics of a programming language, it is equally important to learn what tools are shipped with the language. These tools are generally

in the form of run-time function and **class** libraries. This section presents an overview of functions supplied with C++, whereas C++'s standard **class** library will be covered from Chapter 9 onwards, when classes are discussed in more detail.

Attempting to program in C++ without getting to know the function and **class** libraries would be nearly impossible and clearly impractical. The libraries save you from reinventing the wheel every time you want to do the simplest of things. Also, the shipped libraries are generally robust and efficient and have been developed over a number of years by professionals.

At a first glance, the C++ libraries can appear overwhelming and incomprehensible. Don't attempt to learn every function and every detail of each function. This is impossible and not necessary. Most experienced programmers develop a general knowledge of the C++ libraries and become familiar with most of the functions in the libraries. Try to tailor your understanding of the C++ libraries according to your needs. If you are heavily involved in numerical and computational work, the mathematical functions in the MATH.H header will be extremely useful.

For example, say that you are required to determine the square root of a number. Do you spend a week developing your own square root function? Maybe, but a quicker way is to check your compiler's documentation. A function called *sqrt* () is part of the ANSI library functions and is declared in the header file MATH.H:

```
double sqrt (double x) ;
```

A check with my C++ compiler's documentation confirms that this function calculates the square root of a positive number of type **double**, which is passed as an argument. If the argument is negative, the global variable **errno** is set to the identifier EDOM, which indicates a domain error.

Another related non-ANSI function is:

```
long double sqrtl (long double x) ;
```

which takes a **long double** argument and returns a **long double** square root result.

Further, we note that the *sqrt* () function can be used with **bcd** (binary-coded decimal) and **complex** types. Class **bcd** is declared in the BCD.H header file and the *sqrt* () function is declared as:

```
friend bcd sqrt (bcd&) ;
```

while the **complex** square root function is declared in the header file COMPLEX.H as:

```
friend complex sqrt (complex&) ;
```

This example has illustrated that determining the square root of a number in C++ is well covered. The compiler documentation has provided us with the necessary implementation details and indicated that *sqrt* () can operate also on **bcd** and **complex** types.

### 6.21.1 The ANSI C Standard Library

When ANSI formed a standard for the Clanguage it also defined a standard library of functions. Any compiler that is an ANSI C-compliant compiler has to supply the ANSI C standard library functions, and each of these functions is completely independent of the compiler and hardware used. Table 6.1 lists all macros and functions that are ANSI C-compatible. The table is subdivided into the following routines: classification, conversion, diagnostic, input/output, interna-

**Table 6.1** ANSI C macros and functions.

<i>Function</i>	<i>Description</i>	<i>Header file</i>
<b>Classification</b>		
<i>isalnum</i>	tests for an alphanumeric character	CTYPE.H
<i>isalpha</i>	tests for letters	CTYPE.H
<i>iscntrl</i>	tests for control characters	CTYPE.H
<i>isdigit</i>	tests for digits	CTYPE.H
<i>isgraph</i>	tests for printing characters	CTYPE.H
<i>islower</i>	tests for lowercase letters	CTYPE.H
<i>isprint</i>	tests for printable characters	CTYPE.H
<i>ispunct</i>	tests for punctuation characters	CTYPE.H
<i>isspace</i>	tests for white space	CTYPE.H
<i>isupper</i>	tests for uppercase letters	CTYPE.H
<i>isxdigit</i>	tests for hexadecimal digits	CTYPE.H
<b>Conversion</b>		
<i>atof</i>	converts a string to a floating-point number	STDIO.H
<i>atoi</i>	converts a string to an integer	STDIO.H
<i>atol</i>	converts a string to a long	STDIO.H
<i>strtod</i>	converts a string to a double	STDIO.H
<i>strtol</i>	converts a string to a long	STDIO.H
<i>strtoul</i>	converts a string to an unsigned long	STDIO.H
<i>tolower</i>	converts characters to lowercase	STDIO.H
<i>toupper</i>	converts characters to uppercase	STDIO.H
<b>Diagnostic</b>		
<i>assert</i>	tests a condition and possibly aborts	ASSERT.H
<i>perror</i>	prints a system error message	ERRNO.H
<b>Input/output</b>		
<i>clearerr</i>	resets error indication	STDIO.H
<i>fclose</i>	closes a stream	STDIO.H
<i>feof</i>	detects an end-of-file on a stream	STDIO.H
<i>ferror</i>	detects errors on a stream	STDIO.H
<i>fflush</i>	flushes a stream	STDIO.H
<i>fgetc</i>	gets a character from a stream	STDIO.H
<i>fgetpos</i>	gets the current file pointer	STDIO.H
<i>fgets</i>	gets a string from a stream	STDIO.H
<i>fopen</i>	opens a stream	STDIO.H
<i>fprintf</i>	writes formatted output to a stream	STDIO.H
<i>fputc</i>	puts a character on a stream	STDIO.H
<i>fputs</i>	outputs a string on a stream	STDIO.H
<i>fread</i>	reads data from a stream	STDIO.H
<i>freopen</i>	associates a new file with an open stream	STDIO.H
<i>fscanf</i>	scans and formats input from a stream	STDIO.H
<i>fseek</i>	repositions a file pointer on a stream	STDIO.H
<i>fsetpos</i>	positions the file pointer of a stream	STDIO.H
<i>fstat</i>	gets open file information	SYSSTAT.H
<i>ftell</i>	returns the current file pointer	STDIO.H
<i>fwrite</i>	writes to a stream	STDIO.H
<i>getc</i>	gets a character from a stream	STDIO.H
<i>getchar</i>	gets a character from <code>stdin</code>	STDIO.H
<i>getenv</i>	gets a string from the environment	STDIO.H
<i>gets</i>	gets a string from <code>stdin</code>	STDIO.H
<i>longjmp</i>	performs a non-local goto	SETJMP.H
<i>printf</i>	writes formatted output to <code>stdout</code>	STDIO.H
<i>putc</i>	outputs a character to a stream	STDIO.H
<i>putchar</i>	outputs a character to <code>stdout</code>	STDIO.H
<i>puts</i>	outputs a string to <code>stdout</code>	STDIO.H
<i>remove</i>	removes a file	STDIO.H
<i>rename</i>	renames a file	STDIO.H
<i>rewind</i>	repositions the file pointer to the start of a stream	STDIO.H
<i>scanf</i>	scans and formats input from <code>stdin</code>	STDIO.H

**Table 6.1** (contd.)

<i>Function</i>	<i>Description</i>	<i>Header file</i>
<i>setbuf</i>	assigns buffering to a stream	STDIO.H
<i>setjmp</i>	sets up for non-local goto	SETJMP.H
<i>setvbuf</i>	assigns buffering to a stream	STDIO.H
<i>sprintf</i>	writes formatted output to a stream	STDIO.H
<i>sscanf</i>	scans and formats input from a string	STDIO.H
<i>stat</i>	gets information about a file	SYS\STAT.H
<i>tmpfile</i>	opens a temporary file for binary mode	STDIO.H
<i>tmpnam</i>	creates a unique filename	STDIO.H
<i>ungetc</i>	pushes a character back into an input stream	STDIO.H
<i>vfprintf</i>	write formatted output to a stream	STDIO.H
<i>vprintf</i>	write formatted output to stdout	STDIO.H
<i>vsprintf</i>	write formatted output to a string	STDIO.H
International locale API		
<i>localeconv</i>	gets the current locale structure	LOCALE.H
<i>setlocale</i>	selects or queries a locale	LOCALE.H
Manipulation		
<i>bsearch</i>	binary search of an array	STDLIB.H
<i>memchr</i>	searches for a character	MEM.H, STRING.H
<i>memcmp</i>	compares two blocks	MEM.H, STRING.H
<i>memcpy</i>	copies a block	MEM.H, STRING.H
<i>memmove</i>	copies a block	MEM.H, STRING.H
<i>memset</i>	sets bytes of a block to a specified value	MEM.H, STRING.H
<i>qsort</i>	sorts according to the quicksort algorithm	STDLIB.H
<i>strcat</i>	appends one string to another	STRING.H
<i>strchr</i>	scans a string for a character	STRING.H
<i>strcmp</i>	compares two strings	STRING.H
<i>strcoll</i>	compares two strings	STRING.H
<i>strcpy</i>	copies one string into another	STRING.H
<i>strcspn</i>	scans a string for a subset of characters	STRING.H
<i>strerror</i>	gets an error message string	STRING.H
<i>strlen</i>	gets the length of a string	STRING.H
<i>strncat</i>	appends a portion of one string to another	STRING.H
<i>strncmp</i>	compares a portion of one string with another	STRING.H
<i>strncpy</i>	copies bytes from one string to another	STRING.H
<i>strnbrk</i>	scans a string for a character	STRING.H
<i>strrchr</i>	scans a string for a character	STRING.H
<i>strspn</i>	scans a string for a subset of characters	STRING.H
<i>strstr</i>	scans a string for a substring	STRING.H
<i>strtok</i>	searches a string for tokens	STRING.H
<i>strxfrm</i>	transforms a portion of a string	STRING.H
Mathematical		
<i>abs</i>	absolute value of an integer	COMPLEX.H, STDLIB.H
<i>acos</i>	arc cosine	COMPLEX.H, MATH.H
<i>asin</i>	arc sine	COMPLEX.H, MATH.H
<i>atan</i>	arc tangent	COMPLEX.H, MATH.H
<i>atan2</i>	arc tangent of $y/x$	COMPLEX.H, MATH.H
<i>ceil</i>	rounds up	MATH.H
<i>cos</i>	cosine	COMPLEX.H, MATH.H
<i>cosh</i>	hyperbolic cosine	COMPLEX.H, MATH.H
<i>div</i>	divides two integers	MATH.H
<i>exp</i>	exponential	COMPLEX.H, MATH.H
<i>fabs</i>	absolute value of a floating-point number	MATH.H
<i>floor</i>	rounds down	MATH.H
<i>fmod</i>	$x$ modulo $y$	MATH.H
<i>frexp</i>	divides a number into a mantissa and exponent	MATH.H
<i>log</i>	natural logarithm	COMPLEX.H, MATH.H
<i>log10</i>	logarithm to base 10	COMPLEX.H, MATH.H
<i>pow</i>	$x$ to the power $y$	COMPLEX.H, MATH.H
<i>rand</i>	random number generator	STDLIB.H

**Table 6.1** (contd.)

<i>Function</i>	<i>Description</i>	<i>Header file</i>
<i>sin</i>	sine	COMPLEX.H, MATH.H
<i>sinh</i>	hyperbolic sine	COMPLEX.H, MATH.H
<i>sqrt</i>	square root	COMPLEX.H, MATH.H
<i>rand</i>	random number initialiser	STDLIB.H
<i>tan</i>	tangent	COMPLEX.H, MATH.H
<i>tanh</i>	hyperbolic tangent	COMPLEX.H, MATH.H
Memory		
<i>calloc</i>	allocates main memory	ALLOC.H, STDLIB.H
<i>free</i>	frees an allocated block of memory	ALLOC.H, STDLIB.H
<i>malloc</i>	allocates main memory	ALLOC.H, STDLIB.H
<i>realloc</i>	reallocates main memory	ALLOC.H, STDLIB.H
Process control		
<i>abort</i>	abnormally terminates a program	PROCESS.H
<i>atexit</i>	registers termination function	STDLIB.H
<i>exit</i>	terminate program	PROCESS.H
<i>raise</i>	send software signal to program	SIGNAL.H
<i>signal</i>	specifies signal handling actions	SIGNAL.H
Time and date		
<i>asctime</i>	converts date and time to ASCII	TIME.H
<i>clock</i>	processor time	TIME.H
<i>ctime</i>	converts date and time to a string	TIME.H
<i>difftime</i>	difference between two times	TIME.H
<i>gmtime</i>	converts date and time to Greenwich Mean Time	TIME.H
<i>localtime</i>	converts date and time to a structure	TIME.H
<i>mktime</i>	converts time to calendar format	TIME.H
<i>strftime</i>	formats time for output	TIME.H
<i>time</i>	gets the time of day	TIME.H
Variable arguments		
<i>va_arg</i>	access to variable argument list	STDARG.H
<i>va_end</i>	access to variable argument list	STDARG.H
<i>va_start</i>	access to variable argument list	STDARG.H

tional locale API, manipulation, mathematical, memory, process control, time and date, and variable arguments.

## 6.22 Header Files

Header files (\*.H) provide function and **class** declarations. This section divides the ANSI C and C++ header files into two subsections: ANSI C and C++. Each subsection gives a complete list of the header filenames and a brief description of each header file.

### 6.22.1 ANSI C Header Files

ASSERT.H	<i>assert ()</i> macro
CTYPE.H	character classification and conversion
ERRNO.H	constant mnemonics for error codes
FLOAT.H	implementation-specific floating-point number routines
LIMITS.H	implementation-specific limits on type values
LOCALE.H	country and language specific routines

---

MATH.H	mathematical functions and error handlers
SETJMP.H	defines a <b>typedef</b> jmp_buf and declares the functions <i>setjmp()</i> and <i>longjmp()</i>
SIGNAL.H	signal and raise functions
STDARG.H	definitions for accessing parameters in functions that accept a variable number of arguments, such as <i>vprintf()</i>
STDDEF.H	defines several common data types and macros
STDIO.H	standard input and output
STDLIB.H	declares several common routines
STRING.H	string and memory manipulation
TIME.H	time and time conversion

## 6.22.2 C++ Header Files

BCD.H	<b>class</b> bcd
CHECKS.H	diagnostics.
COMPLEX.H	complex math <b>class</b>
CONSTREA.H	console output support
CSTRING.H	string <b>class</b>
EXCEPT.H	exceptions
FSTREAM.H	file input and output stream classes
IOMANIP.H	input and output stream manipulators
IOSTREAM.H	standard input and output stream classes
NEW.H	new_handler and set_new_handler
REF.H	reference counting support for <b>class</b> string
REGEXP.H	regular expression searching
STDIOSTR.H	stream classes for use with stdio FILE structures
STRSTREA.H	stream classes for use with byte arrays in memory
TYPEINFO.H	run-time type identification

## 6.22.3 Additional Header Files

This section lists header files that are not specifically defined by ANSI C or C++.

ALLOC.H	memory management functions
BIOS.H	IBM PC ROM BIOS routines
CONIO.H	operating system console input and output routines
DIR.H	directory and path names
DIRECT.H	directory and path names
DIRENT.H	POSIX directory operations
DOS.H	MS-DOS and 80x86 specific calls
EXCPT.H	C exceptions support
FCNTL.H	specific constants for the <i>open()</i> function
GENERIC.H	macros for generic class declarations.
IO.H	low-level input and output routines.
SYS\LOCKING.H	definitions for mode parameter of <i>locking()</i> function
MALLOC.H	memory management
MEM.H	memory manipulation
MEMORY.H	memory manipulation
PROCESS.H	process control

SEARCH.H	searching and sorting
SYS\STAT.H	opening and creating files
SYS\TIMEB.H	structure and function declarations for <i>ftime()</i>
SYS\TYPES.H	declares <b>typedefs</b> used with time functions
UTIME.H	declares <i>utimbuf</i> structure and <i>utime()</i> function
VALUES.H	constants for Unix compatibility.
VARARGS.H	variable number of arguments for Unix compatibility.

## 6.23 Not in the Library

If you require a function that is not provided by your compiler, there are the following main options to consider:

1. Do the compiler's libraries provide a similar function?
2. Can a new function be developed which comprises several existing functions?
3. Is a function available in another language which can be called from C++ using external linkage?
4. Is there a suitable library function floating around on the Internet that I can appropriate?
5. Is there a vendor software library available that does the job, and is it one that I can afford?

or finally

6. Oh, no: I'm actually going to have write my own function!

## 6.24 Intersection of Two Two-Dimensional Line Segments

Before we bring this chapter to a close, let's examine the intersection of two straight line segments<sup>4</sup> in a two-dimensional space (Fig. 6.3). This problem is particularly interesting in geometry, sufficiently well covered in the literature, and a good realistic problem to illustrate more complex applications of functions.

At first glance, the intersection of two straight line segments appears to be a straightforward problem, but the more you examine the problem the more complicated it becomes. There are a number of alternative methods that can be used for determining whether two lines intersect or not, the first of which is the most straightforward and direct.

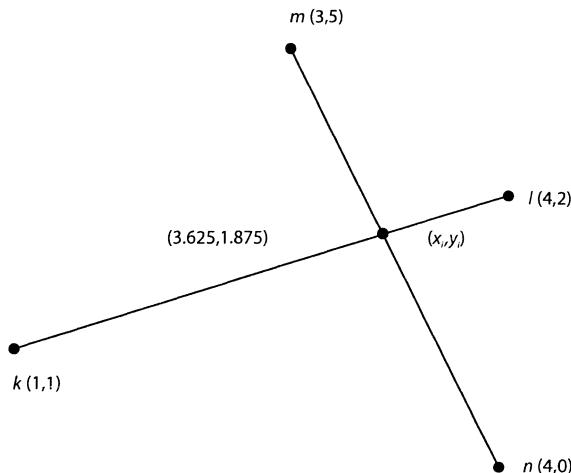
### 6.24.1 Parametric Lines

The procedure outlined in this section follows the method of Bowyer and Woodwark (1983, pp. 48–53).

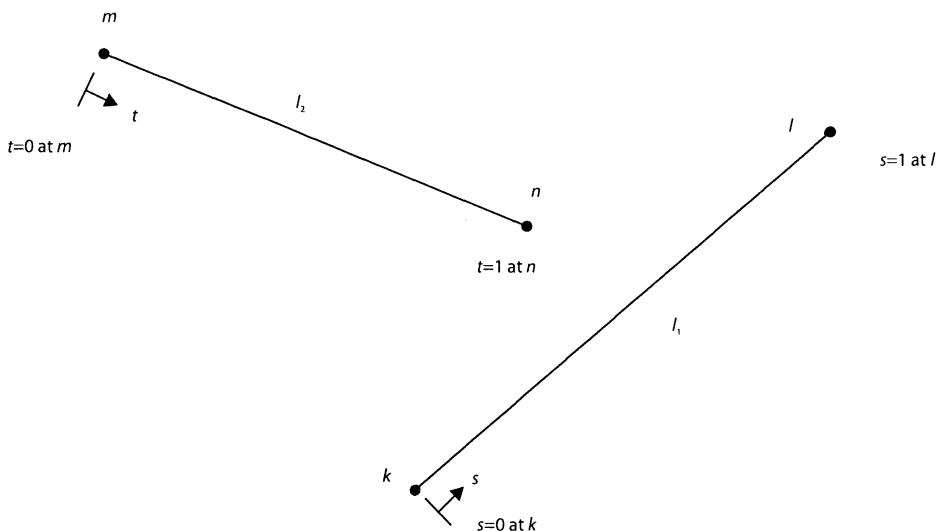
It is possible to determine whether two line segments intersect or not from the parametric form of the two lines. Figure 6.4 shows two lines  $l_1(k, l)$  and  $l_2(m, n)$ , whose end-points are  $(k, l)$  and  $(m, n)$  respectively. The parametric form of line  $l_1$  is:

---

4 Note the distinction between a line and a line segment. A line can, in theory, be of infinite length, whereas a line segment has finite length and has two distinct end-points. Therefore, any straight line segment can be extended indefinitely in a straight line; Euclid, 300 BC.



**Fig. 6.3** Two intersecting straight line segments  $l_1(k, l)$  and  $l_2(m, n)$ . The point of intersection is denoted by  $(x_i, y_i)$ .



**Fig. 6.4** Parametric representation of two lines.

$$x = x_k + (x_l - x_k)s$$

$$y = y_k + (y_l - y_k)s$$

so that  $s=0$  at the start point,  $k$ , and  $s=1$  at the terminal point,  $l$ . Similarly, if we represent line  $l_2$  in parametric form, using the parametric variable  $t$ , we have:

$$x = x_m + (x_n - x_m)t$$

$$y = y_m + (y_n - y_m)t$$

Eliminating  $x$  and  $y$  from the above parametric equations:

$$x_m + (x_n - x_m)t = x_k + (x_l - x_k)s$$

$$y_m + (y_n - y_m)t = y_k + (y_l - y_k)s$$

Hence:

$$s = \frac{(x_m - x_k) + (x_n - x_m)t}{(x_l - x_k)}, \quad t = \frac{(y_k - y_m) + (y_l - y_k)s}{(y_n - y_m)}$$

Solving these equations for  $s$  and  $t$  we find:

$$s = \frac{(x_n - x_m)(y_m - y_k) - (x_m - x_k)(y_n - y_m)}{(x_n - x_m)(y_l - y_k) - (x_l - x_k)(y_n - y_m)}$$

$$t = \frac{(x_l - x_k)(y_m - y_k) - (x_m - x_k)(y_l - y_k)}{(x_n - x_m)(y_l - y_k) - (x_l - x_k)(y_n - y_m)}$$

If both  $s$  and  $t$  are within the range  $0 \leq s, t \leq 1$  then the lines intersect; else the lines do not intersect.

If the line segments do intersect then the  $(x_i, y_i)$  point of intersection can be found from either of the two lines' parametric equations. Using line  $l_1(k, l)$ :

$$x_i = x_k + (x_l - x_k)s$$

$$y_i = y_k + (y_l - y_k)s$$

An implementation of the above procedure is presented in program LI\_LIB.CPP by means of a function called *ParametricIntersection()*:

```
// li_lib.cpp
// implementation of clock/anticlockwise
// and line-intersection functions
//...
Boolean ParametricIntersection (const double& xk,
                                const double& yk,
                                const double& xl,
                                const double& yl,
                                const double& xm,
                                const double& ym,
                                const double& xn,
                                const double& yn,
                                double& xi, double& yi)
{
    // reduce terms
    double xlk = xl - xk ;
    double xnm = xn - xm ;
    double xmk = xm - xk ;
    double ylk = yl - yk ;
    double ynm = yn - ym ;
```

---

```

double ymk = ym - yk ;

double denom = xnm*ylk - xlk*ynm ;

if (fabs (denom) < TOLERANCE) // parallel lines (denom=0)
    return FALSE ;
else
{
    double s = (xnm*ymk - xmk*ynm) / denom ;
    double t = (xlk*ymk - ylk*xmk) / denom ;

    if (s<0 || t<0 || s>1 || t>1) // no intersection
        return FALSE ;
    else // intersection
    {
        xi = xk + xlk*s ; // use either s or t
        yi = yk + ylk*s ; // pt. of intersection (use s)
        return TRUE ;
    }
}
}

```

The header file LI\_LIB.H contains the function declaration of *ParametricIntersection()*:

```

// li_lib.h
// header file for li_lib.cpp
//...
Boolean ParametricIntersection (const double& xk, /*...*/ ) ;

```

and the file LI.CPP tests the implementation:

```

// li.cpp
// intersection of two two-dimensional line segments

#include <iostream.h> // C++ I/O
#include "li_lib.h" // line intersection

void main ()
{
    // intersection of 2 lines:
    double l1x1, l1y1, l1x2, l1y2, l2x1, l2y1, l2x2, l2y2 = 0.0 ;

    // intersecting lines:
    l1x1 = 1.0 ; l1y1 = 1.0 ; l1x2 = 4.0 ; l1y2 = 2.0 ;
    l2x1 = 3.0 ; l2y1 = 5.0 ; l2x2 = 4.0 ; l2y2 = 0.0 ;
    //...
    Boolean intersect ; // intersection

    double xi, yi = 0.0 ;

```

```

Boolean intersect = ParametricIntersection
    (l1x1, l1y1, l1x2, l1y2,
     l2x1, l2y1, l2x2, l2y2, xi,
     yi) ;
if (intersect)
    cout << "lines intersect at point: (" 
        << xi << ", " << yi << ")" << endl ;
else
    cout << "lines do not intersect" << endl ;
//...
}

```

with output:

```
lines intersect at point: (3.625, 1.875)
```

The implementation of function *ParametricIntersection()* is fairly straightforward except for the test of parallel lines. The test case given in program LI.CPP is for the two lines shown in Fig. 6.3.

### 6.24.2 Bounding Boxes and Parametric Intersection

If the two line segments do not intersect then the function *ParametricIntersection()* is inefficient. The procedure of determining whether or not two lines intersect can be improved by introducing a *rejection* stage (Cormen *et al.*, 1990, pp.886–90). The rejection stage involves testing whether the two lines' respective bounding boxes intersect or not. The bounding box of a line segment is the smallest rectangle, with sides parallel to the *x*- and *y*-axes, that encompasses the line. Figure 6.5 illustrates the bounding boxes of lines  $l_1$  and  $l_2$ . Rectangle  $(k, l)$  is defined with lower left point  $\hat{k}(\hat{x}_k, \hat{y}_k)$  and upper right point  $\hat{l}(\hat{x}_l, \hat{y}_l)$ , where  $\hat{x}_k = \min(x_k, x_l)$ ,  $\hat{y}_k = \min(y_k, y_l)$ ,  $\hat{x}_l = \max(x_k, x_l)$  and  $\hat{y}_l = \max(y_k, y_l)$ . By experimenting with the two bounding boxes  $(k, l)$  and  $(m, n)$  it is found that intersection occurs only if the following test is logical-true:

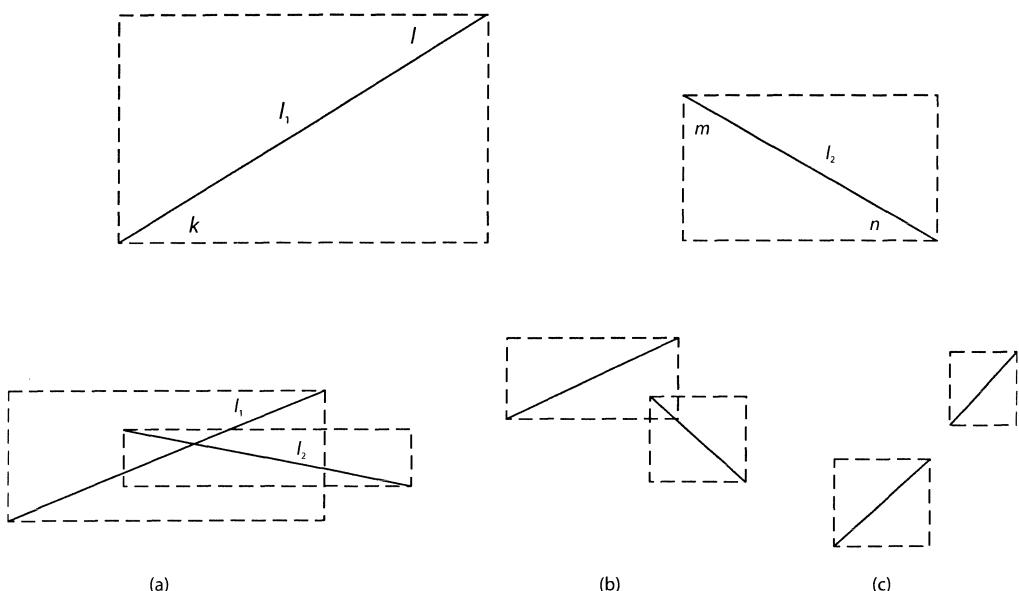
$$(\hat{x}_l \geq \hat{x}_m) \& \& (\hat{x}_n \geq \hat{x}_k) \& \& (\hat{y}_l \geq \hat{y}_m) \& \& (\hat{y}_n \geq \hat{y}_k)$$

The implementation of the line bounding box function is:

```

Boolean LineBBoxIntersection (const double& xk,
                            const double& yk,
                            const double& xl,
                            const double& yl,
                            const double& xm,
                            const double& ym,
                            const double& xn,
                            const double& yn)
{
    // line 11(k, l)
    double xk_hat = Min (xk, xl) ; double yk_hat = Min (yk, yl) ;
    double xl_hat = Max (xk, xl) ; double yl_hat = Max (yk, yl) ;
    // line 12(m, n)
    double xm_hat = Min (xm, xn) ; double ym_hat = Min (ym, yn) ;
    ...
}

```



**Fig. 6.5** Bounding boxes  $(\hat{k}, \hat{l})$  and  $(\hat{m}, \hat{n})$  of lines  $l_1(k, l)$  and  $l_2(m, n)$  respectively. (a) Intersecting lines, intersecting bounding boxes. (b) Non-intersecting lines, intersecting bounding boxes. (c) Non-intersecting collinear lines, non-intersecting bounding boxes.

```
double xn_hat = Max (xm, xn) ; double yn_hat = Max (ym, yn) ;

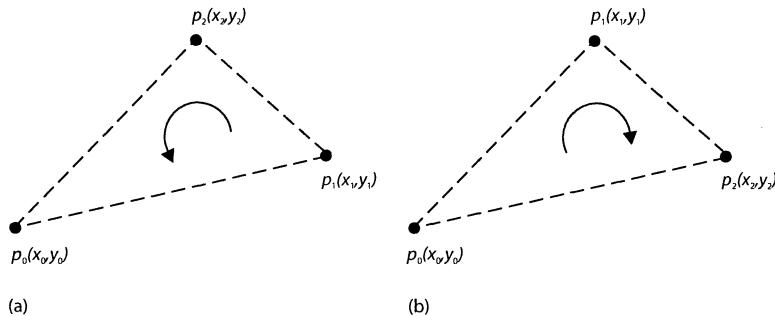
if (xl_hat >= xm_hat && xn_hat >= xk_hat &&
    yl_hat >= ym_hat && yn_hat >= yk_hat)
    return TRUE ;
return FALSE ;
}
```

*LineBBoxIntersection()* allows us now to perform a quick rejection test when testing whether two lines intersect:

```
//...
Boolean ParBBoxIntersection (const double& xk, /*...*/)
{
    // rejection test (line bounding boxes do not intersect)
    if (!LineBBoxIntersection (xk, yk, xl, yl, xm, ym, xn, yn))
        return FALSE ;
    //...
}
```

### 6.24.3 Clockwise or Anticlockwise?

Alternatively, it is possible to determine whether two lines intersect by considering the direction of rotation of three points (Cormen *et al.*, 1990, pp. 888–9; Sedgewick, 1988, pp. 349–51). Figure 6.6 illustrates two sets of points in a two-dimensional plane. On travelling from points  $p_0$  to  $p_1$  to  $p_2$  in Fig. 6.6(a) we turn anticlockwise, whereas the set of points  $(p_0, p_1, p_2)$  in Fig. 6.6(b) make a clockwise rotation.



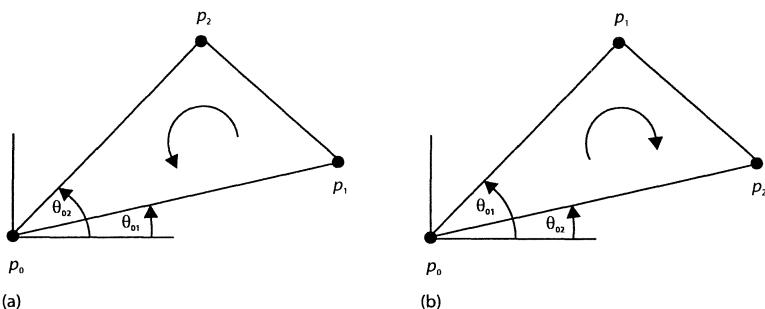
**Fig. 6.6** Direction of rotation of three points  $p_0, p_1$  and  $p_2$ . (a) Points  $(p_0, p_1, p_2)$  make an anticlockwise rotation. (b) Points  $(p_0, p_1, p_2)$  make a clockwise rotation.

There are several different techniques that can be applied to the problem of whether three points make a left or right turn. The first method considers the magnitude of the angles between a fixed global axis ( $x$ -axis in the present case) and the line segments  $(p_0, p_1)$  and  $(p_0, p_2)$  (Fig. 6.7). The following function implements this method:

```
int CACPolar (double x0, double y0, double x1, double y1,
               double x2, double y2)
{
    int cac = 0 ;

    // use point 0 as 'local' origin, compute polar angles
    double angle_01 = atan ((y1-y0)/(x1-x0)) ;
    double angle_02 = atan ((y2-y0)/(x2-x0)) ;

    if (angle_01 < angle_02)           // anti-clockwise
        cac = 1 ;
    else if (angle_01 > angle_02)     // clockwise
        cac = -1 ;
    else                                // collinear
        cac = 0 ;
    return cac ;
}
```



**Fig. 6.7** Direction of rotation of three points  $p_0, p_1$  and  $p_2$  in terms of the magnitude of the angles between the line segments  $(p_0, p_1)$  and  $(p_0, p_2)$  and the  $x$ -axis. (a) Anticlockwise rotation. (b) Clockwise rotation.

---

```
} // CACPolar()
```

Note that this function is far from perfect, since it does not account for vertical points ( $(x_1-x_0)=0$  or  $(x_2-x_0)=0$ ) or angles which are negative or greater than  $90^\circ$ . Also, the `atan()` function is a computationally expensive function to use for this simple function.

A more general method is to consider the determinant of the cross product of the two vectors<sup>5</sup>  $\mathbf{v}_{01}(p_1-p_0)$  and  $\mathbf{v}_{02}(p_2-p_0)$ :

$$|\mathbf{v}_{01} \times \mathbf{v}_{02}| = |(p_1 - p_0) \times (p_2 - p_0)| = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

If the determinant of the cross product of  $\mathbf{v}_{01}$  and  $\mathbf{v}_{02}$  is positive then points  $(p_0, p_1, p_2)$  make an anticlockwise rotation; if the cross product is negative then  $(p_0, p_1, p_2)$  make a clockwise rotation, and if the cross product is zero then the points  $(p_0, p_1, p_2)$  are collinear:

```
int CACCrossProduct (const double& x0, const double& y0,
                      const double& x1, const double& y1,
                      const double& x2, const double& y2)
{
    int cac = 0;

    double cp = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0);

    if (cp > 0)           // anticlockwise
        cac = 1;
    else if (cp < 0)     // clockwise
        cac = -1;
    else                  // collinear
        cac = 0;
    return cac;
} // CACCrossProduct()
```

The *norm*<sup>6</sup> of the cross product  $\mathbf{v}_{01} \times \mathbf{v}_{02}$  is equal to the area of the parallelogram,  $A(p_0, p_1, p_2, p_1+p_2)$ , determined by  $\mathbf{v}_{01}$  and  $\mathbf{v}_{02}$  (Fig. 6.8). The area  $A$  is given by:

$$A(p_0, p_1, p_2, p_1 + p_2) = \|\mathbf{v}_{01}\| \|\mathbf{v}_{02}\| \sin\theta = \|\mathbf{v}_{01} \times \mathbf{v}_{02}\|$$

The signed (+, -) area of the parallelogram indicates the direction of rotation of points  $(p_0, p_1, p_2)$ .

Alternatively, we can examine the signed area of a triangle formed by the three vertices  $(p_0, p_1, p_2)$  (Fig. 6.9). The area of the triangle  $(i, j, k)$  is determined from the area of the three trapezia  $(i, l, m, k), (k, m, n, j)$  and  $(i, l, n, j)$ :

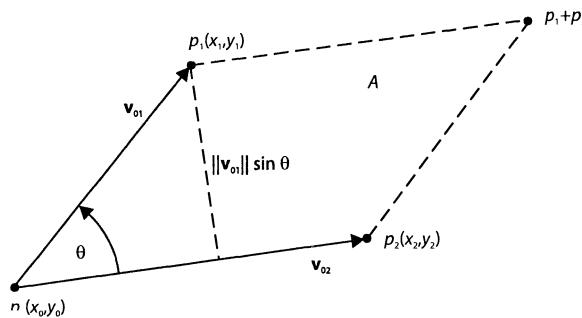
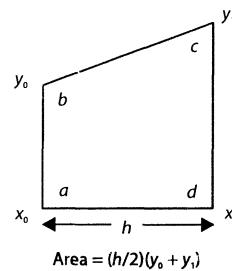
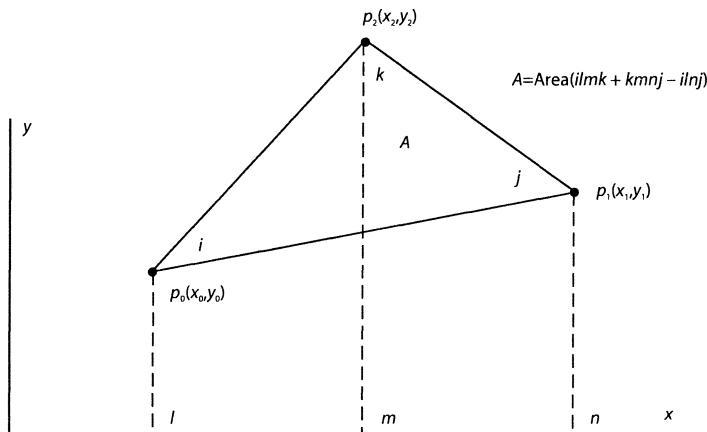
$$A(i, j, k) = \text{Area } (ilmk + kmnj - ilnj)$$

where

---

5 An excellent introduction to the properties of vectors and linear algebra can be found in Anton (1991).

6 The norm of a vector  $\mathbf{v}(v_1, v_2, v_3)$  is the length or magnitude of  $\mathbf{v}$  and is denoted by  $\|\mathbf{v}\|$ . It follows by Pythagoras' theorem that  $\|\mathbf{v}\| = (v_1^2, v_2^2, v_3^2)^{1/2}$ .

**Fig. 6.8** Area of a parallelogram ( $p_0, p_1, p_2, p_1 + p_2$ ).**Fig. 6.9** Area of a triangle ( $p_0, p_1, p_2$ ) and a trapezium ( $a, b, c, d$ ).

$$A(i, l, m, k) = \frac{(x_2 - x_0)}{2}(y_0 + y_2); \quad A(k, m, n, j) = \frac{(x_1 - x_2)}{2}(y_1 + y_2);$$

$$A(i, l, n, j) = \frac{(x_1 - x_0)}{2}(y_0 + y_1)$$

Hence:

$$A(i, j, k) = \frac{1}{2}(x_0 y_1 + x_1 y_2 + x_2 y_0 - x_0 y_2 - x_1 y_0 - x_2 y_1)$$

The area,  $A$ , of triangle ( $p_0, p_1, p_2$ ) is therefore the following determinant:

$$A(p_0, p_1, p_2) = \frac{1}{2} \begin{vmatrix} (x_j - x_i) & (x_k - x_i) \\ (y_j - y_i) & (y_k - y_i) \end{vmatrix} = \frac{1}{2} \begin{vmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{vmatrix}$$

If the determinant is positive then  $(p_0, p_1, p_2)$  make an anticlockwise rotation. If the determinant is negative then  $(p_0, p_1, p_2)$  make a clockwise rotation. If the determinant is zero then  $(p_0, p_1, p_2)$  are collinear. The function `CACSignedAreaOfTriangle()` given in LI.CPP implements the method outlined.

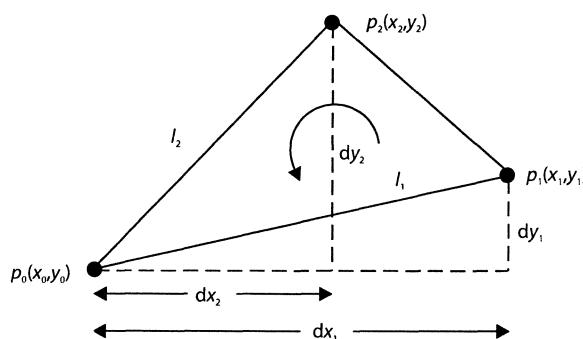
The final algorithm that we will consider is taken from Sedgewick (1988, pp. 349–51); refer also to Sedgewick (1992). This algorithm is particularly elegant and allows a simple implementation of a line intersection function. Figure 6.10 illustrates a set of three points  $(p_0, p_1, p_2)$  with  $p_0$  representing the *local origin*. The gradients of lines  $l_1(p_0, p_1)$  and  $l_2(p_0, p_2)$  are denoted by  $m_{01}(=dy_1/dx_1)$  and  $m_{02}(dy_2/dx_2)$  respectively. If  $m_{02} > m_{01}$  then the direction of rotation is anticlockwise; if  $m_{01} < m_{02}$  the direction of rotation is clockwise; and if  $m_{01} = m_{02}$  the three points are collinear. To prevent division by zero, if either  $dx_1$  or  $dx_2$  is zero the inequality  $m_{02} > m_{01}$  is multiplied by  $dx_1 dx_2$ :

$$m_{02} > m_{01} \Rightarrow \frac{dy_2}{dx_2} > \frac{dy_1}{dx_1} \Rightarrow dx_1 dy_2 > dx_2 dy_1$$

The function implementation is given below:

```
int CAC (const double& x0, const double& y0,
          const double& x1, const double& y1,
          const double& x2, const double& y2)
{
    int cac = 0 ; // return var. indicating direction

    // use point 0 as 'local' origin
    double dx1 = x1 - x0 ; double dy1 = y1 - y0 ;
    double dx2 = x2 - x0 ; double dy2 = y2 - y0 ;
    // eliminate infinite gradients by multiplying gradients
    // between points 0-1 and 0-2 by (dx1*dx2)
```



**Fig. 6.10** Three points  $(p_0, p_1, p_2)$ . The gradients of lines  $l_1$  and  $l_2$  are  $m_{01}$  ( $=dy_1/dx_1$ ) and  $m_{02}$  ( $=dy_2/dx_2$ ) respectively. If  $m_{02} > m_{01}$  then  $(p_0, p_1, p_2)$  make an anticlockwise rotation. If  $m_{01} > m_{02}$  then  $(p_0, p_1, p_2)$  make a clockwise rotation. If  $m_{01} = m_{02}$  then  $(p_0, p_1, p_2)$  are collinear.

```

if (dx1*dy2 > dx2*dy1) // anti-clockwise (m02>m01)
    cac = 1 ;

if (dx1*dy2 < dx2*dy1) // clockwise (m02<m01)
    cac = -1 ;

if ((dx1*dy2 - dx2*dy1) < TOLERANCE) // same gradients
{
    if ((dx1*dx2 < 0) || (dy1*dy2 < 0))
        cac = -1 ;
    else if ((dx1*dx1 + dy1*dy1) >= (dx2*dx2 + dy2*dy2))
        cac = 0 ;
    else
        cac = 1 ;
}
return cac ;
}

```

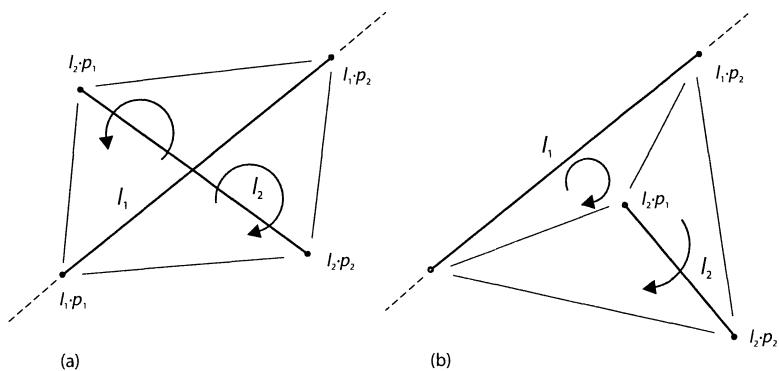
Rather than designing the *CAC()* function to return either logical-true or false to indicate the direction of rotation, the function returns one of three values: -1, 1 or 0. *CAC()* returns 1 or -1 if the three points make an anticlockwise or clockwise order, respectively. If the points are collinear then *CAC()* returns -1 if  $p_0$  is between  $p_1$  and  $p_2$ , 1 if  $p_1$  is between  $p_0$  and  $p_2$  and 0 if  $p_2$  is between  $p_0$  and  $p_1$ . Adopting this convention allows a simple implementation of the line intersection function.

So how do we apply the notion of rotating either clockwise or anticlockwise from one point to a second to a third to the intersection of two lines? Figure 6.11 illustrates two sets of lines. Figure 6.11(a) shows two lines,  $l_1$  and  $l_2$ , intersecting, whereas  $l_1$  and  $l_2$  in Fig. 6.11(b) do not intersect. If  $l_1$  and  $l_2$  intersect then the direction of rotation reverses when we consider the three points  $(l_1.p_1, l_1.p_2, l_2.p_1)$  and  $(l_1.p_1, l_1.p_2, l_2.p_2)$  separately, using  $l_1$  as a reference line. The exception to the rule is shown in Fig. 6.12, with reference line  $l_1$ . In this case the directions of rotation are reversed, but the two lines do not intersect. However, if we also use  $l_2$  as a reference line then the points  $(l_2.p_1, l_2.p_2, l_1.p_1)$  and  $(l_2.p_1, l_2.p_2, l_1.p_2)$  are in the same order. Therefore, the intersection algorithm should compare the directions of rotation with lines  $l_1$  and  $l_2$  as reference lines. The implementation of the intersection function is:

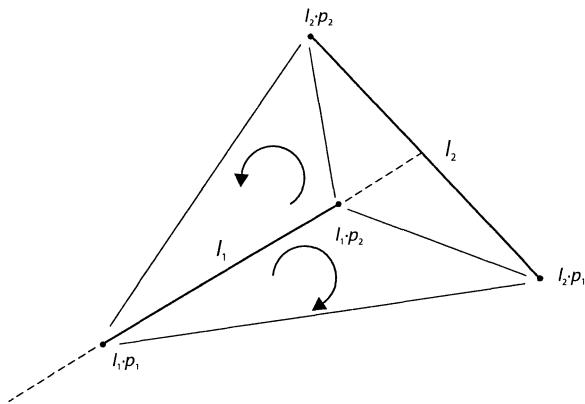
```

Boolean LineIntersection (const double& l1x1,
                           const double& l1y1,
                           const double& l1x2, const double& l1y2,
                           const double& l2x1, const double& l2y1,
                           const double& l2x2, const double& l2y2)
{
    // intersection
    if ( ((CAC (l1x1, l1y1, l1x2, l1y2, l2x1, l2y1) *
              CAC (l1x1, l1y1, l1x2, l1y2, l2x2, l2y2)) <= 0) &&
          ((CAC (l2x1, l2y1, l2x2, l2y2, l1x1, l1y1) *
              CAC (l2x1, l2y1, l2x2, l2y2, l1x2, l1y2)) <= 0) )
        return TRUE ;
    // no intersection
    else
        return FALSE ;
}

```



**Fig. 6.11** (a) Intersecting lines. (b) Non-intersecting lines.



**Fig. 6.12** Non-intersecting lines.

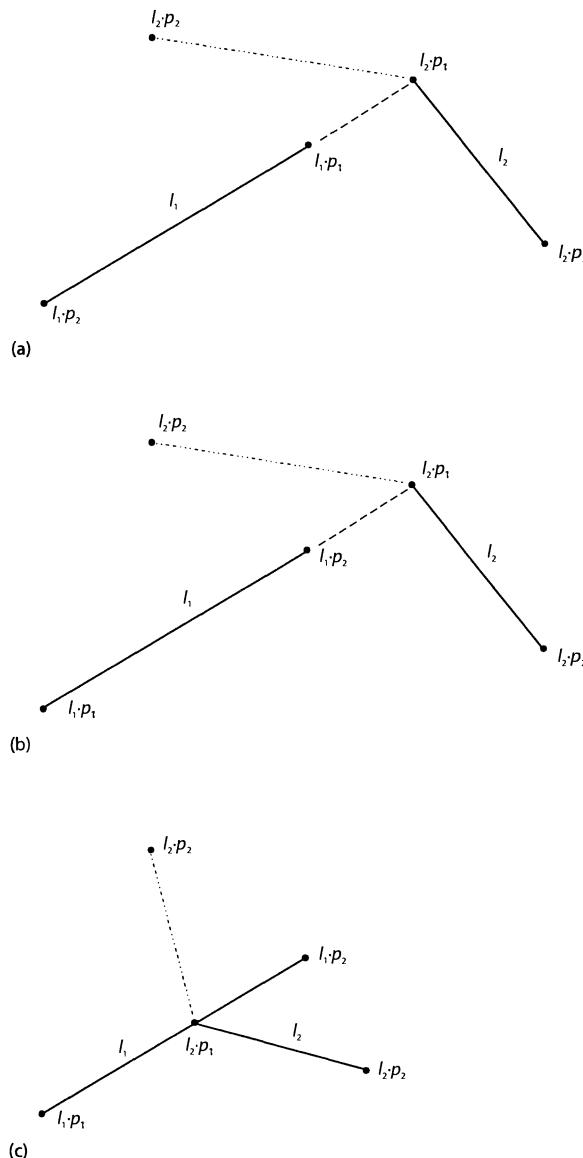
Although the functions `CAC()` and `LineIntersection()` are relatively simple, they are nevertheless very powerful and able to deal with a variety of line intersections and non-intersections; see Fig. 6.13.

We shall return to the problem of line segment intersection in Chapter 9 when we introduce the C++ **class** and member functions.

## 6.25 Summary

We have dealt with the key elements of a function in this chapter. Functions are a cornerstone of programming in that they help a programmer to conceptualise, break down and organise a large program. Functions are reusable, have their own local scope and hide their implementation details from a function-caller.

The function definition is the body of a function, or a function's implementation. The function declaration or prototype is the signature of the function and indicates the function's name, return type specifier and number and type of arguments.

**Fig. 6.13** Line intersections and non-intersections.

Functions can be overloaded. Function overloading is an example of compile-time polymorphism in which a single function name can have multiple different operations. We shall continue the discussion of function overloading and, in particular, run-time polymorphism, when we discuss inheritance and **virtual** functions in Chapter 15.

Functions can have default arguments, so that functions can be called without passing all of the arguments. Default arguments are particularly useful for functions with large argument lists. We have also seen that functions can be **inline**. Inline functions execute faster and are generally restricted to small functions that are frequently called.

We have seen that arguments can be passed to a function either by value or by reference. Passing by value makes a copy of the argument, whereas passing by reference does not make a copy but accesses the caller-argument. Reference arguments can be passed as **const** to ensure that the function-caller argument is not altered during the function call.

C++ compilers are shipped with extensive function and **class** libraries, some of which are ANSI standard-compatible and some of which are not. For instance, the Borland C++ compiler supplies its own Container and ObjectWindows classes, whereas the Microsoft Visual C++ compiler supports the Microsoft Foundation Classes (MFC) library. Spend some time getting to know the libraries – you could save yourself a great deal of blood, sweat and tears by not developing a routine that already exists.

## Exercises

- 6.1 Within the *main()* function add together the global and *main()* *g* variables for the following program:

```
int g = 1 ;

void main ()
{
    int g = 2 ;
    //...
}
```

- 6.2 Develop two functions that are passed a character and return an integer value indicating whether the character is lowercase or uppercase. The signature of the two functions should be of the following form:

```
int IsLowerCase (char c) ;
int IsUpperCase (char c) ;
```

Also, develop two functions which are passed a character and which will convert the character to lowercase and uppercase and have the following signatures:

```
char ToLowerCase (char c) ;
char ToUpperCase (char c) ;
```

- 6.3 Write two functions which convert from degrees to radians and vice versa. Place the function declarations in a header file MY\_MATH.H and the function implementations in MY\_MATH.CPP. Test your functions by including MY\_MATH.H in a test program.

- 6.4 Develop three **inline** overloaded *Abs()* functions for types **int**, **float** and **double** which return the absolute value of a number.

- 6.5 Implement a function called *FractionalPart()* which returns the fractional part of a floating-point number. The return value should always be positive. In your implementation of *FractionalPart()* make use of the C++ library function *fmod()*:

```
double fmod (double x, double y) ; // MATH.H
```

which returns *x* modulo *y* and is the same sign as *x*.

- 6.6 Develop a function that models the following damped, rectified sine curve:

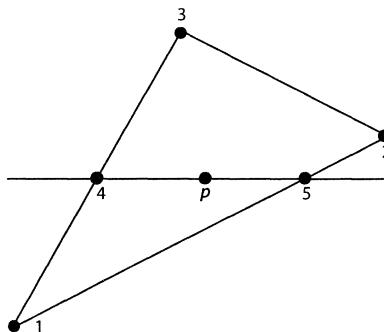
$$y(x) = A |\sin(\omega x + \theta_0)| e^{-kx}$$

where  $A$  is the initial amplitude,  $\omega$  is the angular frequency,  $\theta_0$  is the phase angle and  $k$  is the damping constant.

- 6.7 Develop a function which performs interpolation using the Gouraud shading algorithm for a two-dimensional ( $x,y; z=0$ ) straight-sided triangle. The signature of the function should be of the form:

```
int GouraudInterpolation (double i1, double i2, double i3,
double x1, double y1,
double x2, double y2,
double x3, double y3,
double px, double py,
double& ip) ;
```

where  $i_1, i_2$  and  $i_3$  are the intensities at the triangle vertices 1, 2 and 3 whose ( $x,y$ ) coordinates are denoted by  $(x_1,y_1)$ ,  $(x_2,y_2)$  and  $(x_3,y_3)$  respectively, and  $(px,py)$  denotes a point,  $p$ , which is interior to the triangle with interpolated intensity  $ip$ :



The intensity at point 4 is linearly interpolated from the intensities at vertices 1 and 3 and the intensity at point 5 is linearly interpolated from the intensities at vertices 1 and 2. The intensity at point  $p$  is linearly interpolated from the intensities at points 4 and 5. The Gouraud shading algorithm can be found in several computer graphics textbooks, and a typical reference is Hearn and Baker (1994, pp. 523–5).

# Arrays

Arrays are aggregates of a given fundamental or user-defined data type. An array element is accessed by an index, rather than by name, within a consecutive list of elements. Arrays are the basic tool for handling large amounts of similar data.

The elements of an array can be of any valid type, such as `int`, `double` or a user-defined type. We shall see that an array of characters defines a string. Arrays can be of any dimension, although one- and two-dimensional arrays are the most frequently used when programming.

C++ arrays are frequently referred to as static because their size is set at the time of compilation and remains fixed throughout the lifetime of a program. Irrespective of the dimension of an array, the elements of an array are stored contiguously in memory, and this means that accessing the elements of an array is random; i.e. it takes the same time to access the last element as to access the first element.

Arrays of variable size are possible and frequently used when programming in C++, but they require the use of pointers, which will be discussed in Chapter 12.



## 7.1 An Array

Let's begin our introduction to arrays with a simple program:

```
// array.cpp
// illustrates a simple one-dimensional array
#include <iostream.h> // C++ I/O

void main ()
{
    // define a 1D array of 5 elements of type int
    int array[5] ;

    // get numbers
    for (int i=0; i<5; i++)
```

```

{
    cout << "enter an integer number: " ;
    cin  >> array[i] ;
}
cout << endl ;
// output numbers
for (int j=0; j<5; j++)
{
    cout << "number " << (j+1) << ":" << array[j] << endl ;
}
}

```

Here's some user interaction:

```

enter an integer number: 1
enter an integer number: 12
enter an integer number: 23
enter an integer number: 34
enter an integer number: 45

number 1: 1
number 2: 12
number 3: 23
number 4: 34
number 5: 45

```

So, what is an array? An array is a group of values or objects of *identical* fundamental or user-defined data types in which each *element* of the array can be referenced as a single unit. Note that all the elements of an array are of the same type. In the program above, an array, `array`, is defined of type `int` which has five elements:

```
int array[5] ;
```

A `for`-loop is used to prompt the user to enter five integer numbers. Each number is assigned to be a different, unique element of the array:

```
//...
cin  >> array[i] ;
```

When all of the numbers (five in total) have been entered the program outputs the entered numbers to the screen:

```
//...
cout << "number " << (j+1) << ":" << array[j] << endl ;
```

The general syntax of an array is:

```
typeSpecifier arrayName[number_of_elements] ;
```

`typeSpecifier` is the type of each of the elements of the array (e.g. a fundamental data type, such as `int` or `double`, or a user-defined type, such as `Complex` or `Trian-`

g1e). The name of the array is denoted by `array_name`, and the usual syntax of defining identifiers also applies to the naming of arrays. The number of elements in an array can be as small or as large (within limits) as the programmer specifies and is denoted by `number_of_elements` within the square brackets [ and ]. The array size must be supplied since there is no default array size in C++. Defining an uninitialised array without a size is illegal in C++.

## 7.2 The Size of an Array

The size of the array defined above is fixed at compile time to be equal to five elements of type `int` and is thus referred to as a *fixed-size* or *static* array. A common confusion when defining an array is to use a variable array size. The following program illustrates an array defined as above, but now the array size is determined at run-time by the user:

```
// static.cpp
// illustrates that a C++ array must be defined constant
#include <iostream.h> // C++ I/O

void main ()
{
    int array_size ;

    cout << "enter an array size: " ;
    cin  >> array_size ;

    // define a 1D array of a size determined by a user
    // error: const. expression required.
    int array[array_size] ; // error: const. expression
    //...
}
```

This program will not compile because the array size must be a constant and defined at compilation, since the compiler has to determine how much memory to allocate for the array. In Chapter 12, we shall examine *dynamic* arrays, whose size can vary at run-time, but for now we shall restrict ourselves to static arrays.

Rather than defining the array as:

```
int array[5] ;
```

we could define the array size as a constant identifier:

```
int const ARRAY_SIZE = 5 ; // array size
//...
int array[ARRAY_SIZE] ;
```

which increases the generality of a program.

In conclusion, once the size of an array has been set and the program compiled, the array size is fixed for the entire lifetime of the program.

## 7.3 The **sizeof** Operator

Remember the **sizeof** operator introduced in Chapter 4? We noted that the **sizeof** operator is a unary operator and thus has one operand. For an identifier the syntax is:

```
sizeof operand
```

and for a type specifier:

```
sizeof (type_specifier)
```

A type specifier must be enclosed in parentheses, but the parentheses are optional for an identifier. The **sizeof** operator can also be used to determine the size, in bytes, of an array. The following program, A\_SIZE.CPP, illustrates determining the size of several arrays:

```
// a_size.cpp
// illustrates the sizeof operator for determining
// the number of elements and size of an array.
#include <iostream.h> // C++ I/O

void main ()
{
    char    c_array[10] ;
    int     i_array[10] ;
    float   f_array[10] ;
    double  d_array[10] ;

    cout << "sizeof c_array[]: " << sizeof c_array << endl ;
    cout << "sizeof i_array[]: " << sizeof i_array << endl ;
    cout << "sizeof f_array[]: " << sizeof f_array << endl ;
    cout << "sizeof d_array[]: " << sizeof d_array << endl ;

    double big_array[50][50] ;

    cout << endl
        << "sizeof big_array[][]: "
        << sizeof big_array << endl ;

    // no. of elem.= (total array size) / (size of single
    //           element)
    cout << "number of elements in big_array[][]: "
        << sizeof big_array / sizeof big_array[0][0] ;
}
```

and generates the output:

```
sizeof c_array[]: 10
sizeof i_array[]: 20
sizeof f_array[]: 40
sizeof d_array[]: 80
```

---

```
sizeof big_array[][]: 20000
number of elements in big_array[][]: 2500
```

which outputs sizes of 10, 20, 40 and 80 bytes for four arrays with 10 elements of type **char**, **int**, **float** and **double** respectively. Also, the output displays the size of a large two-dimensional array (50×50), `big_array`, which eats up 20 kbyte of memory. The program outputs the total number of elements in the array by evaluating the expression:

```
sizeof big_array / sizeof big_array[0][0] ;
```

which divides the total storage required for the array by the storage requirements of a single element. In the present case the single element chosen was the first element, although any element could be chosen since all the elements of an array are of the same type. This technique of determining the number of elements in an array is frequently employed, since it is more general than:

```
cout << "number of elements in big_array[][]: 2500" ;
```

For instance, if we now changed the definition of `big_array` and recompiled the program the correct number of elements would still be displayed if we use the **sizeof** operator.

## 7.4 Array Indexing

### 7.4.1 First Element

You may have noticed from the above program (ARRAY.CPP) that both **for**-loops begin indexing the array of numbers entered from zero and increase incrementally from  $i=0$  to  $i=4$ , i.e. `array[0]`, `array[1]`, ..., `array[4]`. C++ arrays start indexing from element 0. The majority of other programming languages start array indexing from element 1. Also, for the majority of people, it is intuitive to begin counting with 1, so array indexing from 0 is a common source of confusion with arrays in C++. So, let's have a note:

*C++ arrays start indexing from element 0*

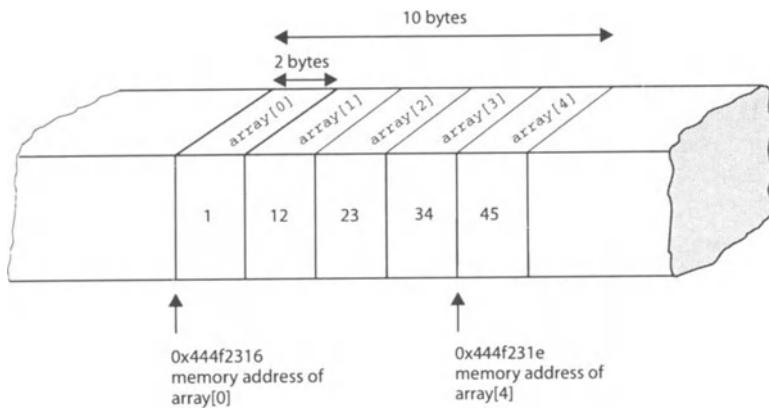
Figure 7.1 schematically (as a block of ice-cream) illustrates the ‘numbers’ array of the ARRAY.CPP program stored consecutively somewhere in memory. Element 0 of the array is the value 1, element 2 is value 12 and so on. The array elements are stored in memory adjacent to one another. Since each element of the array is of type **int** they occupy two bytes of memory. There are five elements in the array, so the total memory allocated to the array is ten bytes.

### 7.4.2 Setting and Getting an Element

In ARRAY.CPP the numbers entered by a user are inserted into `array` by the statement:

```
cin >> array[i] ;
```

which involves the `cin` input stream object and the extraction operator (`>>`). The elements of `array` are extracted by the statement:



**Fig. 7.1** An array somewhere in memory.

```
cout << "number " << (j+1) << ":" << array[j] << endl ;
```

Thus an element of an array is indexed by the array name and the element index number.

### 7.4.3 Last Element

C++ does not have array bound checking. Let's say that again:

*There is no bound checking in C++*

When accessing the elements of an array, C++ assumes that you know what you are doing. Let's look at a program which indexes elements outside of the memory allocated for an array:

```
// indexing.cpp
// illustrates indexing an array out of bounds
#include <iostream.h> // C++ I/O

void main ()
{
    // define a 1D array of 5 elements of type int
    int array[5] ;

    // get numbers
    for (int i=0; i<5; i++)
    {
        cout << "enter an integer number: " ;
        cin >> array[i] ;
    }
    cout << endl ;
    // out of bounds
    for (int j=0; j<10; j++)
    {
        cout << "number " << (j+1) << ":" << array[j] << endl ;
    }
}
```

```

    }
}

```

and some user interaction:

```

enter an integer number: 1
enter an integer number: 2
enter an integer number: 3
enter an integer number: 4
enter an integer number: 5

```

```

number 1: 1
number 2: 2
number 3: 3
number 4: 4
number 5: 5
number 6: 16567
number 7: 8931
number 8: 13265
number 9: 17487
number 10: 1

```

The output illustrates that although we only defined an array of five elements, we are able to index elements `array[5]` to `array[9]`, which are filled with garbage. The compiler does not guarantee that out-of-bounds array elements are initialised to zero or any other value. They are undefined except for the index beyond the high end of the array.

Stepping out of bounds isn't as common as you might at first expect, but bound checking becomes more important when dealing with dynamic arrays, which we will see later.

## 7.5 Initialisation of One-Dimensional Arrays

Just as we can initialise a variable when it is defined, we can similarly initialise arrays when they are defined:

```

// initial.cpp
// illustrates initialising arrays
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...

void main ()
{
    const int NUM_BOOKS = 10 ;
    float computer_book_prices[NUM_BOOKS] =
    { 18.99, 27.45, 15.95, 26.95, 21.95,
      24.95, 29.95, 30.50, 16.95, 26.95 } ;

    float total_price = 0.0 ;

    for (int i=0; i<NUM_BOOKS; i++)

```

```

    {
    total_price += computer_book_prices[i] ;
    }

    cout << setiosflags (ios::fixed)
    << setprecision (2)
    << "total price: " << total_price << endl
    << "average price of my 10 favourite computer books:
    << total_price / NUM_BOOKS ;
}

```

which displays to the screen the total and average prices of ten books.

The above program defines an array, `computer_book_prices`, whose elements are the retail prices (£) of my 10 favourite computer books. The individual elements of the array are separated by the comma operator. This list of initial values for the array elements is referred to as the *initialiser list*. The initialiser list is enclosed between the opening, {, and closing, }, braces, not forgetting the semicolon indicating the end of the statement. The general syntax for initialising an array is:

```
typeSpecifier arrayName[numElements] { elem0, elem1, ...
                                         elemn };
```

It is worth noting that in the initialisation of the one-dimensional array in the program INITIAL.CPP it is not necessary to define the array size explicitly as 10. The following is equally valid and legal:

```
float computer_book_prices[] =
{ 18.99, 27.45, 15.95, 26.95, 21.95,
  24.95, 29.95, 30.50, 16.95, 26.95 } ;
```

The compiler is able to work out for itself how many elements are in the array simply by counting the number of initialisers in the initialiser list.

## 7.6 Two-Dimensional Arrays

Let's take a look at a two-dimensional array of integers:

```
// array_2d.cpp
// illustrates two-dimensional arrays
#include <iostream.h> // C++ I/O

void main ()
{
const int ROWS = 2 ; // no. of rows and columns
const int COLS = 5 ;

// define & initialise a 2D array
int array_2d[ROWS][COLS] = { {0, 1, 2, 3, 4},
                             {5, 6, 7, 8, 9} };
```

```
// O/P nested for loop
for (int i=0; i<ROWS; i++)
{
    for (int j=0; j<COLS; j++)
    {
        cout << array_2d[i][j] << " ";
    }
    cout << endl ; // newline after each row
}
}
```

which generates the output:

```
0 1 2 3 4
5 6 7 8 9
```

A two-dimensional array is defined by the following general syntax:

```
typeSpecifier arrayName[rows][columns] ;
```

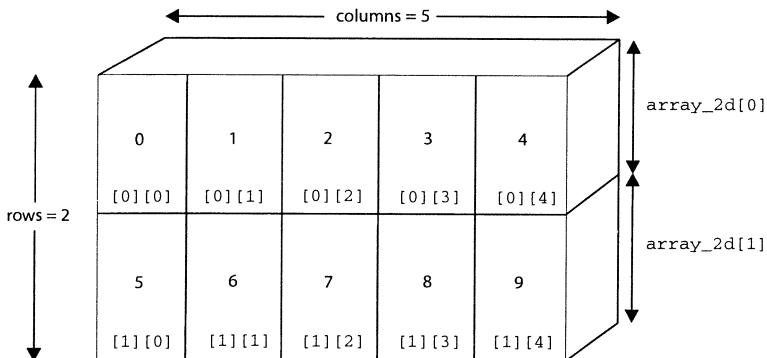
and allocates sufficient memory for (rows by columns) values of type `typeSpecifier`. Note that indexing the elements of an array is (row by column) rather than (column by row).

Figure 7.2 illustrates the array `array_2d` in the above program. To index the integer 7 we access row 1 and column 2; namely `array_2d[1][2]`.

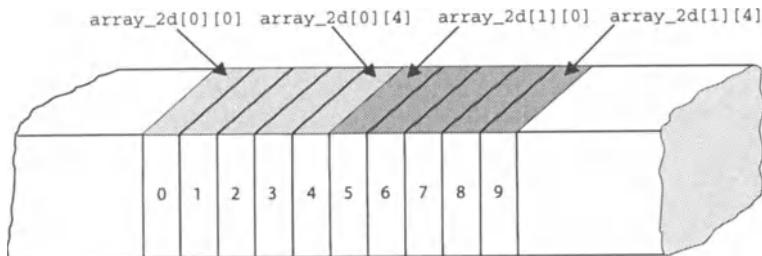
Pay particular attention to the use of the nested `for`-loop in order to access the elements of a two-dimensional array correctly. Note also the placement of the output stream object `cout` and the new line manipulator `endl` to output the array as a  $(2 \times 5)$  array rather than displaying all of the array elements on a single line.

Visualising two-dimensional arrays can be difficult at first, but provided you remember the (row by column) rule then, given time, the penny should drop!

Although it is convenient for us to visualise two-dimensional arrays as a (row by column) table of values, the values are in fact stored linearly in memory, as shown in Fig. 7.3. The compiler generates an equivalent pointer referencing system to access the array elements. The array subscript operator `[ ]` is used for user clarity. When pointers are discussed later, the equivalence between array indexing by means of the subscript operator, `[ ]`, and by a pointer, `*`, will become clearer.



**Fig. 7.2** A two-dimensional array of integers.



**Fig. 7.3** Memory representation of a two-dimensional array.

## 7.7 Initialisation of Two-Dimensional Arrays

The program above, ARRAY2D.CPP, also demonstrates initialisation of two-dimensional arrays:

```
int array_2d[ROWS][COLS] = { {0, 1, 2, 3, 4},
                             {5, 6, 7, 8, 9} } ;
```

Each element of each row of the array is separated by the comma operator, just as in a one-dimensional array. The difference between initialisation of one- and two-dimensional arrays is the use of the inner braces to indicate explicitly each row of the array. Imagine if the inner braces were not present:

```
int array_2d[ROWS][COLS] = { 0, 1, 2, 3, 4,
                            5, 6, 7, 8, 9 } ;
```

Clearly, the compiler would view the right-hand side of the expression as the initialisation of a one-dimensional array. However, if you have a clever compiler, such as the Borland C++ (version 5.0) compiler, such an array definition and initialisation will compile without an error message, but will generate a warning message of the form ‘array initialisation is partially bracketed’.

As with one-dimensional arrays it is possible to neglect the row dimension of an array – the compiler is clever enough to figure out what you mean. The contrary, neglecting the column but including the row dimension, is not true:

```
int array_2d[][COLS] = { {0, 1, 2, 3, 4},
                        {5, 6, 7, 8, 9} } ; // OK

int array_2d[ROWS][] = { {0, 1, 2, 3, 4},
                        {5, 6, 7, 8, 9} } ; // NOT OK

int array_2d[][] = { {0, 1, 2, 3, 4},
                     {5, 6, 7, 8, 9} } ; // definitely
                           // NOT OK
```

However, always try to indicate your intentions as explicitly as possible (for programming languages anyway) and define both the row and column dimensions.

## 7.8 How are Two-Dimensional Arrays Used?

Two-dimensional arrays have several applications. One of the more well-known applications is the solution of systems of simultaneous linear equations. A system of  $m$  simultaneous linear equations in  $n$  unknowns  $x_1, x_2, \dots, x_n$  is of the form:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

 $\vdots$ 

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m$$

where the coefficients  $a_{ij}$  and  $b_i$  are known values. This system of linear equations can be written as a single vector equation:

$$\mathbf{Ax} = \mathbf{b}$$

where the coefficient matrix  $= [a_{ij}]$  is an  $m \times n$  matrix. The column vector  $\mathbf{b}$  represents the known right-hand side values and  $\mathbf{x}$  is the solution column vector:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

For example, the system of equations

$$2x + y + z = 1$$

$$x + 4y + 6z = 2$$

$$3x + 8y + z = 1$$

can be written in matrix form as:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 4 & 6 \\ 3 & 8 & 1 \end{bmatrix}, \quad \mathbf{x} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix}, \quad \mathbf{b} = \begin{Bmatrix} 1 \\ 2 \\ 1 \end{Bmatrix}$$

The method of solution of such a system of equations can be either by Cramer's rule for small systems, direct methods, such as Gaussian elimination, or iterative methods, such as Gauss–Siedel iteration. Chapter 12 presents an implementation of the Gaussian elimination method.

## 7.9 Three- and Higher-Dimensional Arrays

Arrays of dimensions higher than two are possible in C++ and are of the general form:

```
typeSpecifier arrayName[dimension1]...[dimensionN] ;
```

A three-dimensional array is defined in the program below:

```
// array_3d.cpp
// illustrates three-dimensional arrays
#include <iostream.h> // C++ I/O

void main ()
{
    const int X_SIZE = 2 ; // no. of rows, columns, depth
    const int Y_SIZE = 2 ;
    const int Z_SIZE = 2 ;

    // define & initialise a 3D array
    int array_3d[X_SIZE][Y_SIZE][Z_SIZE] =
        { {{10, 11}, {12, 13}},
          {{20, 21}, {22, 23}} } ;

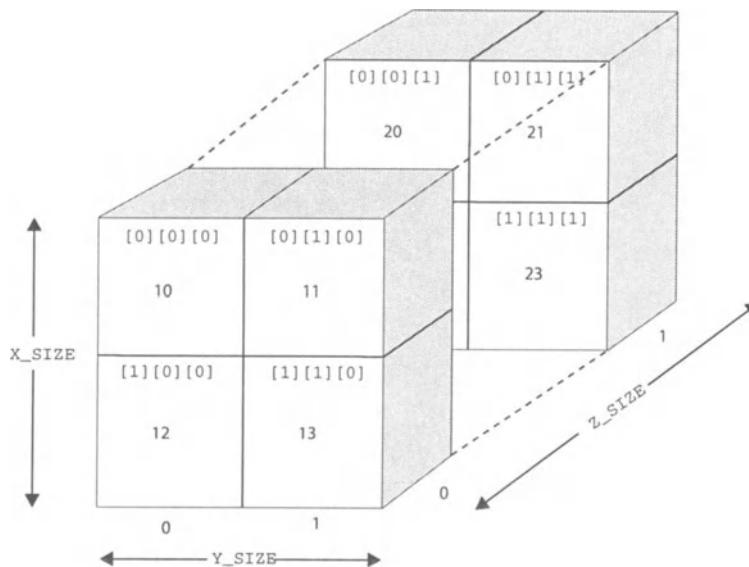
    // O/P nested loop
    for (int i=0; i<X_SIZE; i++)
    {
        for (int j=0; j<Y_SIZE; j++)
        {
            for (int k=0; k<Z_SIZE; k++)
            {
                cout << array_3d[i][j][k] << " " ;
            }
            cout << endl ;
        }
        cout << endl ;
    }
}
```

which defines and initialises a three-dimensional array of 8 elements of type **int**. The output of the program is:

```
10 11
12 13
```

```
20 21
22 23
```

which displays the ‘front’ layer of elements first and then the ‘back’ layer. Clearly, since the display screen is two-dimensional, there is a problem displaying three-dimensional arrays! To assist our visualisation of the above array, **array\_3d**, Fig. 7.4 illustrates an exploded view of



**Fig. 7.4** An exploded three-dimensional array.

the three-dimensional array. Note the addition of the third **for**-loop to access the third-dimensional elements of the array and the addition of braces in the array initialiser list to indicate dimensions 1, 2 and 3.

Three-dimensional and higher order arrays are more difficult to comprehend than are two-dimensional arrays, and are consequently rarely used in practice.

## 7.10 Strings

Strings are one-dimensional arrays of type **char**.

Strings, like other types, can be either constant or variable. Let's start by examining variable strings.

### 7.10.1 Variable Strings

Let's maintain our usual format of introducing a new topic with the aid of a program:

```
// v_string.cpp
// illustrates variable strings
#include <iostream.h> // C++ I/O

void main ()
{
    // string variable: array of char
    char name[10] ;

    cout << "enter your forename: " ;
```

```

    cin >> name ;

    cout << "hello " << name ;
}

```

The definition of the variable string, name, is:

```
char name[10] ;
```

Some output:

```

enter your forename: Maharg
hello Maharg

```

which defines name to be an array of 10 characters, each occupying one byte of memory. Things look the same as with the program ARRAY.CPP for integer arrays, except for one important difference: the *null termination character*.

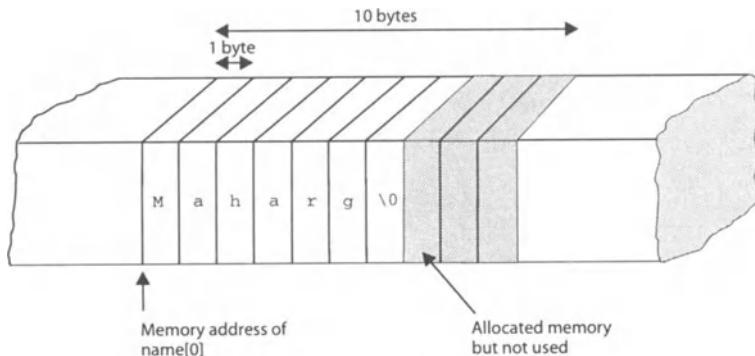
A C++ string must terminate with the null termination character '\0' or the value 0. Figure 7.5 illustrates the above-entered name in memory. Note the null termination character at the end of the variable string name. Why is the null terminator placed at the end of a string? Simply because the null terminator indicates the end of a string. Try running the above program V\_STRING.CPP with the following user input:

```
enter your forename: abcdefghij
```

This *strange* name will cause a system run-time error because the user entered a variable string of exactly ten characters in length, and one more character for the null terminator adds up to eleven. Remember that C++ does not perform bound checking on arrays.

### 7.10.2 Variable Strings with Embedded Spaces

The insertion operator (<< ) displays a string simply by displaying each element of the string until it encounters the null terminator. This is the reason for the null terminator: it indicates the end of a string. Without the null terminator a stream object would not know when the end of string had been encountered. You may be thinking that a space should indicate the end of a string; if so, then try running V\_STRING.CPP and enter two names that are separated by a space:



**Fig. 7.5** A string in memory is composed of an array of characters.

---

```
enter your forename: John Dan
```

```
hello John
```

The problem with the extraction operator (`>>`) is that it also considers white space to be a null terminator. The reason for this is that, as you may remember, the extraction operator can be chained:

```
//...
cout << "enter two numbers: " ;
cin >> number1 >> number2 ;
```

With typical user input:

```
enter two numbers: 12 24
```

which illustrates that the space between 12 and 24 is treated as a null terminator and is necessary for the correct extraction of the two numbers entered.

A way around this problem is the `get()` member function of **class istream** (of which `cin` is an object) which extracts a single character:

```
// cin_get.cpp
// illustrates the istream::get() function
// for extracting single characters from a string
#include <iostream.h> // C++ I/O

void main ()
{
    const int STRING_LEN = 80 ;
    // string variable: array of char
    char string[STRING_LEN] ;

    cout << "enter a string: " ;
    cin.get (string, STRING_LEN) ;

    cout << string ;
}
```

which could produce the output:

```
enter a string: always question the assumptions
always question the assumptions
```

This demonstrates that the function `istream::get()` extracts the entire string entered by a user until a new line is entered by pressing carriage return. Note that the function `get()` is called using the direct member access operator (`.`), since it is a member function of the `istream` **class**. This will become clearer when we examine classes.

As the last chapter discussed function overloading and default function arguments, it is informative to examine the signatures of the `istream::get()` function more closely:

<code>istream&amp; get (char</code>	<code>*buffer, int length, char delim</code>
	<code>= '\n' ) ;</code>

```

istream& get (sighned char *buffer, int length, char delim
              = '\n') ;
istream& get (unsigned char *buffer, int length, char delim
              = '\n') ;

```

`istream::get()` extracts single characters from the input stream and places them in the buffer, `buffer`, until the delimiter, `delim`, is encountered, the end of file is reached or until `(length-1)`<sup>1</sup> bytes have been read in. Note the use of the new line character '`\n`' as a default argument for the delimiter. Refer to your compiler's documentation or the `Iostream.h` header file for further information.

### 7.10.3 Constant Strings

In Chapter 4 we had a brief introduction to constant strings. As the name suggests, a constant string is an array of type `char` that is initialised to a string of characters. The following program illustrates constant strings:

```

// c_string.cpp
// illustrates constant strings
#include <iostream.h> // C++ I/O

void main ()
{
    char C_STRING1[30] = "C++ is a programming language" ;
    char C_STRING2[] = "C++ is a programming language" ;

    cout << C_STRING1 << endl ;
    cout << C_STRING2 ;
}

```

Note that the length of the first string is defined as 30, rather than 29, to account for a null terminator at the end of the string, while the second string is initialised with empty braces. As we saw with arrays of type `int` previously, the compiler can determine for itself the number of elements in a one-dimensional array.

Similarly, a two-dimensional array of constant strings can be initialised:

```

// init_str.cpp
// illustrates initialising a 2D array of strings
#include <iostream.h> // C++ I/O

void main ()
{
    const int MAX_STR_LEN = 9 ;
    const int NUM = 5 ;

    char names[NUM][MAX_STR_LEN] =
        { "Graham", "Ian", "Jim", "Jonathan", "Tony" } ;
}

```

---

<sup>1</sup> `(length-1)` to leave room for the null terminator.

```

for (int i=0; i< NUM; i++)
{
    cout << names[i] << endl ;
}
}

```

An array of strings is a two-dimensional array because a single string is a one-dimensional array of characters. Although `names` is a two-dimensional array, each string is accessed using only one subscript operator:

`names[i]`

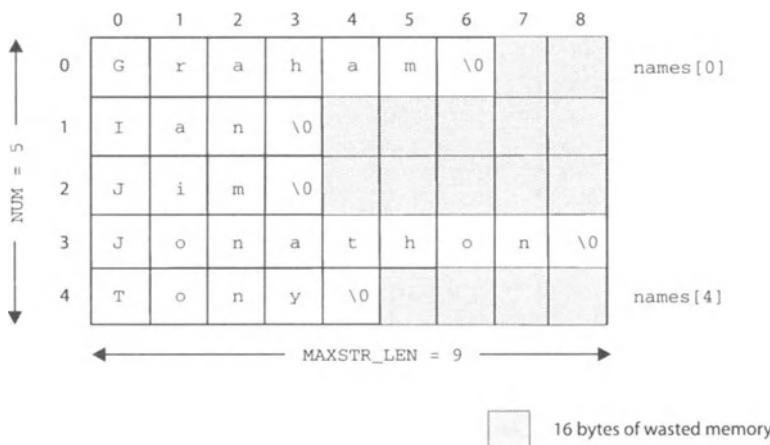
which accesses the start of each string. If you need to access an individual element of the array then indexing is as usual. For example, to access the letter ‘y’ in ‘Tony’ we would index column 3 in row 4:

`names[4][3]`

Figure 7.6 illustrates the five names in the array. Note the 16 bytes of unused memory.

## 7.11 Wasted Space

Returning to Fig. 7.6, we note that 16 bytes of the allocated 45 bytes of memory is not used because four of the five names are shorter in length than the maximum string length of nine. In order to remove this inefficiency in allocated memory we will need pointers. Although static arrays can be inefficient with respect to allocated memory, this inefficiency can be compensated for by the property that access time is the same for accessing any element of an array. Because the elements of an array are stored contiguously in memory, accessing the last element takes no longer than accessing the first element.



**Fig. 7.6** A two-dimensional array of strings.

## 7.12 Arrays as Function Arguments

In the previous chapter we examined functions in C++ and the passing of arguments to functions. Arrays can be passed as function arguments.

### 7.12.1 One-Dimensional Arrays

Let us consider a program which defines a function that is passed a one-dimensional array of numbers of type **float** and returns the average of the numbers. This program is similar to the INITIAL.CPP program presented above:

```
// func_arg.cpp
// illustrates passing of arrays to functions
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...

const int NUM_BOOKS = 10 ;

// returns the average of a 1D array of numbers
// passed as an argument
float Average (float array[NUM_BOOKS])
{
    float total_price = 0.0 ;

    for (int i=0; i<NUM_BOOKS; i++)
    {
        total_price += array[i] ;
    }
    return total_price / NUM_BOOKS ;
}

void main ()
{
    float computer_book_prices[NUM_BOOKS] =
    { 18.99, 27.45, 15.95, 26.95, 21.95,
      24.95, 29.95, 30.50, 16.95, 26.95 } ;

    // call function passing array of numbers
    float av_price = Average (computer_book_prices) ;

    cout << setiosflags (ios::fixed)
        << setprecision (2)
        << "average price of my 10 favourite computer books: "
        << av_price ;
}
```

The function declarator is given by:

```
float Average (float array[NUM_BOOKS])
```

indicating that a one-dimensional array of elements of type **float** is passed to the function *Average()* as an argument. We are able to pass the size of the array, since it is known at the time of compilation. However, it is possible to pass the array without including the array size:

```
float Average (float array[])
```

This is worth examining more closely, because it is a clue to what is actually passed to the function. When functions were discussed in Chapter 6 we noted that function arguments can be passed by *value*, *reference* or *pointer*; i.e. by *rvalue* or *lvalue*. In fact, an array is passed by lvalue, or by memory address. It is advantageous to pass arrays by address rather than by value because passing by value would involve copying each element of the passed array. Clearly, if the array was very large this would be highly inefficient from both memory and time considerations.

The following program examines the passing of an array from *main()* to a function *Func()*:

```
// address.cpp
// examines an array's address
#include <iostream.h> // C++ I/O

const int MAX = 5 ;

void Func (int a_func[MAX])
{
    cout << "rvalue a_func[0]: " << a_func[0]
        << ", lvalue a_func[0]: " << &a_func[0]
        << endl ;
}

void main ()
{
    int a_main[MAX] = { 0, 1, 2, 3, 4 } ;

    cout << "rvalue a_main[0]: " << a_main[0]
        << ", lvalue a_main[0]: " << &a_main[0]
        << endl ;

    // call function, passing array
    Func (a_main) ;

    cout << endl ;
    cout << "a_main : " << a_main << endl ;
    cout << "&a_main[0]: " << &a_main[0] << endl ;
    cout << "a_main[4] : " << a_main[4] << endl ;
    cout << "&a_main[4]: " << &a_main[4] << endl ;
}
```

which generates the following output:

```
rvalue a_main[0]: 0, lvalue a_main[0]: 0x31b7221a
rvalue a_func[0]: 0, lvalue a_func[0]: 0x31b7221a
```

```
a_main      : 0x31b7221a
&a_main[0]  : 0x31b7221a
a_main[4]   : 4
&a_main[4]  : 0x31b72222
```

The memory addresses will probably be different when you execute the above program. The first two lines of the output indicate that when an array is passed as an argument to a function the array elements are *not* copied, and in fact the address of the array is passed (since the lvalues of `a_main[0]` and `a_func[0]` are identical). Although, the above program only illustrates that arrays are passed by address by considering element 0, you may want to confirm for yourself that this is the general case by experimenting with the other array elements.

The last four lines of output illustrate some important properties of arrays. The first of the four lines indicates that when the array name, `a_main`, is used alone it in fact points to the address at which the array is stored. In other words:

*The name of an array is a synonym for the memory location of the array.*

The memory location of an array is at its zeroth element. Alternatively, and equivalently, the memory location at which an array resides can be accessed with the aid of the address-of operator (`&`). In terms of the example above this location is `&a_main[0]`, whereas the value of an array element at a particular index, `i`, from the zeroth element is given by `a_main[i]`. In the example above, we accessed the value of the fifth and last element of `a_main`, namely `a_main[4]`, and the memory address of this element: `&a_main[4]`.

### 7.12.2 Two-Dimensional Arrays

The following program illustrates passing two-dimensional arrays as function arguments:

```
// f_arg_2d.cpp
// illustrates passing two-dimensional
// arrays as function arguments

#include <iostream.h> // C++ I/O

const int SIZE = 2 ; // size of square arrays

// add two 2D arrays
void Add (int array1[][SIZE], int array2[][SIZE],
           int add[][SIZE])
{
    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            add[i][j] = array1[i][j] + array2[i][j] ;
        }
    }
}

// subtract two 2D arrays
```

```

void Sub (int array1[][][SIZE], int array2[][][SIZE],
           int sub[][][SIZE])
{
    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            sub[i][j] = array1[i][j] - array2[i][j] ;
        }
    }
}

// displays a 2D array
void Display (int array[][][SIZE], char string[])
{
    // display string
    cout << string << endl ;

    for (int i=0; i<SIZE; i++)
    {
        for (int j=0; j<SIZE; j++)
        {
            cout << array[i][j] << " " ;
        }
        cout << endl ;
    }
}

void main ()
{
    // define and initialise 2 arrays
    int a1[SIZE][SIZE] = { {1, 2}, {3, 4} } ;
    int a2[SIZE][SIZE] = { {5, 6}, {7, 8} } ;

    // define 2 placement arrays
    int alpa2[SIZE][SIZE] ;
    int alma2[SIZE][SIZE] ;

    // add & subtract
    Add (a1, a2, alpa2) ;
    Sub (a1, a2, alma2) ;

    // display arrays
    Display (a1, "array a1:") ;
    Display (a2, "array a2:") ;
    Display (alpa2, "array alpa2:") ;
    Display (alma2, "array alma2:") ;
}

```

with output:

```

array a1:
1 2
3 4
array a2:
5 6
7 8
array alpa2:
6 8
10 12
array alma2:
-4 -4
-4 -4

```

The declarator of function `Add()` is:

```
void Add (int array1[] [SIZE], int array2[] [SIZE],
           int add[] [SIZE])
```

which illustrates that three arrays are passed as arguments. The first two function arguments are the arrays to be added and the last argument is the resultant array. `Add()` is called from within `main()` with the call:

```
Add (a1, a2, alpa2) ;
```

When the `Add()` function returns, the resultant matrix ( $a1+a2$ ) is placed in matrix `alpa2`.

A frequent mistake when returning an array from a function is to return (using a **return** statement) an array which is defined local to the function:

```
int Add (int array1[] [SIZE], int array2[] [SIZE])
{
    int add [SIZE] [SIZE] ; // local to Add() function
    //...
    return add ; // returning address of array
}
```

Clearly, the return type specifier is in disagreement with the **return** statement. More importantly, the scope of array `add` is limited to the function definition, and when the function returns the memory allocated for array `add` is deleted, making it impossible for the function caller to access the resultant matrix.

Note that the `Display()` function is passed a string as a function argument so that the name of the array can be output by the function:

```
void Display (int array[] [SIZE], char string[]) ;
```

## 7.13 Calling Functions with Arrays as Arguments

In the program above, FUNC\_ARG.CPP, the function `Func()` was called passing an array, `a_main`, as an argument:

---

```
Func (a_main) ;
```

illustrating that when a function is called with an array as an argument, only the name of the array is passed. From the discussion in the previous section, the address of the array is passed, since the array name is in fact a synonym for the array's memory location.

## 7.14 Strings as Function Arguments

From above we saw that a string is an array of characters. Let's now examine the passing of strings to a function. The following program, STR\_ARG.CPP, illustrates passing strings to functions:

```
// str_arg.cpp
// illustrates passing strings as function arguments
#include <iostream.h> // C++ I/O

// compares two strings
// returns 1 if str1==str2, 0 if str1!=str2
int StrCompare (char str1[], char str2[])
{
    int i = 0 ;
    while (str1[i] == str2[i])
    {
        if (str1[i++] == '\0')
            return 1 ;
    }
    return 0 ;
}

// copies str2 into str1
void StrCopy (char str1[], char str2[])
{
    int i = 0 ;
    while ((str1[i] = str2[i]) != '\0')
        i++ ;
}

void main ()
{
    char string1[] = "this is a string" ;
    char string2[] = "this is a string" ;
    char string3[] = "another string" ;

    // compare string1 and string2
    int comp12 = StrCompare (string1, string2) ;

    if (comp12 == 1)
        cout << "string1 == string2" << endl ;
    else
```

```

cout << "string1 != string2" << endl ;

// compare string2 and string3
int comp23 = StrCompare (string2, string3) ;

if (comp23 == 1)
    cout << "string2 == string3" << endl ;
else
    cout << "string2 != string3" << endl ;

// copy string3 to string1
StrCopy (string1, string3) ;

cout << endl
    << "string1: " << string1 << endl ;
}

```

*StrCompare()* performs the comparison of the two strings passed as arguments by simply comparing each character of the two strings that have the same array index and returns a value of 1 if the strings are identical or 0 if the strings are different. *StrCopy()* copies one string, character by character, into another string.

The functions *StrCompare()* and *StrCopy()* are similar to the library functions *strcmp()* and *strcpy()*, respectively, declared in the header file STRING.H. *strcmp()* compares two strings and returns zero if the two strings are identical, or a negative or positive integer, depending on whether or not the first string is lexicographically less than or greater than the second string. *strcpy()* performs a similar operation to *StrCopy()*.

## 7.15 Passing the Size of an Array as an Argument

Consider the program:

```

// arg.cpp
// illustrates that a function does not know the size
// of an array passed as an argument
#include <iostream.h> // C++ I/O
#include <string.h> // strlen()

// uses strlen() to find string length
void StrLen (char str[])
{
    const int LENGTH = strlen (str) ;
    cout << "StrLen::LENGTH: " << LENGTH << endl ;
}

// uses sizeof to find string length
void StrSizeOf (char str[])
{
    const int LENGTH = sizeof str / sizeof str[0] ;
    cout << "StrSizeOf::LENGTH: " << LENGTH << endl ;
}

```

```

}

// returns 1 if str1==str2, 0 if str1!=str2
int StrCompare1 (char str1[], char str2[], int length)
{
    for (int i=0; i< length+1; i++)
    {
        if (str1[i] != str2[i])
            return 0 ;
    }
    return 1 ;
}

// passed a (2x3) array
void Display1 (int array[2][3])
{
    cout << "Display1: " << endl ;
    for (int i=0; i<2; i++)
    {
        for (int j=0; j<3; j++)
        {
            cout << array[i][j] << " " ;
        }
        cout << endl ;
    }
}

// passed a (2x3) array and no. of rows
void Display2 (int array[][3], int row)
{
    cout << "Display2: " << endl ;
    for (int i=0; i<row; i++)
    {
        for (int j=0; j<3; j++)
        {
            cout << array[i][j] << " " ;
        }
        cout << endl ;
    }
}

// passed a (2x3) array and no. of rows & cols
void Display3 (int array[][], int row, int col) // error!
{
    cout << "Display2: " << endl ;
    for (int i=0; i<row; i++)
    {
        for (int j=0; j<col; j++)
        {
            cout << array[i][j] << " " ;
        }
    }
}

```

```

        cout << endl ;
    }
}

void main ()
{
// define & initialise 2 strings
char string1[] = "string" ;
char string2[] = "string" ;

// length of strings
StrLen (string1) ;
StrSizeOf (string1) ;

// compare string1 and string2
int comp = StrCompare1 (string1, string2, 6) ;

if (comp == 1)
    cout << "string1 == string2" << endl ;
else
    cout << "string1 != string2" << endl ;

cout << endl ;

// define & initialise a (2x3) array of int's
int A[2][3] = { {0, 1, 2}, {3, 4, 5} } ;

Display1 (A) ;           // pass A
Display2 (A, 2) ;         // pass A & rows
Display3 (A, 2, 3) ;      // pass A, rows, cols
}

```

When an array is passed as an argument to a function the size of the array is not known to the function. To illustrate this point, the above program, ARG.CPP, defines two functions, *StrLen()* and *StrSizeOf()*, which determine the length of a string argument by means of the *strlen()* library function and the **sizeof** operator, respectively. *StrLen()* outputs the correct string length (not including the null terminator). *StrSizeOf()* outputs the incorrect string length by using the **sizeof** operator because the size of the string array *string*, defined in *main()*, is not available to *StrSizeOf()*. If we don't want to call the *strlen()* library function to determine the length of a string argument, one way around this problem is to pass the string length as an argument. This is the technique exercised in the function *StrCompare1()*, which is a modification of *StrCompare()* in the program STRCMP.CPP.

However, since strings are null-terminated, knowing the exact length of a string argument is not a problem. For instance, *strlen()* determines the correct length of a string argument by means of the null terminator, just as the *StrCompare()* function in STRCMP.CPP compared two string arguments without knowing the length of either string. For the interested reader, the general form of *strlen()* is (Kernighan and Ritchie, 1978):

```

int strlen (char str[])
{

```

---

```

char* p = str ;

while (*p != '\0')
    p++ ;
return p - str ;
}

```

which involves the use of a pointer to **char**, p.

However, two-dimensional arrays are slightly more difficult. The function *Display1()* in the ARG.CPP program listed above displays the integer elements of a (2×3) array passed as an argument, with the array sizes included in the function declarator. Since the first array size is not required by the compiler to determine the memory location of the first element of the array, *Display2()* passes the row size as an integer argument.

What we would really like to do is to define a function that is passed a two-dimensional array that is as general as possible, such as *Display3()*:

```
void Display3 (int array[][], int row, int col) // error!
```

Unfortunately, this is illegal because the second size (number of columns) must be known by the compiler in order to locate the first element of the array. A solution to this problem requires the use of pointers, which will be discussed in Chapter 12.

## 7.16 Small, Medium, Compact, Large and Huge Memory Models

In the A\_SIZE.CPP program a two-dimensional square array, *big\_array*, was defined as 2500 (50×50) elements of type **double** (8 bytes) or 20 kbyte. How big can an array be? Try increasing the array size to (90×90), or 64 800 bytes in total. This array size should be OK. Now try increasing the array size to (91×91), or 66 248 bytes. The program will no longer compile, because the array size is too large. To see why this array size is too large we first need a brief introduction to memory management.

IBM-compatible PCs are based on the Intel 8088 and 80x86 family of microprocessors. The internal registers of the microprocessors are 16 bits in length. The Intel 8088 microprocessor has four internal registers, referred to as *segment registers*, which are characterised as *code*, *data*, *stack* and *extra* segment registers. In order to construct a complete memory address, the 8088 microprocessor shifts the contents of a segment register to create a 20-bit address, which can access up to 1 Mbyte of memory. This 1 Mbyte of memory is divided into the following categories: The highest 64 kbyte is reserved for ROM BIOS; the next 192 kbyte is reserved for ROM expansion; and the next 128 kbyte is reserved for the video display, giving a total of 384 kbyte. Subtracting this memory from the total memory allocation of 1024 kbyte (=1 Mbyte), we are left with 640 kbyte of read-write memory. Of these 640 kbyte, the lowest area is reserved for read-write data required by ROM BIOS, and above this area MS-DOS loads, followed by device drivers. The remaining memory is used by programs executing under MS-DOS.

The Microsoft and Borland C++ compilers support five different memory models: small, medium, compact, large and huge. Different memory models refer to different data and code segments; see Table 7.1.

If a program requires more than 64 kbyte of code or data, it must contain two or more code or data segments respectively.

**Table 7.1** Memory models.

<i>Memory model</i>	<i>Data segments</i>	<i>Code segments</i>
Small	1	1
Medium	1	Multiple
Compact	Multiple	1
Large	Multiple	Multiple
Huge	Multiple	Multiple

### 7.16.1 Small and Medium Memory Models

The small memory model is the default memory model. The majority of small programs for Windows are compiled with the small memory model, with one code segment and one data segment. The compiler usually assumes that the stack segment (SS) is equal to the data segment (DS) for both small and medium memory models, although this can be overridden.

Functions for medium memory model programs generate far calls. Using the medium memory model for small programs is not necessary. Use the medium model when your program involves more than one source code module. The medium model allows each source code to be a different code segment, which can be both moveable and discardable. The space required to fit the program code into memory is the size of the largest code segment.

### 7.16.2 Compact, Large and Huge Memory Models

Compact and large memory models are available for Windows applications, but are not recommended. The additional data segments provided by these memory models must be fixed in memory and cannot be moveable. Also, a program that uses compact and large memory models is restricted to a single instance of a program.

Functions for compact and large memory model programs generate far calls and assume that they are passed long pointers to data. Limited support is offered for the huge memory model.

### 7.16.3 Libraries

Each of the five memory models has C and C++ libraries that are specific to a memory model (e.g. the CWS.LIB and CWM.LIB small and medium library files in the Borland C++ (version 5.0) library directory: \BC5\LIB). This is because a given function is called differently depending on the memory model. Medium and large memory model functions assume that they have been called from another segment, whereas the large and compact models assume that they have been passed long pointers to data. The huge model is similar to the large model, except that individual data items may be greater than 64 kbyte.

### 7.16.4 Windows Programming

Windows programs extensively use far calls to Windows functions, window procedures and call-back functions. The majority of data pointers exchanged between a Windows program and Windows are far pointers.

## 7.17 Summary

This chapter has introduced us to the array. An array is an aggregate of elements of a given fundamental or user-defined data type, and can have one, two, three or higher dimensions. The elements of an array are stored contiguously in memory, and an element of an array is accessed via the array name and an index, which is an integer. Indexing an array in C++ starts with element 0, not element 1. C++ performs no default bounds checking on an array. Arrays, like variables, can be initialised when they are defined. Arrays can be either static or dynamic. The size of a static array is constant and is defined at compilation. An array can be passed, by address, as an argument to a function. When a function is called that passes an array as an argument, only the array name is passed to the function, because the name of an array is a synonym for the array's memory location.

A string is a one-dimensional array of characters. Strings can be either constant or variable.

The elements of an array are stored consecutively in memory. This has the important consequence that when an array is defined at the time of compilation the size of the array remains constant throughout the lifetime of a program. Static arrays can be inefficient if all the allocated memory is not used, but this inefficiency can be compensated for by the fact that array indexing is random. To remove the inefficiency of wasted memory allocation we use an array which allocates memory *dynamically* as and when it requires it. A solution to the problem of dynamically expanding or reducing the size of an array in response to the insertion or deletion of elements is to use data structures such as linked lists, queues, stacks and trees. Linked lists provide a neat way of creating a dynamic list of elements or nodes which respond well to the insertion and deletion of elements, but suffer from the point of view of searching. Dynamic arrays and data structures will be discussed in later chapters.

An additional consequence of arrays storing their element data consecutively is that they do not lend themselves well to searching. Searching an array generally involves an exhaustive search of the entire array for a given element.

There is a direct correlation between arrays and pointers, which have yet to be discussed. Array indexing by means of the subscript operator [ ] can equally be performed with pointers, although with reduced clarity. A pointer *points* to the memory address of an identifier, just as a reference to an array element contains the memory address of the element.

## Exercises

- 7.1 Write a program which gets a set of floating-point numbers from a user and places the numbers in an array.
- 7.2 Extend the program in Exercise 7.1 so that the array of floating-point numbers can be passed to a function that returns the average of the array of numbers.
- 7.3 Develop a function which transposes the elements of a two-dimensional array. The transpose of an ( $m \times n$ ) array  $A = [a_{ij}]$  is the ( $n \times m$ ) array  $A^T = [a_{ji}]$ .
- 7.4 Develop a function called *StrLen()* (without using *strlen()*) which has the same functionality as the C++ library function *strlen()* in that it is passed a string and returns the length of the string, not counting the null-terminator character.
- 7.5 Frequently, when working with data acquisition systems, data arrives from a one-dimensional array buffer. However, when manipulating the data it is often more convenient to manipulate a two-dimensional array. For example, the rows of a two-dimensional array may correspond to different channel numbers. Develop two functions which convert a

(12×1) one-dimensional array to a (3×4) two-dimensional array, and vice versa. The signatures of the two functions should be of the form:

```
void OneDToTwoD (int array1D[3*4], int array2D[3][4]) ;
void TwoDToOneD (int array2D[3][4], int array1D[3*4]) ;
```

- 7.6 Develop three functions which add, subtract and multiply two two-dimensional arrays of floating-point numbers. The functions must cater for matrices of (3×3) and (3×1) sizes and conform with the rules of matrix operations. Also, develop functions which display the elements of (3×3) and (3×1) two-dimensional arrays. Test your program on a variety of arrays that are initialised within your program. Place the function signatures and definitions in files called ARRAY.H and ARRAY.CPP, respectively, so that the next exercise can include ARRAY.H.
- 7.7 This exercise is concerned with the rotation of a point,  $p(x,y,z)$ , in three-dimensional space. For three-dimensional space it is convenient to use matrix algebra in which a rotation is defined by the three angular components  $\theta, \beta$  and  $\gamma$  about the  $x$ -,  $y$ - and  $z$ -axes with respect to the  $x'$ -,  $y'$ - and  $z'$ -axes, respectively. A point  $p(x,y,z)$  is transformed to a point  $p'(x',y',z')$  by an application of a rotation matrix,  $R$ :

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

If we rotate about the  $z$ -axis ( $\theta \neq 0$ ), the rotation matrix is given by:

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If we rotate about the  $y$ -axis ( $\beta \neq 0$ ), the rotation matrix is:

$$R_\beta = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$

Finally, if we rotate about the  $x$ -axis ( $\gamma \neq 0$ ), the rotation matrix is:

$$R_\gamma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix}$$

The total rotation matrix for an arbitrary rotation is  $R=R_\theta R_\beta R_\gamma$  and can be shown to be given by:

$$R = \begin{bmatrix} \cos\theta \cos\beta & \sin\theta \cos\gamma + \cos\theta \sin\beta \sin\gamma & \sin\theta \sin\gamma - \cos\theta \sin\beta \cos\gamma \\ -\sin\theta \cos\beta & \cos\theta \cos\gamma - \sin\theta \sin\beta \sin\gamma & \cos\theta \sin\gamma + \sin\theta \sin\beta \cos\gamma \\ \sin\beta & -\cos\beta \sin\gamma & \cos\beta \cos\gamma \end{bmatrix}$$

Develop four functions for  $R_\theta$ ,  $R_\beta$ ,  $R_\gamma$  and  $R$  of the form:

```
void RTheta (const double& theta, double array[3][3]) ;
```

Test your rotation matrices on various rotated points using your matrix manipulation functions developed in Exercise 7.6.

# Structures, Unions, Enumerations and Typedefs

*This chapter examines structures, unions, enumerations and typedefs which allow a programmer to create new data types. The ability to create new types is an important and very powerful feature of C++ and releases a programmer from being restricted to the integral types offered by the language. Structures enable a programmer to form a collection of similar or different data types into a single user-defined data type, unlike arrays, which are collections of a single data type. The data members of a structure are, by default, publicly accessible. User-defined structures integrate exactly into the language. In other words, there is a single syntax for the definition and manipulation of objects of both integral C++ and user-defined types.*

*A union is similar to a structure except that a union can hold only one of its data members at any given time. Union data members share the same location in memory. An enumerated data type is a collection of named members which have equivalent integer values. A typedef is in fact not a new data type but merely a synonym for an existing type.*



## 8.1 Structures

A structure is a user-defined data type that is a collection of similar or dissimilar data items or members.

The previous chapter illustrated the array, which enables us to form an aggregate of *identical* data items. If we are required to form a collection of different data types, we cannot use an array. The C++ keyword **struct** enables a programmer to create *new* structure data types which encompass a collection of similar or different data types. The ability to develop new user-defined data types means that a programmer is not restricted to the integral data types offered by the C++ language. The following program illustrates a Person structure:

```
// struct.cpp
// illustrates structures
```

```
#include <iostream.h> // C++ I/O

// declare a structure 'Person'
struct Person
{
    char name[20] ; // 4name
    int age ; // yr
    float height ; // m
    float mass ; // kg
};

void main ()
{
    // define a structure variable 'person'
    Person person ;

    // prompt user for data and assign
    // values to struct Person members
    cout << "enter your forename: " ; cin >> person.name ;
    cout << "enter your age: " ; cin >> person.age ;
    cout << "enter your height: " ; cin >> person.height ;
    cout << "enter your mass: " ; cin >> person.mass ;

    // O/P
    cout << endl << "hello "
        << person.name << ", you are: "
        << person.age << " years old, "
        << person.height << "m tall and "
        << person.mass << "kg in mass" ;

    // define and initialise a Person variable 'harry'
    Person harry = { "Harry", 25, 1.85, 80 } ;

    // O/P Harry:
    cout << endl << "hello "
        << harry.name << ", you are: "
        << harry.age << " years old, "
        << harry.height << "m tall and "
        << harry.mass << "kg in mass" ;
}
```

Some user interaction with the above program is:

```
enter your forename: Tom
enter your age: 45
enter your height: 1.8
enter your mass: 74
```

```
hello Tom, you are: 45 years old, 1.8m tall and 74kg in mass
hello Harry, you are 25 years old, 1.85m tall and 80kg in mass
```

### 8.1.1 Declaring a Structure

The above program *declares* a structure called Person, shown again below:

```
struct Person
{
    char name[20] ;
    int age ;
    float height ;
    float mass ;
};
```

Note that this is a declaration rather than a definition. It was noted in Chapter 4 that a declaration does not allocate storage, whereas a definition is a declaration that simultaneously allocates memory for a data item. The structure declaration is a *mould* or a template that outlines the contents of a structure.

The general syntax of a structure declaration is of the form:

```
struct Name
{
    typeSpecifier member_name ;    // structure members
    //...
    typeSpecifier member_name ;
};
```

The structure declaration begins with the keyword **struct**. This is followed by the structure tag or name. Note that the style I adopt is to use an uppercase letter for the first letter of single or multiple-connected structure names (e.g. Person, Triangle, TwoDTriangle, ThreeDTriangle). Between the opening, {, and closing, }, braces of the structure declaration are the members of the structure. Finally (and it's easily forgotten!), there comes the semicolon, following directly after the closing brace. The semicolon after the closing brace helps to distinguish a structure declaration from a function definition.

In the above program, STRUCT.CPP, the structure Person was declared outside the *main()* function and thus has global scope. Thus, any function (including *main()*) can define a variable of type Person. However, if a structure is declared within the body of a function, then the structure declaration is local to that function. For example:

```
// scope.cpp
// illustrates a structure's scope
#include <iostream.h> // C++ I/O

// prototype
void Function () ;

void main ()
{
    // declare a structure 'MainPerson' within main()
    struct MainPerson
    {
        char name[20] ; // 4name
        int age ;       // yr
```

```
    float height ;      // m
    float mass ;        // kg
}

MainPerson m_person ;
}

void Function ()
{
    MainPerson f_person ;  // error: undefined type MainPerson!
}
```

This program declares a structure `MainPerson` within the function body of `main()` and is therefore not accessible outside `main()`.

### 8.1.2 Structure Tag or Name

The structure tag or name is not a variable or an object but rather a name to a collection of data types.

### 8.1.3 Defining a Structure Variable or Object

The program `STRUCT.CPP` defines a variable or an object of type `Person`:

```
//...
Person person ;
```

This statement allocates enough memory to create a single structure variable or object of type `Person`. This definition is similar to defining a variable of a fundamental data type:

```
//...
int var ;
double x ;
```

Just as for an array, a structure variable can be initialised when it is defined. `STRUCT.CPP` illustrates that a variable `harry` is initialised by an *initialiser list*, whose elements correspond exactly to the members of the structure. Each element of the initialiser list is separated by the comma operator, and all elements are placed between the braces { and }:

```
Person harry = { "Harry", 25, 1.85, 80 } ;
```

### 8.1.4 Accessing a Structure's Data Members

Program `STRUCT.CPP` outputs to the screen the data members of a user-entered data structure by accessing each of the structure's data members independently through the use of the dot operator ( . ):

```
Person person ;
//...
// O/P
```

```

cout << endl << "hello "
    << person.name    << ", you are: "
    << person.age     << " years old, "
    << person.height  << "m tall and "
    << person.mass    << "kg in mass" ;

```

The four members of structure Person (name, age, height and mass) are accessed using the dot operator, which is placed between the structure variable name and the structure member. The general syntax for accessing a structure's members is:

```
structure_name.structure_member
```

Such access is often referred to as *direct member access*, as opposed to *indirect member access* via the use of the `->` operator for use with pointers (Chapter 12).

### 8.1.5 The Size of a Structure

The following program illustrates the use of the `sizeof` operator to determine the size of a structure:

```

// sizeof.cpp
// illustrates determining the size of a structure
#include <iostream.h> // C++ I/O

// declare a structure 'Person'
struct Person
{
    char name[20] ; // 4name
    int age ;        // yr
    float height ;   // m
    float mass ;      // kg
};

void main ()
{
    Person person ;

    cout << "sizeof (Person): " << sizeof (Person) ;
}

```

which produces the output:

```
sizeof (Person): 30
```

Note the use of the parentheses ( and ) when the `sizeof` operator is applied to Person. This is because Person is a type.

To see how the size of type Person is determined, refer to Table 8.1. The size of Person is the sum of its individual members.

**Table 8.1** Size of type Person.

<b>Person member</b>	<b>Member datatype</b>	<b>Size (bytes)</b>
name[20]	<b>char</b> (1)	20
age	<b>int</b> (2)	2
height	<b>float</b> (4)	4
mass	<b>float</b> (4)	4
<i>Total</i>		30

### 8.1.6 Declaring and Defining a Structure in a Single Statement

It is possible to declare and define a structure simultaneously in a single statement. The following program code illustrates this:

```
// declare and define a structure
struct Person
{
    char name[20] ; // 4name
    int age ; // yr
    float height ; // m
    float mass ; // kg
} person ;
```

which defines a structure variable name, person, between the closing brace, }, of the Person structure declaration and the semicolon. In addition, we can define several variables at the same time:

```
struct Person
{
    char name[20] ; // 4name
    int age ; // yr
    float height ; // m
    float mass ; // kg
} tom, dick, harry ;
```

The main problem with declaring and defining a structure in a single statement is that the structure variable(s) have the same scope as the structure declaration. It was noted above that generally structure declarations are given global scope to enable all functions to access to the structure. Thus declaring and defining a structure simultaneously creates global variables, the number of which should be minimised if possible.

### 8.1.7 Defining a Structure Variable or Object Without a Structure Name

Structures can also be defined without a structure tag or name:

```
struct // nameless structure!
{
    char name[20] ; // 4name
    int age ; // yr
```

```
float height ;      // m
float mass ;        // kg
} person ;
```

Such an *anonymous* structure definition is of limited use because we have no structure name to enable us to create a variable other than `person`.

### 8.1.8 Output of a Structure's Data Members

You may have noticed from the above program that in order to output the data members of a given structure variable we must explicitly output (by means of the `cout` stream object) each of the structure's data members:

```
//...
// O/P
cout << endl << "hello "
    << person.name    << ", you are: "
    << person.age     << " years old, "
    << person.height << "m tall and "
    << person.mass    << "kg in mass" ;
```

This can be a bit laborious and messy. Instead, why can't we simply write:

```
//...
cout << person ; // error: illegal structure operation
```

This output statement would be nice, but unfortunately the `cout` stream object was not designed for our `Person` structure and hence doesn't know how to output it. Try modifying `STRUCT.CPP` to include the above single output statement. The compiler will issue an error message of the form 'illegal use of structure variable'.

We shall see later when we discuss classes and operator overloading in Chapters 9 and 10 that simple statements such as:

```
cout << person ;
```

are possible.

### 8.1.9 Assignment

One structure variable can be assigned to another. The following program illustrates this for two `Person` variables `susan` and `sarah`:

```
// assign.cpp
// illustrates assignment of structure variables
#include <iostream.h> // C++ I/O

// declare a structure 'Person'
struct Person
{
    char name[20] ; // 4name
```

```

int age ;           // yr
float height ;      // m
float mass ;        // kg
};

void main ()
{
// define two structure variables
Person susan = { "Susan", 31, 1.6, 58 } ;
Person sarah = { "Sarah", 41, 1.7, 60 } ;

// O/P:
cout << "before assignment:" << endl
    << susan.name << " is "
    << susan.age << " years old, "
    << susan.height << "m tall and "
    << susan.mass << "kg in mass"
    << endl ;
cout << sarah.name << " is "
    << sarah.age << " years old, "
    << sarah.height << "m tall and "
    << sarah.mass << "kg in mass"
    << endl ;

// assign 'sarah' to 'susan'
susan = sarah ;

cout << "after assignment:" << endl
    << susan.name << " is "
    << susan.age << " years old, "
    << susan.height << "m tall and "
    << susan.mass << "kg in mass"
    << endl ;
}

```

The assignment statement is given by:

```
susan = sarah ;
```

and copies *all* of sarah's data members to susan's data members. This is the default operation of the assignment operator: it will copy all data members from an object on the right-hand side of the assignment operator, member-by-member, to an object on the left-hand side of the assignment operator.

Try adding the following statement to the above program, ASSIGN.CPP:

```
susan + sarah ; // error: illegal use of structure
```

This will result in a compilation error of the form 'illegal operation on a structure object'. While the assignment operator has a default behaviour, other operators, such as the arithmetic operators, do not have default operations. To write such statements and expressions in which variables and objects of user-defined structures and classes are operands will require the use

of overloaded operators. Overloading operators specifically for a user-defined type is covered in Chapter 10.

### 8.1.10 Arrays of Structures

The following program defines and initialises an array of five elements of type Person:

```
// array.cpp
// illustrates an array of structures
#include <iostream.h> // C++ I/O

const int NUM_OF_PEOPLE = 5 ;

// declare a structure 'Person'
struct Person
{
    char name[20] ; // 4name
    int age ; // yr
    float height ; // m
    float mass ; // kg
};

void main ()
{
    // define and initialise an array of Person's
    Person persons[NUM_OF_PEOPLE] =
        { { "Tom", 23, 1.71, 65 },
          { "Dick", 24, 1.74, 75 },
          { "Harry", 25, 1.85, 80 },
          { "Susan", 31, 1.6, 58 },
          { "Sarah", 41, 1.7, 60 } } ;

    // O/P
    for (int i=0; i<NUM_OF_PEOPLE; i++)
    {
        cout << persons[i].name << " is "
            << persons[i].age << " years old, "
            << persons[i].height << "m tall and "
            << persons[i].mass << "kg in mass" << endl ;
    }
}
```

The array of type Person is defined as:

```
Person persons[NUM_OF_PEOPLE] ;
```

which has exactly the same syntax as an array of elements of a fundamental data type. Accessing the data members of the Person structure now uses the dot member access and array subscript operators:

```
persons[i].name ;
```

which accesses the name data member of Person for each array element.

### 8.1.11 Structures as Function Arguments

Provided a suitable structure declaration exists, function arguments of a user-defined structure type are perfectly feasible:

```
// func_arg.cpp
// illustrates a structure function argument
#include <iostream.h> // C++ I/O

const int NUM_OF_PEOPLE = 5 ;

// declare a structure 'Person'
struct Person
{
    char name[20] ; // 4name
    int age ; // yr
    float height ; // m
    float mass ; // kg
};

// displays a Person
void Display (Person p)
{
    cout << p.name << " is "
        << p.age << " years old, "
        << p.height << "m tall and "
        << p.mass << "kg in mass" << endl ;
}

void main ()
{
    // define and initialise an array of Person's
    Person persons[NUM_OF_PEOPLE] =
    { { "Tom", 23, 1.71, 65 },
      { "Dick", 24, 1.74, 75 },
      { "Harry", 25, 1.85, 80 },
      { "Susan", 31, 1.6, 58 },
      { "Sarah", 41, 1.7, 60 } } ;

    // O/P
    for (int i=0; i<NUM_OF_PEOPLE; i++)
    {
        Display (persons[i]) ;
    }
}
```

The above program modifies ARRAY.CPP by simply placing the cout stream output into a function called *Display()*. The signature of *Display()* is:

---

```
void Display (Person p) ;
```

which passes, by value, a variable of type Person as a function argument. We noted in Chapter 6 that function arguments can also be passed by reference. Passing by reference a Person function argument to function *Display()* could be written as either of the following:

```
void Display (Person& p) ;
```

or

```
void Display (const Person& p) ;
```

The latter ensures that the *Display()* function does not alter the function argument.

All of the properties of functions examined in Chapter 6 apply equally to user-defined structure types.

### 8.1.12 Nested Structures

Once a structure has been declared, it is perfectly feasible to use this structure just like a fundamental data type, and hence nest a structure within another structure. For instance, if we are required to extend our Person structure to include a date of birth data member, we could declare another structure called Date:

```
struct Date
{
    unsigned int day ;
    unsigned int month ;
    unsigned int year ;
};
```

Thus our original Person structure could be modified in order to include a Person's date of birth:

```
struct Person
{
    char name[20] ; // 4name
    unsigned int age ; // yr
    float height ; // m
    float mass ; // kg
    Date dob ; // date of birth
};
```

In addition, an EmployedPerson structure could be declared which contains the necessary data members for the categorisation of an employed person:

```
struct EmployedPerson
{
    Person person ; // Person details
    unsigned int salary ; // wages
    unsigned int hours_per_week ; // working week
};
```

Note the use of **unsigned int** for data members which are known to be positive integers (assuming an employee does not pay an employer for working!). The following program demonstrates the use of nested structures using the Date, Person and EmployedPerson structures declared above.

```
// nest.cpp
// illustrates nested structures
#include <iostream.h> // C++ I/O

struct Date
{
    unsigned int day ;
    unsigned int month ;
    unsigned int year ;
};

struct Person
{
    char name[20] ; // 4name
    unsigned int age ; // yr
    float height ; // m
    float mass ; // kg
    Date dob ; // date of birth
};

struct EmployedPerson
{
    Person person ; // Person details
    unsigned int salary ; // wages
    unsigned int hours_per_week ; // working week
};

void main ()
{
    // define an EmployedPerson variable
    EmployedPerson ep ;

    // set data members
    cout << "enter your name: " ; cin >> ep.person.name ;
    cout << "enter your age: " ; cin >> ep.person.age ;
    cout << "enter your height: " ; cin >> ep.person.height ;
    cout << "enter your mass: " ; cin >> ep.person.mass ;
    cout << "enter your date of birth: " ;
    cin >> ep.person.dob.day
        >> ep.person.dob.month
        >> ep.person.dob.year ;
    cout << "enter your salary: " ;
    cin >> ep.salary ;
    cout << "enter no. of hr's/wk. at work: " ;
    cin >> ep.hours_per_week ;
```

```
// O/P
cout << endl
<< "name: " << ep.person.name << endl
<< "age: " << ep.person.age << endl
<< "height: " << ep.person.height << endl
<< "mass: " << ep.person.mass << endl
<< "date of birth: "
<< ep.person.dob.day << "-"
<< ep.person.dob.month << "-"
<< ep.person.dob.year << endl
<< "salary: " << ep.salary << endl
<< "working week (hrs): " << ep.hours_per_week ;
}
```

To access the data members of nested structures requires the use of multiple dot operators. For instance, to access the day of the month of an employed person's date of birth, we use the dot operator three times as follows:

```
ep.person.dob.day
```

The EmployedPerson structure contains a Person data member which contains a Date data member which contains a day data member. Structures can be nested to any degree.

Finally, it is possible to initialise nested structures upon definition. The following example initialises a variable john of type Person:

```
//...
Person john = { "John", 30, 1.675, 81, {12, 1, 65} } ;
```

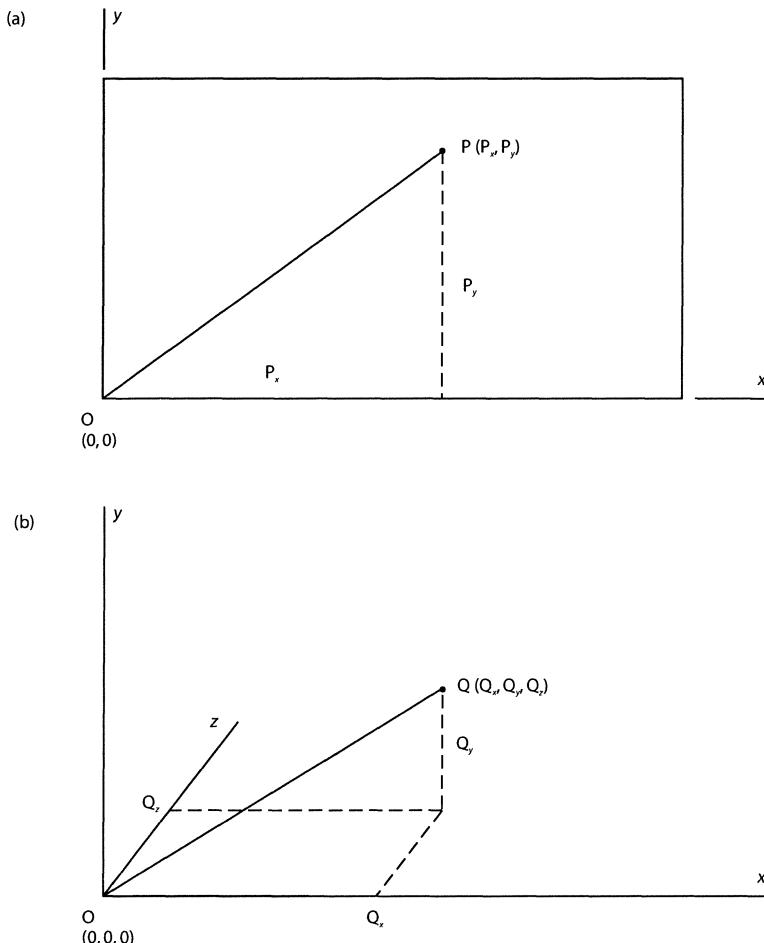
Note the parentheses within the initialiser list, encompassing the fifth data member, dob, of type structure Date.

### 8.1.13 A Point Structure

In geometric modelling we frequently require the coordinates of a point relative to a given reference point or origin. Figure 8.1(a) illustrates an arbitrary point P in a two-dimensional domain, such as a display screen, while Fig. 8.1(b) illustrates a point Q in a general three-dimensional space. The position of P is categorised by the Cartesian coordinate system ( $x, y$ ) with origin at the point O (0, 0). If the  $x$ - and  $y$ -coordinates of P are  $P_x$  and  $P_y$ , respectively, then point P is given by P ( $P_x, P_y$ ). Similarly, the point Q is given by the triple points ( $Q_x, Q_y, Q_z$ ); where z is the out-of-plane dimension.

A general three-dimensional point could simply be modelled as three separate variables:

```
//...
// define & initialise 3 coordinate positions of a point p
double p_x, p_y, p_z = 0.0 ;
//...
// move point, p, to (1,2,0)
p_x = 1.0 ; p_y = 2.0 ; p_z = 0.0 ;
//...
```



**Fig. 8.1** (a) Two-dimensional and (b) three-dimensional points *P* and *Q*.

Alternatively, point *p* could be represented as an array, since each of its three coordinates are of the same type, **double**:

```
//...
const int 3D = 3 ;
//...
// define & initialise 2 point arrays
double p[3D] = { 0.0, 0.0, 0.0 } ;
double q[3D] = { 0.0, 0.0, 0.0 } ;
//...
// move point, p, to (1, 2, 0)
p[0] = 1.0 ; p[1] = 2.0 ; p[2] = 0.0 ;
//...
```

Both of the above two representations of a given point work, but are very clumsy and do nothing to increase the generality of a program and hide the details of a point from a user. A point in a

given space is composed of three individual coordinates  $x, y$  and  $z$ , but it is instructive to view a point as a single entity. Thus a point naturally lends itself to a structure with the three coordinates being the structure's members:

```
struct Point
{
    double x, y, z ;
};
```

This structure is used in the following program, which defines a function *Distance()* that returns the distance between two Points:

```
// point.cpp
// illustrates a Point structure
#include <iostream.h> // C++ I/O
#include <math.h> // sqrt()

// a Point structure for representing a 3D point
struct Point
{
    double x, y, z ;
};

// returns the distance between two Points
double Distance (const Point& p1, const Point& p2)
{
    double x = p2.x - p1.x ;
    double y = p2.y - p1.y ;
    double z = p2.z - p1.z ;

    double r_sq = x * x + y * y + z * z ;

    return sqrt (r_sq) ;
}

void main ()
{
    // define & initialise two Points in the plane z=0
    Point p = {1.0, 1.0, 0.0} ;
    Point q = {3.0, 2.0, 0.0} ;

    cout << "distance between p and q: " << Distance (p, q) ;
}
```

The distance,  $r$ , between two points  $p$  ( $p_x, p_y, p_z$ ) and  $q$  ( $q_x, q_y, q_z$ ) is given by:

$$r = [(q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2]^{1/2}$$

At present our Point structure is beautifully simple, but also fairly limited in its applications. However, the Point type is something that we shall return to again and again throughout

subsequent chapters (when we visit classes) because it is so fundamental to geometry and computer graphics.

### 8.1.14 Data and Functions

In C++, structures are typically used to hold different data types, while classes are used to hold both data and functions. In fact, C++ also allows structures to have data and function members. For instance, we could extend the `Point` structure to provide two member functions `Add()` and `Subtract()`, which add and subtract, respectively, two objects of type `Point`:

```
// dat&fun.cpp
// illustrates a structure declaration
// with data and function members
#include <iostream.h> // C++ I/O

struct Point
{
    // data members:
    double x, y, z;

    // function members:
    void Add (const Point& p1, const Point& p2)
    {
        x = p1.x + p2.x ;
        y = p1.y + p2.y ;
        z = p1.z + p2.z ;
    }

    void Subtract (const Point& p1, const Point& p2)
    {
        x = p1.x - p2.x ;
        y = p1.y - p2.y ;
        z = p1.z - p2.z ;
    }
}; // Point

void main ()
{
    Point p1, p2, p3 ;

    p1.x = 1.0 ; p1.y = 2.0 ; p1.z = 3.0 ;
    p2.x = 4.0 ; p2.y = 5.0 ; p2.z = 6.0 ;
    p3.x = 0.0 ; p3.y = 0.0 ; p3.z = 0.0 ;

    // O/P before Add():
    cout << "before: " << endl ;
    cout << "p3.x: " << p3.x << endl ;
    cout << "p3.y: " << p3.y << endl ;
    cout << "p3.z: " << p3.z << endl ;
```

```

p3.Add (p1, p2) ;

// O/P after Add():
cout << "after: " << endl ;
cout << "p3.x: " << p3.x << endl ;
cout << "p3.y: " << p3.y << endl ;
cout << "p3.z: " << p3.z << endl ;
}

```

The declaration of type `Point` now includes two member functions `Add()` and `Subtract()`. These member functions are passed two arguments of type `Point` and manipulate these two `Points` by adding or subtracting their respective (`x`, `y` and `z`) data members. The result of the addition or subtraction is assigned to the data members of the object that calls the member functions. If this seems unclear at the moment then don't worry – the next chapter will discuss in detail **class** member functions.

In C++, structures are generally reserved as a data encapsulation mechanism that holds only data members, whereas the **class** is used to define types that hold both data and functions. Thus a more detailed discussion of member functions is left to the next chapter.

### 8.1.15 Private or Public?

We saw above that, if a structure is declared with global scope, its data members are accessible from any function. In other words, its members are *public*. This may be what we require, but it can be the source of problems. The members of a structure are by default public, and are therefore accessible to an object.

In the previous section two member functions were added to the structure `Point` for adding and subtracting its data members. However, the structure declaration did not correspondingly restrict the access of the structure's data members to the member functions only. Thus, with both data and function members declared public by default, we are still able to manipulate `Point`'s data members from outside the structure declaration:

```

//...
// define and initialise two Points:
p1.x = 1.0 ; p1.y = 2.0 ; p1.z = 3.0 ;
p2.x = 4.0 ; p2.y = 5.0 ; p2.z = 6.0 ;
p3.x = 0.0 ; p3.y = 0.0 ; p3.z = 0.0 ;
//...
Point p3 ;
// p3 = p1 + p2:
p3.x = p1.x + p2.x;
p3.y = p1.y + p2.y ;
p3.z = p1.z + p2.z ;
// or:
p3.Add (p1, p2) ;
//...

```

A further discussion of private and public accessibility will be delayed until the next chapter, when the C++ **class** is introduced.

### 8.1.16 Point, Line and Triangle Structures

Let us make use of the `Point` structure by modelling a line segment and a triangle in a three-dimensional space. A straight line segment, `Line`, can be represented as start and terminal `Points`. A triangle, `Triangle`, can be composed of three vertices and three edges. The following program, `PLT.CPP`, declares `Point`, `Line` and `Triangle` structures and defines three functions `Distance()`, `TrianglePerimeter()` and `TriangleCentroid()` which calculate the minimum distance between two `Points`, the perimeter of a `Triangle` and the centroid of a `Triangle`, respectively. The perimeter,  $p$ , of a triangle is simply the sum of the triangle's edge lengths. The centroid,  $c$ , of a triangle  $(i, j, k)$  is:

$$c(c_x, c_y, c_z) = \left( \frac{x_i + x_j + x_k}{3}, \frac{y_i + y_j + y_k}{3}, \frac{z_i + z_j + z_k}{3} \right)$$

Figure 8.2 illustrates a triangle in the plane  $z=0$  for which the perimeter and centroid are:

$$p = \frac{2\sqrt{5} + \sqrt{17} + \sqrt{45}}{2}, \quad c = \left( \frac{13}{6}, \frac{7}{3}, 0 \right)$$

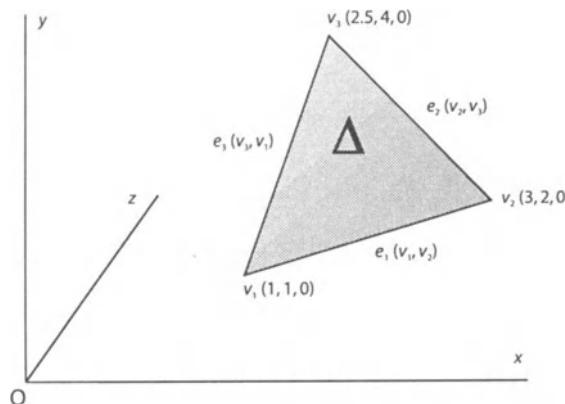
Program `PLT.CPP` tests the `Point`, `Line` and `Triangle` structures for the triangle shown in Fig. 8.2:

```
// plt.cpp
// Point, Line and Triangle structures

#include <iostream.h> // C++ I/O
#include <math.h> // sqrt()

struct Point
{
    double x, y, z; // triple coordinates of Point
};

struct Line
```



**Fig. 8.2** A triangle in the plane  $z=0$ .

```
{  
    Point p1, p2 ;           // two end Points of line  
};  
  
struct Triangle  
{  
    Point v1, v2, v3 ;     // three vertices of triangle  
    Line e1, e2, e3 ;     // three edges of triangle  
};  
  
// returns the distance between two Points  
double Distance (const Point& p1, const Point& p2)  
{  
    double x = p2.x - p1.x ;  
    double y = p2.y - p1.y ;  
    double z = p2.z - p1.z ;  
  
    double r_sq = x * x + y * y + z * z ;  
  
    return sqrt (r_sq) ;  
}  
  
// returns the perimeter of a triangle  
double TrianglePerimeter (const Triangle& t)  
{  
    double e1_length = Distance (t.e1.p1, t.e1.p2) ;  
    double e2_length = Distance (t.e2.p1, t.e2.p2) ;  
    double e3_length = Distance (t.e3.p1, t.e3.p2) ;  
  
    return e1_length + e2_length + e3_length ;  
}  
  
// returns the centroid of a triangle  
Point TriangleCentroid (const Triangle& t)  
{  
    Point cent ;  
  
    cent.x = (t.v1.x + t.v2.x + t.v3.x) / 3.0 ;  
    cent.y = (t.v1.y + t.v2.y + t.v3.y) / 3.0 ;  
    cent.z = (t.v1.z + t.v2.z + t.v3.z) / 3.0 ;  
  
    return cent ;  
}  
  
void main ()  
{  
    // define a Triangle object tri  
    Triangle tri ;  
  
    // set tri's data members:  
    // vertices
```

```

tri.v1.x = 1.0 ; tri.v1.y = 1.0 ; tri.v1.z = 0.0 ;
tri.v2.x = 3.0 ; tri.v2.y = 2.0 ; tri.v2.z = 0.0 ;
tri.v3.x = 2.5 ; tri.v3.y = 4.0 ; tri.v3.z = 0.0 ;
// edges
tri.e1.p1.x = 1.0 ; tri.e1.p1.y = 1.0 ; tri.e1.p1.z = 0.0 ;
tri.e1.p2.x = 3.0 ; tri.e1.p2.y = 2.0 ; tri.e1.p2.z = 0.0 ;
tri.e2.p1.x = 3.0 ; tri.e2.p1.y = 2.0 ; tri.e2.p1.z = 0.0 ;
tri.e2.p2.x = 2.5 ; tri.e2.p2.y = 4.0 ; tri.e2.p2.z = 0.0 ;
tri.e3.p1.x = 2.5 ; tri.e3.p1.y = 4.0 ; tri.e3.p1.z = 0.0 ;
tri.e3.p2.x = 1.0 ; tri.e3.p2.y = 1.0 ; tri.e3.p2.z = 0.0 ;

double perimeter = TrianglePerimeter (tri) ;
Point cent = TriangleCentroid (tri) ;

// O/P
cout << "perimeter of triangle: " << perimeter << endl ;
cout << "centroid of triangle: (" << cent.x << ", "
<< cent.y << ", "
<< cent.z << ")"
<< endl ;
}

```

with output:

```

perimeter of triangle: 7.68172
centroid of triangle: (2.16667, 2.33333, 0)

```

The declarations of the Line and Triangle structures are:

```

struct Line
{
    Point p1, p2 ;           // two end Points of line
};

struct Triangle
{
    Point v1, v2, v3 ;      // three vertices of triangle
    Line e1, e2, e3 ;      // three edges of triangle
};

```

Since a Line incorporates two Points the Triangle structure could have been declared with only three Line data members rather than three Point and Line data members. However, the encapsulation of three Point and Line data members greatly assists the use of the Triangle structure.

Since all of the data members of structures Point, Line and Triangle are **public**, by default, the Point and Line data members of a Triangle object, tri, are typically accessed by:

```

void main ()
{
//...

```

---

```

Triangle tri ;           // Triangle object tri

tri.v1.x = 1.0 ;         // x coor. of Point vertex v1 of tri
//...
tri.e1.p1.x = 1.0 ;     // x coor. of Line e1, Point p1 of tri
//...
}

```

### 8.1.17 Bit Fields

Before discussing the **union**, let's take a brief look at bit fields, which are similar in syntax to structures except that they allow a programmer to specify the number of bits for each data member. Consider the following program:

```

// bit_fld.cpp
// illustrates bit fields
#include <iostream.h> // C++ I/O

struct X
{
    unsigned int mem0 : 2 ;
    unsigned int mem1 : 6 ;

    int mem2 : 2 ;
};

void main ()
{
    X x ;

    cout << "sizeof x: " << sizeof x << endl ;

    // assign values to bit fields of x and display
    x.mem0 = 2 ; x.mem1 = 21 ; x.mem2 = -1 ;
    cout << "x.mem0: " << x.mem0
        << "; x.mem1: " << x.mem1
        << "; x.mem2: " << x.mem2 << endl ;

    // assign values to bit fields of x and display
    x.mem0 = 4 ; x.mem1 = 64 ; x.mem2 = 4 ;
    cout << "x.mem0: " << x.mem0
        << "; x.mem1: " << x.mem1
        << "; x.mem2: " << x.mem2 << endl ;
}

```

with output:

```

sizeof x: 2
x.mem0: 2; x.mem1: 21; x.mem2: -1
x.mem0: 0; x.mem1: 0; x.mem2: 0

```

The syntax of a bit field is similar to that of a structure, except that a colon and bit length are placed between the member name and the semicolon:

```
struct X
{
    unsigned int mem0 : 2 ;
    unsigned int mem1 : 6 ;

    int mem2 : 2 ;
};
```

which defines three bit fields to structure X. Bit field mem0 is allocated two bits, mem1 six bits and mem2 two bits. The number of bits allocated to each bit field is completely arbitrary and dependent on the particular application. The total number of bits in the three bit fields of **struct X** is ten (i.e. 1 byte and 2 bits), but note that the **sizeof** operator rounds up the size of object x.

A bit field must be of an integral type. Bit fields mem0 and mem1 are both of type **unsigned int**, whereas bit field mem2 is of type **int**. Because mem0 is allocated two bits and is **unsigned** it can hold values from 0 to 3 ( $2^4 - 1$ ), while mem1 can hold values from 0 to 63 ( $2^6 - 1$ ), since it is allocated six bits and **unsigned**. Although bit field mem2 is allocated two bits, it is of type **int** and can hold values from -1 to +1 (not 0 to 3), because a bit is reserved for the ± sign. If a value is assigned to a bit field which is outside the range allocated to the field, the result is implementation-dependent. The above program illustrates assigning values to the three bit fields of x that are outside the allocated ranges. The decision of the compiler used for the above program was to assign a value of 0 to the **unsigned int** and **int** bit fields.

The ability to specify exactly the number of bits for each member of a structure is very important when memory allocation is to be minimised.

## 8.2 Unions

A **union** is similar to a **struct** in that it is a collection of similar or dissimilar data types, but it holds only one of its members at any one time. The size of a **union** is the size of its largest data member. The general syntax of a **union** declaration is:

```
union Name
{
    typeSpecifier member_name ; // union members
    ...
    typeSpecifier member_name ;
};
```

A program example will help explain the **union**:

```
// union.cpp
// illustrates unions
#include <iostream.h> // C++ I/O

// declare a union 'UnionTypes'
union UnionTypes
```

```

{
char c ;
int i ;
float f ;
double d ;
};

void main ()
{
// define a union variable 'type'
UnionTypes type ;

// set type's data members
type.c = 'C' ; type.i = 2 ; type.f = 4.0 ; type.d = 8.0 ;

// O/P union type as initialised
cout << "type.c= " << type.c << ", type.i= " << type.i
    << ", type.f= " << type.f << ", type.d= " << type.d
    << endl ;

cout << endl ;

type.c = 'A' ;
cout << "type.c= " << type.c << ", type.i= " << type.i
    << ", type.f= " << type.f << ", type.d= " << type.d
    << endl ;

type.i = 12 ;
cout << "type.c= " << type.c << ", type.i= " << type.i
    << ", type.f= " << type.f << ", type.d= " << type.d
    << endl ;

type.f = 14.0 ;
cout << "type.c= " << type.c << ", type.i= " << type.i
    << ", type.f= " << type.f << ", type.d= " << type.d
    << endl ;

type.d = 18.0 ;
cout << "type.c= " << type.c << ", type.i= " << type.i
    << ", type.f= " << type.f << ", type.d= " << type.d
    << endl ;
}

```

which generates the output:

```

type.c= , type.i =0, type.f= 0, type.d= 8

type.c= A, type.i= 65, type.f= 9.10844e-44, type.d= 8
type.c= , type.i= 12, type.f= 1.68156e-44, type.d= 8
type.c= , type.i= 0, type.f= 14, type.d= 8
type.c= , type.i= 0, type.f= 0, type.d= 18

```

The output illustrates that the **union** variable `type` can be used to hold either a **char**, **int**, **float** or **double**, but only one of these four types at any given time. Each of the last four displayed lines has been output directly after the data members `c`, `i`, `f` and `d` of the **union** `UnionTypes` have been assigned the values 'A', 12, 14.0 and 18.0 respectively. The data members that have not been assigned a value hold garbage.

Note that a **union** can only be initialised through the **union**'s first data member:

```
UnionTypes type = { 'Z' } ;
```

### 8.2.1 **sizeof** a **union**

The following program code fragment outputs the size of a **struct** and a **union** with identical data members:

```
//...
// declare a union 'UnionTypes'
union UnionTypes
{
    char    c ;
    int     i ;
    float   f ;
    double d ;
};

// declare a structure 'StructTypes'
struct StructTypes
{
    char    c ;
    int     i ;
    float   f ;
    double d ;
};
//...
cout << "sizeof (StructTpyes) : " << sizeof (StructTypes) <<
endl ;
cout << "sizeof (UnionTpyes) : " << sizeof (UnionTypes) <<
endl ;
```

which would output the following:

```
sizeof (StructTypes): 15
sizeof (UnionTypes) : 8
```

The size of a **struct** is the sum of all of its data members ( $1+2+4+8=15$ ), whereas the size of a **union** is the size of its largest data member (8). When the `UnionTypes` variable `type` holds a **double** no bytes are unused. When `type` holds a **float** 4 bytes are unused, 6 bytes are unused when `type` holds an **int** and 7 bytes are unused when `type` holds a **char**.

Similarly to a **struct**, a **union** can be anonymous and can contain bit fields, but only one at any given time. A **union** cannot have **static** data members, although a **union** can have

member functions (including constructors and destructors). Unions cannot be used in a **class** hierarchy in that they cannot act as a base **class** or be derived from. The **class**, constructors, destructors and inheritance will be covered in later chapters.

### 8.2.2 Application of a union

A popular application of the **union** is for MS-DOS interrupt programming. The following extract from the DOS.H header file lists two structures WORDREGS and BYTEREGS and a **union** REGS:

```
// dos.h
//...
struct WORDREGS
{
    unsigned int ax, bx, cx, dx, si, di, cflag, flags ;
};

struct BYTEREGS
{
    unsigned char al, ah, bl, bh, cl, ch, dl, dh ;
};

union REGS
{
    WORDREGS x ;
    BYTEREGS h ;
};
```

WORDREGS simulates the 80x86 CPU registers and its data members represent two-byte register pairs. The BYTEREGS structure represents single-byte registers. The REGS **union** contains two data members of type WORDREGS and BYTEREGS. Hence the maximum storage allocation of a REGS variable is 16 bytes, since structure WORDREGS has eight data members each of size 2 bytes.

## 8.3 Enumerations

An **enum** is a user-defined type with named constants of type integer. For example:

```
enum
{
    RED, GREEN, BLUE
};
```

defines three constants RED, GREEN and BLUE equal to 0, 1 and 2 respectively. The data members of an **enum** are called enumerators and the default integer constant of the first enumerator in the list of data members is equal to 0. Note that, unlike **struct** and **union**, the

members of an **enum** are separated by the comma operator. The above enumeration is equivalent to defining the three constants, RED, GREEN and BLUE, as:

```
const int RED    = 0 ;
const int GREEN = 1 ;
const int BLUE   = 2 ;
```

An enumeration can be named:

```
enum WeekDays
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
};
```

Alternatively, the enumerators can be assigned constant values or constant expressions during the declaration:

```
enum Hexahedron
{
    VERTICES = 8,      // no. of vertices
    EDGES    = 12,     // no. of edges
    FACES    = 6,      // no. of faces
};
```

or, using a constant expression<sup>1</sup>:

```
enum Hexahedron
{
    VERTICES = 8,          // no. of vertices
    EDGES    = 12,          // no. of edges
    FACES    = 2 + EDGES - VERTICES, // no. of faces
};
```

The general syntax of the **enum** declaration is:

```
enum Name
{
    enumerator1, enumerator2, ..., enumeratorn
};
```

or

```
enum Name
{
    enumerator1 = constant_expression1,
    enumerator2 = constant_expression2,
```

---

<sup>1</sup> The Hexahedron enumeration makes use of Euler's formula  $V-E+F=2$  for a *simple* polyhedron, where  $V$  is the number of vertices,  $E$  the number of edges and  $F$  the number of faces.

---

```
    ...,
enumerator = constant_expression
};
```

Undoubtedly, the two most common uses of **enum** are the implementation of Switch and Boolean types:

```
enum Switch
{
OFF ;
ON ;
};
```

```
enum Boolean
{
FALSE ;
TRUE ;
};
```

Switch and Boolean lend themselves well to the **enum**, since they are essentially binary operations: on (1)-off (0) or true (1)-false (0). The following program illustrates the implementation and use of the Boolean enumeration:

```
// enum.cpp
// illustrates enum
#include <iostream.h> // C++ I/O
#include <conio.h> // getch()

// Boolean enumeration
enum Boolean
{
FALSE, TRUE
};

// returns TRUE if a vowel passed to function, else FALSE
Boolean Vowel (char c)
{
switch (c)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    return TRUE ;
default:
    return FALSE ;
}
}

void main ()
```

```

{
char ch = 'x' ; // initialise ch

do
{
    cout << endl << "enter a letter (non-vowel to stop): " ;
    ch = getche () ;
} while (Vowel (ch)) ;
}

```

and user interaction:

```

enter a letter (non-vowel to stop): a
enter a letter (non-vowel to stop): e
enter a letter (non-vowel to stop): z

```

Note the use of *fall-through* in the **switch** statement. The function *Vowel()* returns either TRUE (1) or FALSE (0), depending on whether a user enters a vowel or a consonant. The use of Boolean is more informative than simply using an integer:

```
Boolean Vowel (char c) ;
```

This function signature clearly indicates to a user that the function returns one of two possible values (0 or 1).

Finally, note that since **enum** is a ‘type of’ **struct** I adopt the same notation of using uppercase for the first letter in the enumeration name, e.g. Colour.

## 8.4 **typedefs**

A **typedef** allows a programmer to create a synonym for an existing integral or user-defined data type. **typedefs** are *not* new types.

The following program code illustrates a use of **typedef** to declare a new name, *Length*, for the integral data type **double**:

```

//...
typedef double Length ;
//...
Length Distance (const Point& p1, const Point& p2)
{
    //...
    Length length = sqrt (r_sq) ;
    return length ;
}

```

The **typedef** *Length* allows a programmer to give a more descriptive name to the return type of a *Distance()* function, which returns the straight distance (a **double**) between the two Points *p1* and *p2*.

**typedefs** can offer a programmer a convenient shorthand notation for lengthy types or types that are frequently used:

---

```
typedef char* String ; // pointer to char
typedef const char* CString ; // pointer to const char
```

The disadvantage of **typedefs** is that they introduce additional type names and can make existing types more abstract.

The following program illustrates a use of **typedef** with the Point structure:

```
// typedef.cpp
// illustrates the typedef
#include <iostream.h> // C++ I/O

const int NUM_VERTICES = 3 ;

// a Point structure for representing a 3D point
struct Point
{
    double x, y, z ;
};

void main ()
{
    // declare a typedef Polygon array
    // with 3 elements of type Point
    typedef Point Polygon[NUM_VERTICES] ;

    // define a Polygon variable
    Polygon poly ;

    // assign coordinate value to vertices
    poly[0].x = 1.0 ; poly[0].y = 1.0 ; poly[0].z = 0.0 ;
    poly[1].x = 3.0 ; poly[1].y = 2.0 ; poly[1].z = 0.0 ;
    poly[2].x = 3.0 ; poly[2].y = 4.0 ; poly[2].z = 0.0 ;

    // O/P poly vertices
    for (int i=0; i<NUM_VERTICES; i++)
    {
        cout << "poly[" << i << "]: (" 
            << poly[i].x << ", "
            << poly[i].y << ", "
            << poly[i].z << ")"
            << endl ;
    }
}
```

The following statements declare **Polygon** a synonym for an array of **Points** and defines a variable **poly** of type **Polygon**:

```
typedef Point Polygon[NUM_VERTICES] ;
//...
Polygon poly ;
```

### 8.4.1 **typedefs** for Windows

If you have experience with programming for Windows you may have noticed that virtually every data type is a **typedef**. The Windows **typedefs** are defined in the header file WINDOWS.H. A few are listed below:

```
// windows.h
//...
typedef unsigned char BYTE ;
typedef unsigned short WORD ;
typedef unsigned long DWORD ;
typedef unsigned int UINT ;
//...
```

These alternative data types were introduced to ease the porting of Windows applications to different architectures. As mentioned in Chapter 4, certain C++ data types are machine-specific and the alternative data types were developed to keep programs as independent as possible of machine architecture. This point is particularly important when developing both 16-bit and 32-bit Windows applications.

WINDOWS.H also contains several applications of the combined use of **typedef** and **struct**. The following program code illustrates two **typedefs**, RECT and POINT, which are structures that encapsulate the integer dimensions of a two-dimensional planar rectangle and the position of a two-dimensional point:

```
// windows.h
//...
typedef struct tagRECT
{
    int left ;
    int top ;
    int right ;
    int bottom ;
} RECT ;

typedef struct tagPOINT
{
    int x ;
    int y ;
} POINT ;
```

### 8.4.2 **typedefs** Offer Little Type Safety

Since a **typedef** is not a new type but a synonym for an alternative type, **typedefs** are weakly typed in C++ and offer little type safety. An example illustrates this:

```
typedef int Int ;
//...
int i = 1 ;
Int j = 2 ;
//...
```

---

```
j = i ; // mixing of types int and Int
```

This example illustrates that it is perfectly legal to mix an integral type with a **typedef**. This mixing of data types can be confusing and is prone to errors that are difficult to detect. Other languages, such as Ada and Eiffel, are more strict regarding the typing of fundamental data types than C++.

### 8.4.3 **typedef** Style

Several programmers adopt the style of using entirely uppercase letters for **typedefs**. The majority of programmers reserve uppercase identifiers exclusively for constants, and thus it is easy to confuse a **typedef** with a constant. The style that I adopt for naming a **typedef** is the same as for structures, unions, enumerations and classes, and reserves uppercase identifiers for constants.

## 8.5 Summary

A structure allows data members of the same or different types to be encapsulated in a single implementation, unlike arrays, which are aggregates of a single data type. The data members of a structure are, by default, publicly accessible, although a structure can have different member access rights other than **public**. In addition, a structure can in fact encapsulate functions which operate on the structure's data, but such capabilities are generally reserved for the **class**.

An important feature of user-defined structure types is their equivalence with C++'s integral data types. In the C programming language the name of a structure must be preceded by the **struct** keyword:

```
struct Person p ; // C-style
double d ;
```

C++ neglects the **struct** keyword:

```
Person p ; // C++-style
double d;
```

which illustrates that there is no syntactical difference made between integral and user-defined data types. Similarly, user-defined structures have the same capabilities as integral types with regard to definition, initialisation, arrays, function arguments and return type etc. In other words, and as will become apparent in the following chapters, a user-defined type can be designed to be as powerful and flexible as a C++ integral type.

The **union** data type is similar to **struct** in that it is a collection of data members of similar or different types, but unlike a **struct** definition, which allocates sufficient memory to hold all data members, a **union** can hold only one data member at any given time, and the size of a **union** is therefore the size of its largest member.

An enumerated list of integers can be generated in C++ by the enumeration keyword **enum**. Enumerated types are useful in assisting a programmer to attribute identifiers to constant values when the members of a data type map exactly to a set of integers, such as the Boolean **enum** with literals FALSE (0) and TRUE (1). Associating meaningful identifier names with

integer values can increase the clarity of programs, as illustrated by the combined use of **enums** and functions which return a Boolean logical value rather than an integer.

A **typedef** is not a new data type but a synonym for an existing type. The use of **typedefs** can introduce convenient, more descriptive, names to types.

## Exercises

- 8.1 Declare **Address** and **Telephone** structures to model a person's address and telephone number. Modify the **Person** structure presented in this chapter to include both **Address** and **Telephone** data members.
- 8.2 Declare a **City** structure which encapsulates the name, county, country and population of a city.
- 8.3 Debug the following program:

```
#include <iostream.h> // C++ I/O

struct Point
{
    double x, y, z ;
};

void main ()
{
    Point p ;

    p.x = 1.1 ; p.y = 2.3 ; p.z = 0.0 ;

    cout << "Point p: " << p << endl ;
}
```

- 8.4 Declare a **Circle** structure to represent a circle. Use structure **Point** in your declaration of **Circle**.
- 8.5 Develop a **Line** structure which uses a **Point** structure to model a straight line. Also, develop a **Triangle** structure which uses both **Point** and **Line** structures to model a triangle. Develop three functions which calculate the perimeter, centroid and area of a general three-dimensional triangle which is described in terms of three edges (Lines) and three vertices (Points). Test your program on a triangle whose three vertices are (1, 1, 0), (3, 2, 0) and (2.5, 4, 0).

The centroid,  $c$  ( $c_x, c_y, c_z$ ), of a triangle defined by the three points  $i$  ( $x_i, y_i, z_i$ ),  $j$  ( $x_j, y_j, z_j$ ) and  $k$  ( $x_k, y_k, z_k$ ) is given by:

$$c = \left( \frac{x_i + x_j + x_k}{3}, \frac{y_i + y_j + y_k}{3}, \frac{z_i + z_j + z_k}{3} \right)$$

**Hint:** Use the *Distance()* function in program POINT.CPP to enable you to calculate the perimeter length of the triangle:

```
double Distance (const Point& p1, const Point& p2) ;
```

- 8.6 Declare an **enum** called Colours which encapsulates various colours. Compare your Colours with the COLORS **enum** in the C++ library header file CONIO.H.
- 8.7 Declare a **typedef** called Percentage for representing percentages.

# The C++ Class

```
class Foo { /* ... */};
```

*The class is the key element for object-oriented programming in C++. Classes enable a programmer to create new types that have all the features of integral types. In fact, by applying the full features of classes, user-defined types can be designed to be as powerful, well-behaved and ‘concrete’ as integral types such as int.*

*The class is analogous to the structure, in that a class is a data encapsulation mechanism, but with the additional support of functions specific to the class which operate on the class’s data. Both class data and member functions can have associated access rights which allow the designer of a class to either allow or restrict access to specific data members to a non-class member or an object of the class. Thus, a class can be designed with an interface to non-class members which encapsulates the functionality of the class and at the same time hides the inner details of the class.*

*A class defines a mould from which objects or instances can be created, just as Jean is a particular instance of the class Person. An object is a single, unique entity that comprises both data and member functions which operate on their data.*

*Throughout this chapter, emphasis is placed on the development of a single class called Point which characterises a point in three-dimensions. As the chapter progresses we shall steadily see the functionality of class Point grow, so that when the chapter concludes Point will be a simple yet powerful class which reflects the majority of features that are characteristic of all C++ classes.*

*It is important to develop a good understanding of the class because it is the cornerstone of object-oriented programming in C++ and will play an increasing role in subsequent chapters.*



## 9.1 A Point class

This chapter introduces the C++ **class**, which is used in the creation of new user-defined types. Classes are introduced with the help of a Point **class**. Point is an excellent **class** to illustrate the key features of classes and is essential for graphics programming in C++.

Point-style classes are widely used in computer graphics. For example, in programming for Windows use is frequently made of the POINT structure defined in the WINDOWS.H header file:

```
// windows.h
//...
typedef struct tagPOINT
{
    int x ;
    int y ;
} POINT ;
```

The **typedef** POINT is extensively used to model a point on a two-dimensional display screen. Note that tagPOINT's data members are of type integer.

The Borland C++ (version 5.0) compiler supports an ObjectWindows library **class** TPoint. The **class** TPoint inherits the tagPOINT x and y data points and provides a sufficient number of overloaded operators to enable a natural use of TPoint objects:

```
// \OWL\POINT.H and \WINSYS\GEOMETRY.H
//...
class TPoint : public tagPOINT
{
public:
    // constructors
    TPoint () {}
    //...
};
```

Both tagPOINT and TPoint x and y data members are **public**. The distinction between **public** and **private** will be discussed later.

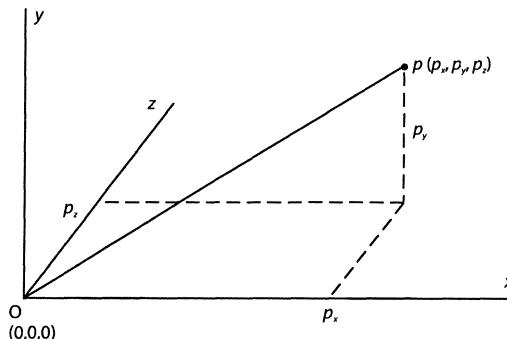
In this chapter we will be interested in a more accurate representation of a Point for general three-dimensional geometric modelling. Therefore, there are three data members for our Point **class**, each of type **double**. Type **double** can at first appear rather memory-expensive when compared with type **float**, but type **double** is necessary for the required level of accuracy. When we examine the C++ **template** we will observe that a generic Point **class** can be declared with data members whose types are determined when an object of the Point **class** is defined.

Figure 9.1 illustrates an arbitrary point,  $p$  ( $p_x, p_y, p_z$ ), in a three-dimensional space. The position of point  $p$  is given relative to a reference point or origin O. The origin O can have any assigned position, but it is generally assumed that O has coordinates (0, 0, 0). Point O is the origin of a three-dimensional coordinate system that consists of three mutually perpendicular straight lines or axes. Such a rectangular coordinate system is referred to as a *Cartesian* coordinate system.

Let's begin our examination of the Point **class** by copying the Point structure presented in the program POINT.CPP in Chapter 8 and simply replacing the keyword **struct** in the Point structure declaration by the keyword **class**:

```
// class_1.cpp
// introduces the class
#include <iostream.h> // C++ I/O

// Point class declaration
class Point
{
    double x, y, z ;
```



**Fig. 9.1** A point  $p$  in a three-dimensional Cartesian coordinate system ( $x, y, z$ ).

```

};

void main ()
{
// object p of class Point
Point p ;

p.x = 1.0 ; // error: Point's data members not accessible
p.y = 2.0 ;
p.z = 3.0 ;
}

```

Unfortunately, the above program will not compile. So why include a program that doesn't even compile? This program does not compile because the data members of **class** Point are not accessible to an object, p, of **class** Point. To see why let's first look at the **class** declaration of Point:

```

class Point
{
double x, y, z ;
};

```

It was noted in Chapter 8 that the data members of a **struct** are by default **public**. Thus an object of **struct** Point has access to Point's three data members. The data members of a **class** are by default **private**, which means that a **class**'s data members can only be accessed from within the **class**. Although **private** is the default, it is good programming practice to indicate explicitly which members are **private**:

```

class Point
{
private:
double x, y, z ;
};

```

Note the colon (:) directly after the keyword **private**.

**private** and **public** are keywords that signal which data members are only accessible from within a **class** or have general outside access, respectively. For example, the following **class** declaration illustrates that the data member `pri_data` is not accessible to `object` but `pub_data` is:

```
// pri_pub.cpp
// illustrates private and public data members
#include <iostream.h> // C++ I/O

// Name class declaration
class Name
{
    private:           // private data member
        int pri_data ;
    public:            // public data member
        int pub_data ;
};

void main ()
{
    // define object of class Name
    Name object ;

    object.pub_data = 1 ; // O.K.: public data member
                        //           is accessible

    object.pri_data = 1 ; // error: private data member
                        //           is not accessible
}
```

Within the delimiting braces { and } of the **class** declaration, all data members following the keyword **private** are **private** unless the keyword **public** is encountered, and vice versa. In addition, several **private** or **public** keywords can be placed within a **class** declaration:

```
class Name
{
    private:
        int pri_data1 ;
        int pri_data2 ;
    public:
        int pub_data1 ;
//...
    private:
        int pri_data3 ;
//...
//...
    public:
        int pub_data2 ;
//...
//...
```

---

```
}
```

This can be a useful feature when dealing with a large **class** declaration, and is in agreement with C++'s *define-anywhere* style of defining an identifier wherever it is required, rather than the C *define-at-top* style.

So why the emphasis on **private** and **public**? The keyword **private** enables a programmer to hide certain data or algorithms specific to a **class**'s implementation from external variables, objects, classes, functions etc. Data hiding is one of the key features of the C++ **class** and of object-oriented programming and restricts the access of data that is prescribed as **private**.

## 9.2 class Declaration Syntax

As a recap let's look at the general syntax of a **class** declaration. Later, this syntax will be extended to incorporate member functions, constructors and destructors, but for now here it is:

```
class ClassName
{
    typeSpecifier pri_member_name ;      // class private members
    //...
private:
    typeSpecifier pri_member_name ; // class private members
    //...
public:
    typeSpecifier pub_member_name ; // class public members
    //...
} object_list ;
```

The **class** declaration begins with the keyword **class**. This is directly followed by the user-defined name of the **class**. Data members are by default **private**, but can be explicitly indicated to be **private** by the keyword **private**. Public data members are similarly preceded by the **public** keyword. Both of the **private** and **public** keywords have an associated colon (:). **private** or **public** data members can be of either an integral or a valid user-defined type. The syntactical rules of naming **private** or **public** data members are identical to those of naming *normal* variables and objects. The body of a **class** is delimited by the braces { and } with the closing brace, }, directly followed by a semicolon (;). It was noted in Chapter 8, when we discussed declaring and defining a structure in a single statement, that we could define an object between the closing brace of a structure declaration and the semicolon. Similarly, it is possible to declare and define a **class** in a single statement:

```
class Point
{
    //...
} p, q, r ;
```

which defines the objects p, q and r of **class** Point.

The style adopted for **class** declarations is similar to that of **struct** declarations. The **class** body braces are indented a single horizontal tab position from the **class** keyword.

Similarly, the **private** and **public** members of a **class** are indented a single tab position from the **private** and **public** keywords. If both of the keywords **private** and **public** appear only once within a **class** declaration, I always place the **class's private** data members at the top of the declaration. Different people adopt different styles, and one of the more popular alternatives is:

```
class Name {  
    int pri_data ;  
    //...  
public:  
    int pub_data ;  
    //...  
};
```

As always, choose what's best for you.

## 9.3 Objects and Instances

Above we saw the following definition:

```
Point p ;
```

This defines an *object* or *instance* *p* of **class** *Point*. Object *p* is a particular instance of **class** *Point*. The class *Point* is a type of *mould* that encapsulates the essential features of a three-dimensional point in space, whereas we act on an object *p* that has *Point*'s characteristics but holds its own unique set of data.

## 9.4 A public Point class

Having discussed the keywords **private** and **public**, let's now return to the *Point* **class** presented in program CLASS\_1.CPP, but now make **class** *Point*'s data members **public** so that they are accessible to objects of *Point*:

```
// pubpoint.cpp  
// a Point class with public data members  
#include <iostream.h> // C++ I/O  
  
// class Point  
class Point  
{  
public:           // public data members  
    double x, y, z ;  
};  
  
void main ()  
{
```

```

// define objects p and q
Point p, q;

// assign values to objects p and q data members
p.x = 1.0; p.y = 2.0; p.z = 3.0;
q.x = 4.0; q.y = 5.0; q.z = 6.0;

// O/P objects
cout << "object p (" 
    << p.x << ", " << p.y << ", " << p.z << ")" << endl
    << "object q (" 
    << q.x << ", " << q.y << ", " << q.z << ")" << endl ;
}

```

with output:

```

object p (1, 2, 3)
object q (4, 5, 6)

```

The output of program PUBPOINT.CPP illustrates that the **public** data members *x*, *y* and *z* of **class** Point are now accessible to the objects *p* and *q*, which are both of type Point. Thus, the **class** Point now acts as a **struct** since data members of a **struct** are **public** by default.

Objects *p* and *q* are defined in the statement:

```
Point p, q;
```

and then values are assigned to each object's data members:

```

p.x = 1.0; p.y = 2.0; p.z = 3.0;
q.x = 4.0; q.y = 5.0; q.z = 6.0;

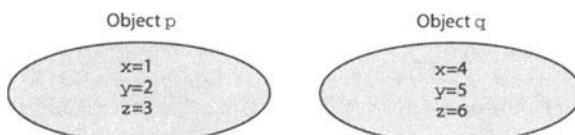
```

Figure 9.2 schematically illustrates the two objects *p* and *q*. Each object has its own *x*, *y* and *z* data members. A common source of confusion to programmers who are new to object-oriented programming is thinking that several different objects of a given **class** all access the same data, as shown in Fig. 9.3. This incorrect conceptualisation is analogous to the idea of different functions accessing the same global data; this is also illustrated in Fig. 9.3.

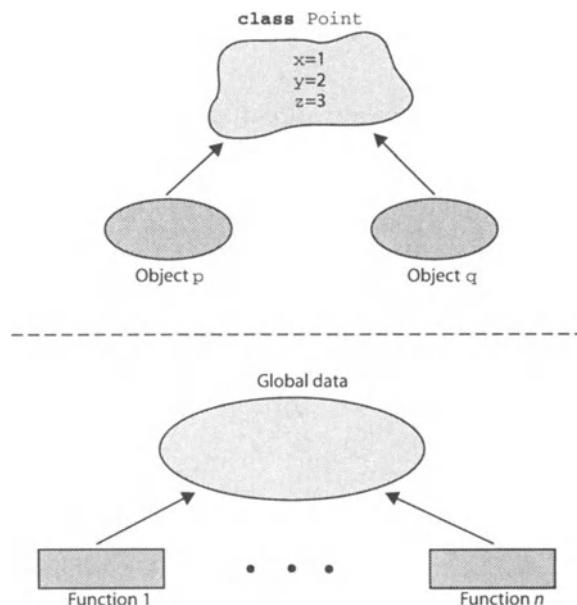
It must be reiterated that:

*Each object carries its own unique set of data.*

An object is a single entity with its own characteristics and not simply a pointer to some global data bank.



**Fig. 9.2** Two objects *p* and *q* of **class** Point.



**Fig. 9.3** Incorrect conceptualisation of objects.

To illustrate this important point further, try adding the following two lines to the above program PUBPOINT.CPP:

```
cout << "sizeof p: " << sizeof p << endl
      << "sizeof q: " << sizeof q << endl ;
```

which will generate the output:

```
sizeof p: 24
sizeof q: 24
```

This output illustrates that the size of each object is 24 ( $3 \times 8$ ) bytes, since the size of type **double** is 8 bytes and **class Point** has three **double** data members. Thus each object is allocated sufficient storage to hold its own data.

Also, when objects **p** and **q** are defined to be of type **Point**, the **class Point** declaration does not alter. The **class Point** declaration is a mould or a specification for **Point** objects.

## 9.5 Member Functions

In the previous section, the data members of **class Point** were made **public** so that an object has access to the data members. This enabled us to assign values and to access (via the `cout` output stream object) each object's data members. Thus, our present design of the **Point** **class** appears to be working well, but there is a problem! Assigning **public** access to **Point**'s data members unfortunately makes them easily accessible to objects of **class Point** from outside the **class** declaration. This leaves **Point**'s data members vulnerable to accidental alteration.

When learning a programming language, via small programming exercises, for the first time it is difficult to appreciate the importance of data hiding. As programming projects become larger, with a corresponding increase in the number of object interactions, it becomes increasingly important to reduce, if not eliminate entirely, the risk of an object being accidentally altered or altered in ways not intended by the designer of the object's **class**. By making certain **class** data members **private** we can prescribe the manner in which an object's data is to be manipulated by specifying a **public** interface for the object.

So how do we solve the dilemma of assigning **private** access to a **class**'s data members while at the same time making them accessible to an object? The answer lies in **class member functions**<sup>1</sup>. Assigning **private** access to a **class**'s data members and simultaneously assigning **public** access to member functions allows an object *indirect* access to its data members while ensuring data-member safety. Another reason for the hiding of data members is to ensure that the details of a **class**'s implementation are not disclosed. To illustrate this let's extend the **Point class** by making the **x**, **y** and **z** data members **private** and introducing **set** and **get** member functions of **class Point**:

```
// mempoint.cpp
// illustrates class member functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{
    private: // private data members
    double x, y, z ;
    public: // public member functions
    void SetX (double x_arg)
    { x = x_arg ; }
    void SetY (double y_arg)
    { y = y_arg ; }
    void SetZ (double z_arg)
    { z = z_arg ; }
    double GetX ()
    { return x ; }
    double GetY ()
    { return y ; }
    double GetZ ()
    { return z ; }
}; // class Point

void main ()
{
    // define objects p and q
    Point p, q ;

    // assign values to objects p and q data members
    // by aid of Point's member functions
    p.SetX (1.0) ; p.SetY (2.0) ; p.SetZ (3.0) ;
```

---

<sup>1</sup> Member functions are frequently referred to as *methods*, *operations* or *messages* in the literature.

```

q.SetX (4.0) ; q.SetY (5.0) ; q.SetZ (6.0) ;

// O/P objects
// access objects data members
// by aid of Point's member functions
cout << "object p (" 
    << p.GetX () << ", " << p.GetY () << ", "
    << p.GetZ () << ")" << endl
    << "object q (" 
    << q.GetX () << ", " << q.GetY () << ", "
    << q.GetZ () << ")" << endl ;
}

```

There are two types of member function in **class** Point, *Set?*(*)* and *Get?*(*)*, where ? is equal to X, Y or Z. The following illustrates these for setting and getting the x data member:

```

void SetX (double x_arg)
{ x = x_arg ; }
//...
double GetX ()
{ return x ; }

```

Note that all member functions are, at present, defined within the **class** declaration, i.e. within the delimiting braces of the **class** declaration. The member functions are function definitions, and defined within the **class** declaration are *inline* functions; refer to Chapter 6. For small member functions, such as the *set* and *get* functions above, it is convenient to define them within a **class** declaration, but for functions with larger function bodies a member function definition is generally placed outside the **class** declaration. Out-of-line member function definitions are described later.

The *SetX()* member function sets the value of the x data member of **class** Point to the function argument, *x\_arg*, that is passed by value. The *GetX()* member function simply returns an object's x data member.

If you recall from Chapter 6, which dealt with *normal* functions, C++ allows us to overload functions. Thus, we are able to assign different operations to the same function name. This is illustrated in the following program:

```

// mem_over.cpp
// illustrates overloaded class member functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private: // private data members
    double x, y, z ;
public: // public member functions
    void X (double x_arg)
    { x = x_arg ; }
    void Y (double y_arg)
    { y = y_arg ; }
    void Z (double z_arg)

```

```

    { z = z_arg ; }
double X ()
    { return x ; }
double Y ()
    { return y ; }
double Z ()
    { return z ; }
}; // class Point

void main ()
{
// define objects p and q
Point p, q ;

// assign values to objects p and q data members
// by aid of Point's member functions
p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;
q.X (4.0) ; q.Y (5.0) ; q.Z (6.0) ;

// O/P objects
// access objects data members
// by aid of Point's member functions
cout << "object p (" 
    << p.X () << ", " << p.Y () << ", "
    << p.Z () << ")" << endl
    << "object q (" 
    << q.X () << ", " << q.Y () << ", "
    << q.Z () << ")" << endl ;
}

```

Overloading the member functions *X()*, *Y()* and *Z()* has now eliminated the need for the *set* and *get* member function names.

You may be thinking that using member functions is a long-winded way of accessing a **class**'s data members. However, for the increased safety that member functions offer a programmer they are well worth it. In fact, let's compare the two opposing views of making Point's data members **private** and **public**:

```

// public class data members:
Point p ;
//...
p.x = 1.0 ;
//...
cout << "object p (" << p.x //...

```

or

```

// private class data members
Point p ;
//...
p.X (1.0) ;
//...

```

```
cout << "object p (" << p.X() //...
```

Put this way, we see that there is little extra effort required from a user's perspective by making **Point**'s data members **private**, although greater implementation is required. Further, when we examine constructors shortly we shall see that objects can be defined and initialised more cleanly:

```
// private class data members
Point p (1.0, 2.0, 3.0) ;
//...
cout << "object p (" << p.X() //...
```

## 9.6 Naming of Member Functions

The convention adopted for the naming of **class** member functions is identical to the naming of *normal* functions, in that the first letter(s) of single or multi-word names are in uppercase. For example:

```
p.GetX () ;
```

## 9.7 Calling Member Functions

In the program MEM\_OVER.CPP an object **p** of **class Point** called the member function **X()** with the following statement:

```
Point p ;
//...
p.X () ;
```

Notice that object **p** is connected to **class Point**'s member function **X()** by the *dot operator*. It makes no sense to write the following statement:

```
//...
X () ;
```

because the function **X()** is a member of **class Point**.

The **public** member function **X()** is accessible only to objects of **class Point**, and thus the function call is associated with an object **p** of **class Point** by the dot operator or the *member access operator* (**.**). In general a **class**'s member function, **MemberFunction()**, is accessed by an object of the **class** by:

```
object.MemberFunction (parameter_list) ;
```

The function call:

```
p.X (1.0) ;
```

---

causes the `x` data member of object `p` to be assigned the value of 1.

## 9.8 Defining Member Functions

The program `MEM_OVER.CPP` defined `Point`'s member functions `X()`, `Y()` and `Z()` within the `class` declaration:

```
// mem_over.cpp
//...
// class Point
class Point
{
private:           // private data members
    double x, y, z;
public:            // public member functions
    void X (double x_arg)
    { x = x_arg; }
    //...
    //...
};
```

This is OK, but as the number of member functions increases and the size of certain member functions increases due to increased complexity it is desirable to define member functions outside a `class` declaration. The following program illustrates this for the `Point` `class` presented in the file `MEM_OVER.CPP`:

```
// outside.cpp
// a Point class with member functions
// defined outside of the class declaration
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:           // private data members
    double x, y, z;
public:            // public member functions
    void X (double x_arg) ;
    void Y (double y_arg) ;
    void Z (double z_arg) ;
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// class Point member function definitions:

void Point::X (double x_arg)
```

```

{ x = x_arg ; }

void Point::Y (double y_arg)
{ y = y_arg ; }

void Point::Z (double z_arg)
{ z = z_arg ; }

double Point::X ()
{ return x ; }

double Point::Y ()
{ return y ; }

double Point::Z ()
{ return z ; }

void main ()
{
// define objects p and q
Point p, q ;

// assign values to objects p and q data members
// by aid of Point's member functions
p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;
q.X (4.0) ; q.Y (5.0) ; q.Z (6.0) ;

// O/P objects
// access objects data members
// by aid of Point's member functions
cout << "object p ("
<< p.X () << ", " << p.Y () << ", "
<< p.Z () << ")" << endl
<< "object q ("
<< q.X () << ", " << q.Y () << ", "
<< q.Z () << ")" << endl ;
}

```

Observe how elegant our Point **class** declaration is in the above program now that the member functions are defined outside the **class** declaration. The member function **X()** is now declared within the Point **class** declaration:

```

class Point
{
private:           // private data members
    double x, y, z ;
public:            // public member functions
    void X (double x_arg) ;
    //...
};

```

Such a member function declaration indicates to the compiler that the member function *X()* is defined outside the **class** declaration. In the program OUTSIDE.CPP the member function definition of *X()* is placed after the **class** declaration:

```
//...
void Point::X (double x_arg) // X() definition
{ x = x_arg ; }
```

The syntax of a member function declarator defined outside a **class** declaration is:

```
return_type ClassName::MemberFunction (parameter_list)
```

The first part of the declarator is the return type of the member function and is then followed by the **class** name to which the member function belongs. Two colons placed together ( :: ) are called the *scope resolution operator*. If *ClassName::* is removed from the member function declarator we return to a normal function declarator.

In addition, a **class** declaration is conventionally placed in a header file, .H, and the member function definitions are placed in an implementation file,.CPP. Consider the **Point** **class** declaration placed in the header file PT\_LIB.H:

```
// pt_lib.h
// header file for Point class

#ifndef _PT_LIB_H // prevent multiple includes
#define _PT_LIB_H

// class Point
class Point
{
    private:           // private data members
    double x, y, z ;
    public:          // public member functions
    void X (double x_arg) ;
    void Y (double y_arg) ;
    void Z (double z_arg) ;
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

#endif // _PT_LIB_H
```

and the **class** member functions defined in the file PT\_LIB.CPP:

```
// pt_lib.cpp
// implementation file for Point class
#include "pt_lib.h" // Point class

// class Point member function definitions:

void Point::X (double x_arg)
```

---

```

{ x = x_arg ; }

void Point::Y (double y_arg)
{ y = y_arg ; }

void Point::Z (double z_arg)
{ z = z_arg ; }

double Point::X ()
{ return x ; }

double Point::Y ()
{ return y ; }

double Point::Z ()
{ return z ; }

```

We can now simply include the header file PT\_LIB.H in a program MY\_PROG.CPP which defines objects p and q of **class** Point and assigns and accesses their respective x,y and z data members:

```

// my_prog.cpp
// application of Point class
#include <iostream.h> // C++ I/O
#include "pt_lib.h" // Point class header file

void main ()
{
    // define objects p and q
    Point p, q;

    // assign values to objects p and q data members
    // by aid of Point's member functions
    p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;
    q.X (4.0) ; q.Y (5.0) ; q.Z (6.0) ;

    // O/P objects
    // access objects data members
    // by Point's member functions
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
        << "object q ("
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl ;
}

```

The separation of **class** declarations and implementations into separate files is important for large programming projects which contain numerous separate program components developed independently. Although different classes and program components will interact in a program, they are generally developed independently, either one at a time by a single

programmer or simultaneously by a group of programmers. The development of classes from problem domain abstractions can be similarly reflected in the organising of program files arranged in **class** hierarchies. Placing **class** declarations in header files not only assists in the reuse of classes, but eliminates entire program recompilations (which can be a time-consuming process for large programming projects) resulting from a change in a single **class**. Separate program components can be compiled and tested largely independently and brought together into a single executable program with Link and Make utilities.

## 9.9 static Data Members and Member Functions

A data member or member function can be declared as **static** in a **class**'s declaration. To illustrate the implications of **static** data and function members consider the following program, which is a modification of program OUTSIDE.CPP:

```
// static.cpp
// illustrates static data and function members of a class
//...
class Point
{
    private:           // private static data members
    static double x, y, z ;
    public:          // public member functions
    //...
    // static member function
    static void Member () ;
}; // class Point

// class Point member function definitions:
//...
void Point::Member ()
{
    cout << "(" << x << ", " << y << ", " << z << ")" << endl ;
}

// global definitions of Point's data members
double Point::x ;
double Point::y ;
double Point::z ;

void main ()
{
    // define objects p and q
    Point p, q ;

    // O/P objects
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
```

```

    << "object q ("
    << q.X () << ", " << q.Y () << ", "
    << q.Z () << ")" << endl ;

// assign values to object p
p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;

// O/P objects
cout << "object p ("
    << p.X () << ", " << p.Y () << ", "
    << p.Z () << ")" << endl
    << "object q ("
    << q.X () << ", " << q.Y () << ", "
    << q.Z () << ")" << endl ;

p.Member () ;
}

```

with output:

```

object p (0, 0, 0)
object q (0, 0, 0)
object p (1, 2, 3)
object q (1, 2, 3)
(1, 2, 3)

```

The x,y and z data members of **class Point** are declared as **static**:

```

class Point
{
    private:           // private static data members
    static double x, y, z ;
    //...
};

```

The implication of declaring a **class**'s data members **static** is that **static** data members of a **class** are shared by *all* objects of the **class**. The **static** data members of an uninitialised object are initialised to zero. We shall see later, in the program DEF\_CON.CPP, that this is not the case for non-**static** data members. Thus, the above program output illustrates that following the uninitialised definitions of objects p and q in *main()*, the x, y and z data members of objects p and q are zero. Next, floating-point values are assigned to the data members of object p in *main()*:

```

//...
p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;

```

but the output illustrates that the x, y and z data members of both objects p and q have been set to 1, 2 and 3, respectively. This is because, as mentioned above, a **class**'s **static** data members are shared by all objects of a **class**. Hence, if we alter the data members of object p we also alter the data members of object q.

It is important to note that when a **class**'s data members are declared as **static** they are not defined. Global definitions must be provided for each **static** data member of a **class**. This is performed in program STATIC.CPP by the following global definitions:

```
// global definitions of Point's data members
double Point::x ;
double Point::y ;
double Point::z ;
```

The scope resolution operator ( :: ) associates each **static** data member with its respective **class**.

Program STATIC.CPP also declares a **static** member function of **class** Point:

```
static void Member () ;
```

This function simply displays an object's data members. Static member functions only have access to **static** data members, cannot access non-**static** data members and cannot call non-**static** member functions.

Static data and function members are rarely used, but there are occasions when they prove to be very useful. For instance, **static** data members can eliminate the use of global variables.

If you have wondered whether you can use the **auto**, **extern** and **register** storage class specifiers as **class** data members, the answer is simply no:

```
// au_ex_re.cpp
// illustrates that the storage class specifiers
// auto, extern and register cannot be used for
// declaring class data members

class X
{
private:
    auto      int data0 ; // error
    extern    int data1 ; // error
    register int data2 ; // error
};

void main ()
{
}
```

## 9.10 **const** and **mutable** Data Members

The data members of a **class** can be declared as **const**:

```
class X
{
private:
    const int c_data ;
```

```
public:
    X ()
        : data (0) {}
    //...
};
```

Note, however, that a data member declared as **const** must be initialised in the constructor declarator and cannot be initialised via an assignment statement in the body of the constructor.

The ANSI/ISO draft standard defines a new keyword, **mutable**, which is used to allow a member of an object to cast away **const**:

```
// con_mut.cpp
// illustrates const and mutable data members

class X
{
    public:
        int           data ;
        const int      c_data ; // const data member
        mutable int   m_data ; // mutable data member

        mutable const int mc_data ; // error: mutable const
        mutable static int ms_data ; // error: mutable static
    X ()
        : data (0), c_data (0), m_data (0) {}
};

void main ()
{
    X         x ; // non-const object
    const X cx ; // const object

    x.data     = 1 ; // o.k.: non-const object, non-const member
    x.c_data   = 2 ; // error: non-const object, const member
    cx.data    = 3 ; // error: const object, non-const member
    cx.c_data = 4 ; // error: const object, const member

    // o.k.: mutable member of const object can be modified
    cx.m_data = 5 ;
}
```

The above program illustrates that a **const class** data member or **const** object cannot be altered. However, if a data member is declared **mutable** then the data member of a **const** object is not **const** and can therefore be modified. The **mutable** keyword is available because when a **const** object is defined this should not necessarily imply that the object's data members are **const**. The above program also illustrates that the **mutable** specifier cannot be applied to **const** or **static class** data members.

Note in the above **class** declaration that since X declares a **const** data member (**c\_data**) and **const** objects must be initialised when they are defined, a constructor is required to initialise the X::**c\_data** data member explicitly. **const** member functions are discussed separately in a later section.

## 9.11 volatile Data Members

The data members of a **class** can be declared **volatile**:

```
// vol.cpp
// illustrates volatile data members
#include <iostream.h> // C++ I/O

class X
{
public:
    int data ;
    volatile int v_data ;
    const volatile int cv_data ;
    X ()
        : data (0), v_data (0), cv_data (0) {}
};

void main ()
{
    X x ;
}
```

The above program also illustrates that a data member can be both **const** and **volatile**. **volatile** member functions are discussed in a later section.

## 9.12 Bit Fields

Before discussing constructors in the next section let us just note that a **class** declaration, similarly to a **struct** declaration, can declare bit fields. The following program is the **class** version of BIT\_FLD.CPP presented in Chapter 8:

```
// bit_fld.cpp
// illustrates bit fields

class X
{
private:
    unsigned int mem0 : 2 ;
    unsigned int mem1 : 6 ;
public:
    X ()
        : mem0 (0), mem1 (0) {}
};

void main ()
{
    X x ;
}
```

## 9.13 Constructors

To date, we have defined objects of **class** Point and then assigned values to each object's data members by using member functions. For instance, the following is a typical extract from OUTSIDE.CPP:

```
//...
// define objects p and q
Point p, q;

// assign values to objects p and q data members
// by using Point's member functions
p.X (1.0) ; p.Y (2.0) ; p.Z (3.0) ;
q.X (4.0) ; q.Y (5.0) ; q.Z (6.0) ;
```

C++ provides *special* member functions called constructors<sup>2</sup> which allow us to initialise an object's data members when an object is defined. Let us illustrate the use of a constructor on our Point **class**:

```
// constr.cpp
// illustrates constructors
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructor
    Point (double x_arg, double y_arg, double z_arg) ;
    // public member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// constructor
Point::Point (double x_arg, double y_arg, double z_arg)
{
    x = x_arg ;
    y = y_arg ;
    z = z_arg ;
}

// class Point member function definitions:
```

---

<sup>2</sup> Member functions are frequently referred to as *instance methods*, since they operate on objects of a **class**, whereas constructors are *class methods*.

```

double Point::X ()
{ return x ; }

double Point::Y ()
{ return y ; }

double Point::Z ()
{ return z ; }

void main ()
{
    // define objects p and q
    Point p (1.0, 2.0, 3.0) ;
    Point q (4.0, 5.0, 6.0) ;

    // O/P objects
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
        << "object q ("
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl ;
}

```

The Point constructor declaration is:

```
Point (double x_arg, double y_arg, double z_arg) ;
```

which has three arguments that initialise the three data members of **class** Point. The constructor name is the same as the **class** name – this is how the compiler knows which member functions are constructors. Therefore, a member function cannot have the same name as the **class** name unless it is a constructor. In general, a **class** and constructor declaration are of the form:

```

class ClassName
{
    private:
    //...
    public:
    // constructor
    ClassName (parameter_list) ;
    //...
};

```

The Point **class** declaration above indicates that constructors do not return a value, since there is no return type. An attempt to declare a constructor with the keyword **void** will result in a compilation error of the form ‘constructor cannot have a return type’:

```
void Point () ; // error: constructor cannot have a return
                // type
```

Similarly, it makes no sense to call a constructor as if it is an *ordinary* member function:

```
cout << p.Point() ; // error: Point() member function does  
// not exist
```

Such a call will generate a compilation error of the form ‘member function does not exist’.

Since the three-argument constructor of **class** Point defines an object, the set X(), Y() and Z() member functions are not required. The get X(), Y() and Z() access member functions are still required to access the x, y and z data members of a Point object.

Constructors must generally be declared as **public**. To illustrate why, consider the following:

```
class Point  
{  
private:  
    double x, y, z ;  
    // private constructor  
    Point (double x_arg, double y_arg, double z_arg) ;  
public:  
    //...  
};  
//...  
Point q (4.0, 5.0, 6.0) ; // error: constructor not  
// accessible
```

The definition of object q will produce a compilation error of the form ‘Point::Point() constructor is not accessible’. Since object q is defined outside the Point **class** declaration, the three-argument constructor has to be **publicly** accessible.

Note that constructors can be defined outside the **class** declaration. Constructors, like member functions, can be defined inside or outside a **class**’s declaration.

### 9.13.1 Constructors are Called Automatically

Provided **class** Point implements the appropriate constructor, a definition of an object p of **class** Point, given by:

```
Point p (1.0, 2.0, 3.0) ;  
//...
```

will instruct the compiler to call the respective constructor when an object is created. The general syntax for defining an object is:

```
ClassName object (parameter_list) ;
```

Thus, constructors are called automatically and there is no need to define additional member functions explicitly for object initialisation.

Alternatively, object p could be defined via assignment:

```
Point p = Point (1.0, 2.0, 3.0) ;
```

However, such object definitions are rarely used, since the previous form offers a more concise definition.

It is worth noting that the above object definition–assignment statement:

```
Point p = Point (1.0, 2.0, 3.0) ;
```

is actually converted to:

```
Point p (1.0, 2.0, 3.0) ;
```

We shall continue the discussion of implicit conversions when we examine conversion functions and converting constructors in Chapter 10.

### 9.13.2 Default Initialisation

The above use of a three-argument constructor for the `Point class` illustrated how an object can be initialised at the same time as it is defined. It would be nice for a user of our `Point class` to be able simply to define an object of `class Point` and be assured of the values of its data members. This is possible if we define a no-argument constructor (using empty parentheses or parentheses with the keyword `void`) and assign values to the `class` data members:

```
//...
Point::Point ()
{
    x = y = z = 0.0 ;
}
```

This program code initialises the `x`, `y` and `z` data members of `Point` to zero. Note that initialisation can assign any value(s) to a `class`'s data member(s), and not necessarily zero. It is tempting to try to initialise a `class`'s data members in the `class` declaration rather than in a `class` constructor:

```
class Point
{
private:
    // error: can't initialise data members in declaration
    double x, y, z = 0.0 ;
    //...
};
```

This is illegal in C++ and initialisation of data members is restricted to constructors or member functions.

Once a default constructor of `class Point` has been defined, an object of `class Point` can be defined as follows:

```
void main ()
{
    //...
    Point p ; // no parentheses or arguments
```

```
//...
}
```

Object p is defined without parentheses or arguments, and not as follows:

```
//...
Point p () ; // function declaration
```

which declares a function called p that has no arguments and returns a Point object.

### 9.13.3 Overloaded Constructors

We have already seen that member functions can be overloaded, and since constructors are merely *special* member functions it is natural to overload constructors. Overloading constructors enables an object of a given **class** to be initialised differently when defined. Combining the above observations of default initialisation and the Point **class** in program CONSTR.CPP we have the program:

```
// over_con.cpp
// illustrates overloaded constructors
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    // public access member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// constructors:

// no arg. constructor (default)
Point::Point ()
{
    x = y = z = 0.0 ;
}

// 3 arg. constructor
Point::Point (double x_arg, double y_arg, double z_arg)
{
    x = x_arg ;
    y = y_arg ;
    z = z_arg ;
}
```

```

z = z_arg ;
}
//...
void main ()
{
// define objects p and q
Point p (1.0, 2.0, 3.0) ;
Point q ;

// O/P objects
cout << "object p ("
<< p.X () << ", " << p.Y () << ", "
<< p.Z () << ")" << endl
<< "object q ("
<< q.X () << ", " << q.Y () << ", "
<< q.Z () << ")" << endl ;
}

```

In this program an object q is defined to be of **class** Point, but it is not explicitly initialised:

```
Point q ;
```

The default no-argument constructor is called for such an object definition of **class** Point. The output of the program is:

```
object p (1, 2, 3)
object q (0, 0, 0)
```

which illustrates that each of q's three data members is assigned the value 0.

#### 9.13.4 Default Constructors

If constructors are not defined for a given **class** then objects of the **class** can be defined, but the values of an object's data members are not guaranteed. The following program is similar to OVER\_CON.CPP except that no constructors for **class** Point are defined:

```

// def_con.cpp
// illustrates the use of default constructors
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // public member functions
    //...
}; // class Point
//...

```

```

void main ()
{
    // define objects p and q
    Point p, q;

    // O/P objects
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
        << "object q ("
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl ;
}

```

which generated the following output when I ran this program:

```

object p (3.48121e-35, 5.99741e-33, 2.47093e-157)
object q (1.53015e-138, 6.73993e-311, 1.6991e-309)

```

Try adding the three-argument Point constructor declaration to the **class** Point declaration and the outside constructor definition listed in OVER\_CON.CPP to the above program DEF\_CON.CPP:

```

class Point
{
    //...
    Point (double x_arg, double y_arg, double z_arg) ;
    //...
};

//...
Point::Point (double x_arg, double y_arg, double z_arg)
{
    //...
}

//...
void main ()
{
    Point p, q; // error: no match found for Point::Point()
    //...
}

```

Such a program will generate a compilation error of the form ‘No match found for the constructor `Point::Point()`’. This program helps illustrate that either no constructors can be defined or constructors must be explicitly defined if a constructor is defined with one or more arguments. Generally, it is good programming practice always to define at least a no-argument constructor for each **class** that you develop.

### 9.13.5 Data Member Initialisation

Program OVER\_CON.CPP illustrated that Point’s three data members can be assigned values when an object of **class** Point is defined by means of a three-argument constructor:

---

```
//...
Point::Point (double x_arg, double y_arg, double z_arg)
{
    x = x_arg ;
    y = y_arg ;
    z = z_arg ;
}
```

There is a neater way to initialise Point's data members:

```
// con_init.cpp
// illustrates data member initialisation
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    // public member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// constructors:

// no arg. constructor (default)
Point::Point ()
    : x(0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
Point::Point (double x_arg, double y_arg, double z_arg)
    : x (x_arg), y (y_arg), z(z_arg)
{ }
//...
void main ()
{
    // define objects p and q
    Point p (1.0, 2.0, 3.0) ;
    Point q ;

    // O/P objects
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" ;
}
```

```

    << p.Z () << ")" << endl
    << "object q ("
    << q.X () << ", " << q.Y () << ", "
    << q.Z () << ")" << endl ;
}

```

Point's three-argument constructor initialises the x, y and z data members as follows:

```

Point::Point (double x_arg, double y_arg, double z_arg)
: x (x_arg), y (y_arg), z(z_arg)
{}

```

Rather than initialising the data members by assignment within the constructor body, initialisation takes place in the constructor declarator. Note the use of the colon (:) after the constructor argument list and the use of the comma operator (,) between data members and the empty constructor body, since initialisation is now performed outside the constructor body. This data member initialisation is very neat, in that it uses the one-argument constructor of type **double** for each data member! For instance, in earlier chapters we have seen **int** variables defined and initialised as follows:

```
int i = 1 ;
```

when in fact we could initialise variable i using **int**'s one argument constructor:

```
int i (1) ;
```

The use of data member constructors with **class** constructors helps to emphasise the similarity between C++ integral and user-defined types and classes.

The general syntax is:

```

ClassName::ClassName (typeSpecifier arg1, ...
typeSpecifier argn)
: data_member1 (arg1), ..., data_membern (argn)
{
//...
}

```

### 9.13.6 Copy Constructor

The previous section illustrated a three-argument constructor for initialising the three data members of a Point object:

```

Point::Point (double x_arg, double y_arg, double z_arg)
: x (x_arg), y (y_arg), z(z_arg)
{}

```

In fact, for each of the three data members x, y and z a *copy constructor* has been called to perform data member initialisation. To help illustrate what a copy constructor actually is, consider the following program:

```
// copy_con.cpp
```

```

// illustrates the copy constructor
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    // public member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// constructors:

// no arg. constructor (default)
Point::Point ()
    : x(0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
Point::Point (double x_arg, double y_arg, double z_arg)
    : x (x_arg), y (y_arg), z(z_arg)
{ }
//...
void main ()
{
    // define objects p, q and r
    Point p (1.0, 2.0, 3.0) ; // define, 3 arg. con.
    Point q (p) ; // copy constructor
    Point r = p ; // copy initialised

    // O/P objects
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
        << "object q ("
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl
        << "object r ("
        << r.X () << ", " << r.Y () << ", "
        << r.Z () << ")" << endl ;
}

```

with output:

```
object p (1, 2, 3)
object q (1, 2, 3)
object r (1, 2, 3)
```

The following two statements, from program COPY\_CON.CPP, illustrate defining two objects q and r which are both initialised to object p:

```
Point q (p) ;           // copy constructor
Point r = p ;           // copy initialised
```

The first statement calls the C++ default copy constructor, which performs an exact copy of p's data members to q's respective data members, while the second statement performs a similar initialisation. Since copying and assigning one object to another are frequent requests, the compiler provides default routines to perform these tasks. The above two statements are slightly different from assignment:

```
Point s, t ;
//...
s = t ;
```

since *pure* assignment assumes that the objects s and t have previously been defined, whereas the copy constructor defines a new object as well as assigning one object's data members to another object.

Let's overloaded the default copy constructor for **class Point**:

```
// over_cc.cpp
// overloads Point's copy constructor
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    Point (const Point& p) ;
    // public member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point

// constructors:

// no arg. constructor (default)
Point::Point ()
: x(0.0), y (0.0), z (0.0)
```

```

    {}

// 3 arg. constructor
Point::Point (double x_arg, double y_arg, double z_arg)
    : x (x_arg), y (y_arg), z(z_arg)
{ }

// copy constructor
Point::Point (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    cout << "copy constructor called" << endl ;
}
//...
void main ()
{
    Point p (1.0, 2.0, 3.0) ; // define, 3 arg. con.
    cout << "object p (" 
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl ;

    Point q ; // define, no arg. con.
    cout << "object q (" 
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl ;

    Point r (p) ; // copy constructor
    cout << "object q (" 
        << r.X () << ", " << r.Y () << ", "
        << r.Z () << ")" << endl ;

    Point s = p ; // copy initialised
    cout << "object r (" 
        << s.X () << ", " << s.Y () << ", "
        << s.Z () << ")" << endl ;
}

```

which generates the output:

```

object p (1, 2, 3)
object q (0, 0, 0)
copy constructor called
object r (1, 2, 3)
copy constructor called
object s (1, 2, 3)

```

The output illustrates that the copy constructor is called when objects *r* and *s* are defined. The three-argument constructor is still called for object *p*, and the no-argument constructor for object *q*.

The copy constructor declarator is:

---

```
Point (const Point& p)
```

The object *p* is passed to the copy constructor by reference and **const** modified to ensure that *p* is not accidentally altered. Why wasn't the copy constructor declarator of the form:

```
Point (Point p) // error: X (X) not allowed
```

which passes object *p* by value? Such a copy constructor will produce a compilation error of the form '*Point (Point)* is not a valid copy constructor'. A copy constructor which passes an object by value is not allowed, because when the object is passed a copy of the object is made, which calls the copy constructor, which copies the object – and so on, until there is no memory left. Thus, pass by reference to a copy constructor!

The general syntax of the copy constructor is therefore:

```
ClassName (const ClassName& object)
{
//...
}
```

Copy constructors are also invoked when an object is passed by value to a function and when a temporary object is created for the return object of a function. To illustrate this, consider the program:

```
// copy_fun.cpp
// illustrates the copy constructor with
// function arguments and return values
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    Point (const Point& p) ;
    // public member functions
    //...
    void Function1 (Point p) ;
    Point Function2 () ;
}; // class Point
//...
// copy constructor
Point::Point (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    cout << "copy constructor called" << endl ;
}
```

```
//...
void Function1 (Point p)
{
    cout << "Function1 () called" << endl ;
}

Point Function2 ()
{
    cout << "Function2 () called" << endl ;
    Point temp ;
    return temp ;
}

void main ()
{
    Point p (1.0, 2.0, 3.0) ; // define, 3 arg. con.

    Function1 (p) ;
    Function2 () ;

    Point q ; // no arg. con.

    q = Function2 () ;
}
```

with output:

```
copy constructor called
Function1 () called
Function2 () called
copy constructor called
Function2 () called
copy constructor called
```

When *Function1()* is called a Point object is passed as an argument:

```
void Function1 (Point p)
{
    cout << "Function1 () called" << endl ;
}
```

Since the object is passed by value to *Function1()*, a copy is made. Copying the object invokes the copy constructor. If a Point object is passed by reference to *Function1()*:

```
void Function1 (Point& p)
{
    //...
}
```

the copy constructor is not called since the object is passed by reference.

Calling *Function2()* calls the copy constructor when the function returns a Point object:

```
Point Function2 ()
{
    cout << "Function2 () called" << endl ;
    Point temp ;
    return temp ;
}
```

When an object is returned by a function, a temporary object is created by the compiler to hold a copy of the object. This copying of an object to a temporary object invokes the copy constructor. The statement:

```
q = Function2 () ;
```

in the above program similarly invokes the copy constructor when a temporary object is returned by *Function2()*. However, assigning this object to object q will invoke the assignment operator and not the copy constructor.

The copy constructor is very useful for ensuring that an object allocates its own memory when it is created. This is necessary in certain circumstances, because when an object is returned from a function the object goes out of scope, thus causing the object's destructor to be called. If the copy constructor is not adequately defined, the object's destructor can delete memory allocated to an object in the calling function. We shall cover this topic when we discuss pointers.

### 9.13.7 Constructors, const and volatile

A **class** constructor can be called by a **const** or **volatile** object, but cannot be declared **const** or **volatile**:

```
// con_cv.cpp
// illustrates constructors, const and volatile

class X
{
public:
    X () {}
};

class Y
{
public:
    Y () const {}           // error: const constructor
    Y (int) volatile {}   // error: volatile constructor
};

void main ()
{
    const     X cx ;    // o.k.: const object
    volatile X vx ;    // o.k.: volatile object
}
```

## 9.14 Destructors

If a **class** constructor creates an object, then what destroys an object? In C++ the complement of the constructor is the *destructor*. The program DESTR.CPP illustrates a destructor for the Point **class**:

```
// destr.cpp
// illustrates destructors
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructor
    Point () ;
    // destructor
    ~Point () ;
    // public member functions
    //...
}; // class Point

// constructor
Point::Point ()
: x (0.0), y (0.0), z (0.0)
{
    cout << "constructor called" << endl ;
}

// destructor
Point::~Point ()
{
    cout << "destructor called" << endl ;
}
//...
void main ()
{
    // define object p
    Point p;

    // O/P object
    cout << "object p ("
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl ;
}
```

which produces the output:

```
constructor called  
object p (0, 0, 0)  
destructor called
```

The Point **class** destructor is declared within the **class** declaration as:

```
class Point  
{  
//...  
public:  
//...  
~Point () ;  
//...  
};
```

In general, a **class** and destructor declaration are of the following syntax:

```
class ClassName  
{  
private:  
//...  
public:  
// destructor  
~ClassName () ;  
//...  
};
```

The **class** destructor has the same name as the **class** name, and hence the constructor name, but is preceded by a tilde (~). A destructor does not return a type and has no function arguments. Destructors are not passed arguments, since it makes no sense to pass arguments to an object that is just about to be destroyed.

It was noted above that constructors generally have to be **public** so that they are accessible to objects of a **class**. At first you may think that it doesn't matter whether a destructor is **private** or **public** – well, it does. Just as a constructor does, a **class**'s destructor also needs to be **publicly** accessible to an object. The following program code has declared Point's destructor as **private**, and a compilation error will be generated of the form 'class Point's destructor is not accessible to object p':

```
class Point  
{  
private:  
//...  
// destructor  
~Point () ;  
//...  
public:  
//...  
};  
//...  
void main ()  
{
```

---

```
//...
Point p ; // error: destructor class Point not accessible
//...
}
```

Destructors are frequently used to deallocate (using the **delete** operator) memory which was dynamically allocated using the **new** operator. Thus, a more detailed discussion of destructors will be left until we discuss pointers and the **new** and **delete** operators in Chapter 12. For now, just bear in mind that a destructor destroys an object that was created by a constructor.

### 9.14.1 Destructors, **const** and **volatile**

As with constructors, a destructor can be invoked by a **const** or **volatile** object, but cannot be declared **const** or **volatile**:

```
// des_cv.cpp
// illustrates destructors, const and volatile

class X
{
public:
    ~X () {}
};

class Y
{
public:
    ~Y () const {}          // error: const destructor
    ~Y () volatile {}        // error: volatile destructor
};

void main ()
{
    const     X cx ; // o.k.: const object
    volatile X vx ; // o.k.: volatile object
}
```

## 9.15 **inline** Member Functions

Chapter 6 discussed **inline** functions with reference to *normal* functions. This section now examines **inline class** member functions. The following program illustrates **inline class** **Point** **X()**, **Y()** and **Z()** access member functions:

```
// inline.cpp
// illustrates inline member functions
#include <iostream.h> // C++ I/O

// class Point
```

```

class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    // public member functions
    double X () ;
    double Y () ;
    double Z () ;
}; // class Point
//...
// class Point member function definitions:

inline double Point::X ()
{ return x ; }

inline double Point::Y ()
{ return y ; }

inline double Point::Z ()
{ return z ; }

void main ()
{
//...
}

```

An **inline** member function is simply a function that is expanded inline at the point at which the member function is called. The general syntax of defining a function to be **inline** is:

```

inline return_type MemberFunction (parameter_list)
{
//...
}

```

which uses the **inline** keyword modifier before the member function return type. It was mentioned in Chapter 6 that inline functions are generally reserved for small functions that are frequently called. Thus, the access member functions *X()*, *Y()* and *Z()* are good candidates for making **inline**, since they simply return data members.

### 9.15.1 Automatic **inline** Member Functions

If we move the constructor and member function definitions from outside the **class** declaration in the above program, **INLINE.CPP**, and place them within **class** *Point*'s declaration, these functions are automatically made **inline**:

```
// auto_in.cpp
```

---

```
// illustrates automatic inline member functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // public member functions
    double X () { return x; }
    double Y () { return y; }
    double Z () { return z; }
}; // class Point

void main ()
{
//...
}
```

It was discussed in Chapter 6 that the use of the keyword **inline** is in fact a request, and the compiler ultimately decides whether the member function will be made **inline** or not. There are several circumstances in which the compiler may issue warning messages regarding **inline** member functions. For example, the following `Point::Divide()` member function tests whether an integer number, `n`, is not equal to zero and if so proceeds to return a `Point` object that is `Point` object `p1` divided by `n`:

```
inline Point Divide (const Point& p1, const int& n)
{
    if (n != 0)
        return Point (p1.x/n, p1.y/n, p1.z/n) ;
}
```

The Borland C++ (version 5.0) compiler generates a compilation warning message of the form 'A function containing a missing return statement is not expanded inline'. In addition, member functions containing **for**, **while** and **do-while**-loops, **switch** and **goto** statements, and **static** variables are not expanded **inline**.

## 9.15.2 **inline** Constructors

Constructors can also be defined as **inline** functions:

```
class Point
{
```

```

//...
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    //...
}; // class Point

// constructors:

// no arg. constructor (default)
inline Point::Point ()
    : x (0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
inline Point::Point (double x_arg, double y_arg,
                   double z_arg)
    : x (x_arg), y (y_arg), z (z_arg)
{ }

```

Since the two constructors for **class** Point simply initialise the three data members of **class** Point, it is feasible to make them **inline**.

## 9.16 const Member Functions

It was mentioned in Chapter 6 that a **class** member function declarator can include the keyword **const** modifier. This allows a member function access to a **class**'s data members, but prevents the member function from altering the value of a data member. The following program illustrates the use of **const** for **class** Point's member functions:

```

// con_mem.cpp
// illustrates const member functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg) ;
    // public member functions
    double X () const ;
    double Y () const ;
}

```

```

double Z () const ;
}; // class Point

// constructors:

// no arg. constructor (default)
inline Point::Point ()
: x (0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
inline Point::Point (double x_arg, double y_arg,
                   double z_arg)
: x (x_arg), y (y_arg), z (z_arg)
{ }

// class Point member function definitions:

inline double Point::X () const
{ return x ; }

inline double Point::Y () const
{ return y ; }

inline double Point::Z () const
{ return z ; }

void main ()
{
// define objects p and q
Point p (1.0, 2.0, 3.0) ;
Point q ;

// O/P objects
cout << "object p ("
    << p.X () << ", " << p.Y () << ", "
    << p.Z () << ")" << endl
    << "object q ("
    << q.X () << ", " << q.Y () << ", "
    << q.Z () << ")" << endl ;
}

```

The above program illustrates the use of the **const** keyword modifier in the member functions *X()*, *Y()* and *Z()* declarators:

```

inline double Point::X () const
{ return x ; }
//...

```

Note that the keyword **const** must appear in both the declaration and definition of a **const** member function. The use of **const** in a member function declarator indicates to the compiler

that the function is to have access to the **class**'s data members only, and should not alter the values of the data members. Thus, if the member function `Point::X()` attempted to alter the data member `Point::x` before returning, a compilation error would result:

```
inline double Point::X () const
{
    x = x + 1; // error: cannot modify a constant object
    return x;
}
```

You must be careful when operating on **const** member functions and **const** objects. A **const** member function can be called for both non-**const** and **const** objects, because the **const** member function declaration guarantees that the data members of the object that the member function operates on cannot be altered. However, a non-**const** member function cannot (or should not) be called for a **const** object because the non-**const** member function could alter the data members of a **const** object. For example:

```
// con_mem1.cpp
// further illustrates constant
// member functions
#include <iostream.h> // C++ I/O

class X
{
private:
    int data ;
public:
    X ()
        : data (0) {}
    int NonConstFunction ()
    { return data++ ; }
    int ConstFunction () const
    { return data ; }
};

void main ()
{
    X non_const_obj ;
    const X const_obj ;

    // non-const object
    non_const_obj.NonConstFunction () ; // o.k.
    non_const_obj.ConstFunction () ; // o.k.

    // const object
    const_obj.NonConstFunction () ; // warning/error
    const_obj.ConstFunction () ; // o.k.
}
```

The use of **const** member functions makes data member access watertight.

## 9.17 volatile Member Functions

Member functions can be declared **volatile** in a similar manner to **const** member functions:

```
// vol_mem.cpp
// illustrates volatile member functions
#include <iostream.h> // C++ I/O

class X
{
private:
    int data ;
public:
    X ()
        : data (0) {}
    void DataV () volatile
        { cout << data << endl ; }
    void DataCV () const volatile
        { cout << data << endl ; }
};

void main ()
{
    X x ;
    volatile X vx ; // volatile object

    x.DataV () ; x.DataCV () ; // warning/error
    vx.DataV () ; vx.DataCV () ;
}
```

**volatile** member functions can only be called for **volatile** objects. The above program also illustrates that **const volatile** member functions are also legal.

## 9.18 Default Member Function Arguments

Chapter 6 illustrated that function arguments can be assigned default values to enable a function call in which not all of the function arguments are passed to the function. Function arguments are assigned default values within the function declaration. To illustrate this for our **Point class**, let us assign a default value of zero to the **z** data member for the three-argument constructor declaration. This is particularly useful when applying the **Point class** to two-dimensional points (**x** and **y**) that lie in a plane **z=0**, such as a display screen. Program DEF\_ARG.CPP illustrates default member function arguments:

```
// def_arg.cpp
// illustrates default member function arguments
#include <iostream.h> // C++ I/O
```

```

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point () ;                                // default arg: z=0
    Point (double x_arg, double y_arg, double z_arg=0.0) ;
    // public member functions
    //...
}; // class Point

// constructors:

// no arg. constructor (default)
inline Point::Point ()
: x (0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
inline Point::Point (double x_arg, double y_arg,
                   double z_arg)
: x (x_arg), y (y_arg), z (z_arg)
{ }
//...
void main ()
{
    // define objects p and q, neglecting
    // z data member
    Point p (1.0, 2.0) ;
    Point q (3.0, 4.0) ;

    // O/P objects
    cout << "object p (" 
        << p.X () << ", " << p.Y () << ", "
        << p.Z () << ")" << endl
        << "object q (" 
        << q.X () << ", " << q.Y () << ", "
        << q.Z () << ")" << endl ;
}

```

Objects p and q are defined as:

```

Point p (1.0, 2.0) ;
Point q (3.0, 4.0) ;

```

The use of the default argument for the z data member in the declaration of the three-argument constructor of **class** Point relieves the user from having to pass a value of zero to the third argument of the constructor each time a two-dimensional Point object is created:

---

```
Point p (1.0, 2.0, 0.0) ;
Point q (3.0, 4.0, 0.0) ;
//...
```

It is worth noting that a default argument only appears in a constructor or member function declaration, and not in the definition.

What about default arguments for user-defined classes? Consider a Line **class** which encapsulates two Point objects:

```
class Line
{
private:
    Point p1, p2 ;
public:
    // constructors
    Line ()
        : p1 (), p2 () {}
    Line (Point p, Point q)
        : p1 (p), p2 (q) {}
    //...
};
```

If it is required that the default behaviour of the two argument constructor is to define a Line object which has one of its two Point objects at the origin of coordinates (0, 0, 0), then simply use Point's constructor in the default argument expression:

```
class Line
{
//...
public:
    //...
    Line (Point p, Point q=Point())
        : p1 (p), p2 (q) {}
    //...
};
```

The Line **class** is discussed in more detail later in the next chapter.

## 9.19 Functions with Object Arguments and Functions that Return Objects

When a user-defined **class** has been declared, there is no difference between passing objects of the user-defined **class** and passing identifiers or objects of a C++ integral type or **class** as arguments to a function. Similarly, returning objects of a user-defined **class** from a function follows the same rules as encountered in Chapter 6.

To illustrate passing object arguments to a function and returning an object from a function, let us extend our Point **class** by adding two member functions *Add()* and *Subtract()*, which perform the addition and subtraction of two objects of **class** Point:

```
// arg_ret.cpp
// illustrates passing object arguments to functions
// and returning an object from a function
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point () ; // default arg: z=0
    Point (double x_arg, double y_arg, double z_arg=0.0) ;
    // public member functions
    //...
    Point Add      (const Point& p) ;
    Point Subtract (const Point& p) ;
}; // class Point
//...
// p1 + p2
Point Point::Add (const Point& p)
{
    return Point (x+p.x, y+p.y, z+p.z) ;
}

// p1 - p2
Point Point::Subtract (const Point& p)
{
    return Point (x-p.x, y-p.y, z-p.z) ;
}

void main ()
{
    // define objects p1 and p2
    Point p1 (3.0, 4.0, 5.0) ;
    Point p2 (1.0, 2.0, 3.0) ;

    // O/P objects p1 & p2
    cout << "object p1 ("
        << p1.X () << ", " << p1.Y () << ", "
        << p1.Z () << ")" << endl
        << "object p2 ("
        << p2.X () << ", " << p2.Y () << ", "
        << p2.Z () << ")" << endl ;

    Point p3, p4 ;

    // add & subtract p1 & p2
    p3 = p1.Add (p2) ;
```

---

```

p4 = p1.Subtract (p2) ;

// O/P objects p3 & p4
cout << "object p3 (p1+p2): ("
    << p3.X () << ", " << p3.Y () << ", "
    << p3.Z () << ")" << endl
    << "object p4 (p1-p2): ("
    << p4.X () << ", " << p4.Y () << ", "
    << p4.Z () << ")" << endl ;
}

```

This program defines the *Add()* member function as:

```

Point Point::Add (const Point& p)
{
    return Point (x+p.x, y+p.y, z+p.z) ;
}

```

which enables two *Point* objects, *p1* and *p2*, to be added, and the result assigned to another object, *p3*, by a statement of the form:

```

//...
p3 = p1.Add (p2) ;

```

The *Add()* member function declarator indicates that the member function return type is **class Point**. Note that *Add()*'s function argument, *p*, is passed by reference, and **const** modified to ensure that *p*'s data members are not accidentally altered in the member function call. The statement:

```

return Point (x+p.x, y+p.y, z+p.z) ;

```

adds the respective *x*, *y* and *z* data members of objects *p1* and *p2* and returns a *new* object. This returned object is then assigned to the object *p3* in *main()*. Note that the object returned from *Add()* is created using the three-argument constructor. Alternatively, we could have defined *Add()* as:

```

Point Point::Add (const Point& p)
{
    Point p_temp ;
    p_temp.x = x + p.x ;
    p_temp.y = y + p.y ;
    p_temp.z = z + p.z ;

    return p_temp ;
}

```

The above two versions of *Add()* are equivalent, but the former is more concise.

Adding two *Point* objects *p1* and *p2* and assigning the result to object *p3* using the *Add()* function above:

```

//...

```

```
p3 = p1.Add (p2) ;
```

is rather messy. At present, the *Add()* member function is acting on the data members of object *p1* and returns the result. A recommended alternative design for *Add()* is to remove the **return** statement and let the member function act on the data members of the object calling *Add()*:

```
void Point::Add (const Point& p1, const Point& p2)
{
    x = p1.x+p2.x ; y = p1.y+p2.y ; z = p1.z+p2.z ;
}
```

Note that a **return** statement is not required and two arguments are now passed to *Add()*. Thus, to add two Points *p1* and *p2* we now write:

```
//...
p3.Add (p1, p2) ;
```

which no longer requires the use of the assignment operator to assign *p1+p2* to *p3*.

This approach is illustrated in the program ARG\_RET2.CPP:

```
// arg_ret2.cpp
// illustrates passing object arguments to functions
// and assigning values to an object's data members
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    //...
    void Add      (const Point& p1, const Point& p2) ;
    void Subtract (const Point& p1, const Point& p2) ;
}; // class Point
//...
// p1 + p2
inline void Point::Add (const Point& p1, const Point& p2)
{
    x = p1.x+p2.x ; y = p1.y+p2.y ; z = p1.z+p2.z ;
}

// p1 - p2
inline void Point::Subtract (const Point& p1, const Point& p2)
{
    x = p1.x-p2.x ; y = p1.y-p2.y ; z = p1.z-p2.z ;
}

void main ()
```

```

{
// define objects p1 and p2
Point p1 (3.0, 4.0, 5.0) ;
Point p2 (1.0, 2.0, 3.0) ;

// O/P objects p1 & p2
cout << "object p1 (" 
    << p1.X () << ", " << p1.Y () << ", "
    << p1.Z () << ")" << endl
    << "object p2 (" 
    << p2.X () << ", " << p2.Y () << ", "
    << p2.Z () << ")" << endl ;

Point p3, p4 ;

// add & subtract p1 & p2
p3.Add (p1, p2) ;
p4.Subtract (p1, p2) ;

// O/P objects p3 & p4
cout << "object p3 (p1+p2): (" 
    << p3.X () << ", " << p3.Y () << ", "
    << p3.Z () << ")" << endl
    << "object p4 (p1-p2): (" 
    << p4.X () << ", " << p4.Y () << ", "
    << p4.Z () << ")" << endl ;
}

```

Note that the *Add()* and *Subtract()* member functions are now **inline** functions, since they are small functions.

### 9.19.1 Returning by Reference

The member functions *X()*, *Y()* and *Z()* at present only allow us to access the **private** data members of **class Point**. Once an object of **class Point** has been defined, it is not possible to alter the *x*, *y* and *z* data members of the object using the *X()*, *Y()* and *Z()* member functions. However, if we make the return type of each member function a reference to a **double** (rather than simply a **double**), then the return value of each function will refer to its respective data member. This is demonstrated in the following program:

```

// ret_ref.cpp
// illustrates returning by reference for both
// constant and non-constant objects
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members

```

```
    double x, y, z ;
public:
//...
// public member functions
double& X () ;
double& Y () ;
double& Z () ;
const double& X () const ;
const double& Y () const ;
const double& Z () const ;
//...
}; // class Point
//...
// class Point member function definitions:

// non-const objects
inline double& Point::X ()
{ return x ; }

inline double& Point::Y ()
{ return y ; }

inline double& Point::Z ()
{ return z ; }

// const objects
inline const double& Point::X () const
{ return x ; }

inline const double& Point::Y () const
{ return y ; }

inline const double& Point::Z () const
{ return z ; }

//...
void main ()
{
// define objects p1 and p2
Point p (1.0, 2.0) ;

// O/P object p
cout << "object p ("
    << p.X () << ", " << p.Y () << ", "
    << p.Z () << ")" << endl ;

p.X () = 3.0 ;
p.Y () = 4.0 ;

// O/P object p
cout << "object p ("
    << p.X () << ", " << p.Y () << ", "
```

---

```
<< p.Z () << " ) " << endl ;  
}
```

Two sets of the `X()`, `Y()` and `Z()` member functions are given, one for non-constant objects and one for constant objects.

## 9.20 Combining Constructor and Member Function Calls

The previous section demonstrated that an object can be returned from a function without explicitly defining the object:

```
Point Point::Add (const Point& p)  
{  
    return Point (x+p.x, y+p.y, z+p.z) ;  
}
```

The compiler generates a temporary object which is returned from the function. Similarly, C++ allows a member function of a given **class** to be called without explicitly defining an object. In general:

```
ClassConstructor ().MemberFunction () ;
```

The following program illustrates such a member function call for **class** `Point`:

```
// no_obj.cpp  
// illustrates a constructor and member function  
// call in a single statement  
#include <iostream.h> // C++ I/O  
  
// class Point  
class Point  
{  
    private:  
        // private data members  
        double x, y, z ;  
    public:  
        // constructors  
        Point () ; // default arg: z=0  
        Point (double x_arg, double y_arg, double z_arg=0.0) ;  
        // public member functions  
        double& X () ;  
        //...  
}; // class Point  
//...  
void main ()  
{  
    double d = Point (1.0, 2.0, 3.0).X () ;
```

```
cout << "double d: " << d << endl ;  
}
```

The statement:

```
double d = Point (1.0, 2.0, 3.0).X () ;
```

illustrates the combined use of Point's constructor and the access member function Point::X().

This concise notation may appear a little strange at first, but it can be useful in certain circumstances. A popular application of the above syntax is in executing dialog boxes in a Windows environment:

```
// application class  
class MyApplication: public TApplication  
{  
    private:  
        TWindow* Client ;  
        //...  
    protected:  
        void CmHelpAbout () ;  
    public:  
        MyApplication () ;  
        virtual ~MyApplication () ;  
        //...  
        virtual void InitMainWindow () ;  
        //...  
};  
//...  
// About dialog class  
class AboutDialog: public TDDialog  
{  
    //...  
};  
//...  
// initialise main window of application  
void MyApplication::InitMainWindow ()  
{  
    Client = new TWindow /*...*/ ;  
    TFrameWindow* frame = new TFrameWindow /*...*/ ;  
    //...  
    MainWindow = frame ;  
    //...  
}  
//...  
// executes Help/About dialog box  
void MyApplication::CmHelpAbout ()  
{  
    AboutDialog (MainWindow).Execute () ;  
}  
//...
```

The above program code incorporates the two **class** declarations MyApplication and AboutDialog. The **class** MyApplication is derived from the Borland *ObjectWindows TApplication class*. A TApplication-derived object encapsulates a Windows application. The **class** AboutDialog is derived from the Borland ObjectWindows **TDialog class**. An AboutDialog-derived object and an about-dialog box resource enable a dialog box to pop up which displays program information. The function *CmHelpAbout ()* associates the pull-down menu option Help | About with the about-dialog box. The *CmHelpAbout ()* function demonstrates the combined use of calling the one-argument constructor of **class** AboutDialog to create a temporary object, and in the same statement calling the *Execute ()* function to display the about-dialog box.

## 9.21 Arrays of Objects

The following program defines an array of three objects of **class** Point:

```
// array_ob.cpp
// illustrates an array of objects
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point () ; // default arg: z=0
    Point (double x_arg, double y_arg, double z_arg=0.0) ;
    // public member functions
    //...
}; // class Point

// constructors:

// no arg. constructor (default)
inline Point::Point ()
    : x (0.0), y (0.0), z (0.0)
{ }

// 3 arg. constructor
inline Point::Point (double x_arg, double y_arg,
                  double z_arg)
    : x (x_arg), y (y_arg), z (z_arg)
{ }
//...
void main ()
{
    // define an array of 3 Point objects
```

```

Point tri_vertices[3] ;

// initialise array objects
tri_vertices[0] = Point (1.0, 1.0) ;
tri_vertices[1] = Point (4.0, 2.0) ;
tri_vertices[2] = Point (3.0, 5.0) ;

// O/P objects
for (int i=0; i<3; i++)
{
    cout << "tri_vertices[" << i << "]:" <<
        "(" << tri_vertices[i].X () <<
        ", " << tri_vertices[i].Y () <<
        ", " << tri_vertices[i].Z () <<
        ")" << endl ;
}
}

```

In the `main()` function an array `tri_vertices` of three elements of **class** `Point` is defined:

```
Point tri_vertices[3] ;
```

This definition is identical in format to an array of elements of a C++ integral type:

```
int i_array[3] ;
```

The operations on an array whose elements are of a user-defined **class** are identical to those of a fundamental or integral C++ type.

Similarly, arrays can be used as **class** data members. `Point`'s `x`, `y` and `z` data members could have been represented as an array with three elements:

```

class Point
{
private:
    double pt[3] ;
public:
    Point () { pt[0]=0.0 ; pt[1]=0.0 ; pt[2]=0.0 ; }
    //...
} ;

```

The `x`, `y` and `z` data members of `Point` are now represented as `pt[0]`, `pt[1]` and `pt[2]`, respectively.

## 9.22 Local Classes

The C++ programming language allows a **class** to be declared local to a function:

```
// local.cpp
```

```

// illustrates local classes
#include <iostream.h> // C++ I/O

void Function (int i)
{
    class X
    {
        private:
            int data ;
        public:
            //X (int i) ;
            X (int i)
                : data (i) {}
            void Set (int i) { data = i ; }
            int Get () { return data ; }
    }; // local class X

    //X::X (int i) // error: non-inline function of local
                    // class
    // : data (i) {}

    X x (i) ; // function argument i
    cout << x.Get () << endl ;

    int j (11) ; // local function variable j
    x.Set (j) ;
    cout << x.Get () << endl ;
}

void main ()
{
    Function (10) ;
}

```

with output:

```

10
11

```

The above program illustrates a **class** X declared within the scope of a function. LOCAL.CPP illustrates that objects of a local **class** can access function arguments and local function variables, but non-inline functions of a local **class** are illegal. Local classes are rarely used in practice, but nevertheless exist if the need arises!

## 9.23 Nested Classes

Before we summarise the syntax of a **class** declaration in the next section, let us briefly discuss nested classes. Consider the following, somewhat impractical, **class** declaration of Point:

```
// nested.cpp
// illustrates nested classes
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // nested class
    class DataMembers
    {
public:
    double x, y, z;
    }dm;
public:
    // constructors
    Point () ;
    //...
    // public member functions
    double& X () ;
    //...
}; // class Point

// constructors:

// no arg. constructor (default)
inline Point::Point ()
{
    dm.x = dm.y = dm.z = 0.0 ;
}
//...
// class Point member function definitions:

inline double& Point::X ()
{
    return dm.x ;
}
//...
void main ()
{
    Point p (1.0, 2.0) ;

    cout << "p: " << "(" << p.X () << ", " << p.Y () 
                    << ", " << p.Z () << ")" << endl ;
}
```

This program illustrates a **class** called **DataMembers** declared within the declaration of **class Point**. **DataMembers** is declared a **private** nested **class** of **class Point** and is therefore inaccessible to objects of **class Point**. The **x**, **y** and **z** data members of **class DataMembers** are declared **public** and are therefore accessible to **class Point**. An object, **dm**, of **class DataMembers** is declared as a **private** data member of **class Point**, and the program illustrates that access to the **x**, **y** and **z** data members is now via the **dm** object.

As a further illustration of nested classes, consider the following program:

```
// nested1.cpp
// further illustrates nested classes
#include <iostream.h> // C++ I/O

class A // class A
{
public:
    int adata ;
    A () ;
    class B // nested class B
    {
public:
        int bdata ;
        B () ;
        class C // nested-nested class C
        {
public:
            int cdata ;
            C () ;
        }; // C
    }; // B
}; // A

// class A constructor
A::A ()
{ adata = 0 ; }

// nested class B constructor
A::B::B ()
{ bdata = 1 ; }

// nested-nested class C constructor
A::B::C::C ()
{ cdata = 2 ; }

void main ()
{
    A a ;
    A::B b ;
    A::B::C c ;

    cout << "A::adata: " << a.adata << endl ;
    cout << "B::bdata: " << b.bdata << endl ;
    cout << "C::cdata: " << c.cdata << endl ;

    // without explicit qualification
    B b_noqual ;
    C c_noqual ;
}
```

I am not suggesting that you adopt this programming style of nesting classes to the *n*th degree, but it does, more importantly, illustrate explicit qualification via the scope resolution operator for nested classes. The `main()` function above further illustrates that when a nested `class` is declared and there is no other `class` of the same name declared in the scope of the program, explicit qualification to the nested `class` is not essential, although most compilers will issue a compilation warning asking you to use explicit qualification.

## 9.24 Another `class` Declaration

A more comprehensive `class` declaration syntax which includes `class` constructors, de-structor and member functions is of the form:

```
class ClassName
{
    // private data members
    typeSpecifier pri_member_name ;
    //...
    typeSpecifier PrivateMemberFunction (parameter_list) ;
    //...
private:
    // private data members
    typeSpecifier pri_member_name ;
    //...
    // private member functions
    typeSpecifier PrivateMemberFunction (parameter_list) ;
    //...
public:
    // public data members
    typeSpecifier pub_member_name ;
    //...
    // constructors
    ClassName () ;
    ClassName (parameter_list) ;
    //...
    // destructor
    ~ClassName () ;
    // public member functions
    typeSpecifier PublicMemberFunction () ;
    //...
    //...
} object_list ;
```

## 9.25 `struct` and `class`

We have now seen `struct` and `class` and it is instructive to remind ourselves of the similarities and differences between them. The data members of a `struct` are `public` by

default, whereas **class** data members are **private** to the **class** by default. Access can be changed, for both **struct** and **class**, by the access specifiers **private** and **public**. Like the **class**, C++ **structs** may have member functions, but in C++ programming the **struct** is generally reserved for data encapsulation, whereas the **class** is used for user-defined data types which encapsulate both data and member functions. In fact, the **struct** has the same scope and inheritance (Chapter 15) rules as the **class**, and differs only in the default access rights. There are instances in which the use of a **struct** is slightly more convenient than a **class** because of its default **public** access rights and in emphasising that a structure is a simple aggregation of data and not a concrete data type. For a discussion of the importance of maintaining a similarity between **struct** and **class** in the evolution of C++, refer to Stroustrup (1994, Section 3.5.1).

## 9.26 Intersection of Two Two-Dimensional Line Segments

When functions were covered in Chapter 6, we concluded the chapter with a discussion of the intersection of two two-dimensional line segments, and promised to return to the topic of line segment intersection in Chapter 9 – well, here we are. The following program implements the clockwise/anticlockwise and intersection functions, *CAC()* and *LineIntersection()*, of Chapter 6, but now from a **class** perspective:

```
// li_class.cpp
// Point and Line classes with Point clockwise/anticlockwise
// and Line intersection routines

#include <iostream.h> // C++ I/O

const double TOLERANCE = 1e-06 ;

// Boolean type
enum Boolean { FALSE, TRUE };

// class Point
class Point
{
private:
    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    // default arg: z=0
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    double& X () { return x ; }
    double& Y () { return y ; }
```

```

double& Z () { return z ; }
const double& X () const { return x ; }
const double& Y () const { return y ; }
const double& Z () const { return z ; }
void Display (char msg[]);
int CAC (const Point& p1, const Point& p2) ;
} // class Point

// class Point member functions:

// passed 2 Points in 2D plane
// returns -1 1 or 0 depending whether points are clockwise,
// anti-clockwise, collinear, ...
int Point::CAC (const Point& p1, const Point& p2)
{
    int cac = 0 ; // return var. indicating direction

    // use point 0 as 'local' origin
    double dx1 = p1.X() - x ; double dy1 = p1.Y() - y ;
    double dx2 = p2.X() - x ; double dy2 = p2.Y() - y ;

    // eliminate infinite gradients by multiplying gradients
    // between points 0-1 and 0-2 by (dx1*dx2)

    if (dx1*dy2 > dx2*dy1) // anti-clockwise (m02>m01)
        cac = 1 ;

    if (dx1*dy2 < dx2*dy1) // clockwise (m02<m01)
        cac = -1 ;

    if ((dx1*dy2 - dx2*dy1) < TOLERANCE) // same gradients
    {
        if ((dx1*dx2 < 0) || (dy1*dy2 < 0))
            cac = -1 ;
        else if ((dx1*dx1 + dy1*dy1) >= (dx2*dx2 + dy2*dy2))
            cac = 0 ;
        else
            cac = 1 ;
    }
    return cac ;
} // Point::CAC()

void Point::Display (char msg[])
{
    cout << msg << ":" <<
        x << ", " << y << ", " << z << ")" << endl ;
} // Point::Display()

// class Line
class Line
{

```

```

private:
    Point p1, p2 ;
public:
    // constructors
    Line ()
        : p1 (), p2 () {}
    Line (Point p, Point q)
        : p1 (p), p2 (q) {}
    // copy constructor
    Line (const Line& l)
        : p1 (l.p1), p2 (l.p2) {}
    // member functions
    Point& P1 () { return p1 ; }
    Point& P2 () { return p2 ; }
    const Point& P1 () const { return p1 ; }
    const Point& P2 () const { return p2 ; }
    void Display (char msg[]) ;
    Boolean Intersection (const Line& l) ;
} // class Line

// class Line member functions:

// intersection of 2 Lines
// returns FALSE if no intersection, TRUE if intersection
Boolean Line::Intersection (const Line& l)
{
    // intersection
    if ( ((p1.CAC (p2, l.p1) *
            p1.CAC (p2, l.p2)) <= 0) &&
        ((l.p1.CAC (l.p2, p1) *
            l.p1.CAC (l.p2, p2)) <= 0) )
        return TRUE ;
    // no intersection
    else
        return FALSE ;
} // Line::Intersection()

void Line::Display (char msg[])
{
    cout << msg << ":" "
        << "p1: (" << p1.X() << ", " << p1.Y()
           << ", " << p1.Z() << ") , "
        << "p2: (" << p2.X() << ", " << p2.Y()
           << ", " << p2.Z() << ")" << endl ;
} // Line::Display()

void main ()
{
    // define Points & Lines
    Point i (1, 1), j (4, 2), k (3, 5), l (3, 0) ;
    Line l1 (i, j), l2 (k, l) ;
}

```

```

// O/P Point's
i.Display ("i") ; j.Display ("j") ;
    k.Display ("k") ; l.Display ("l") ;
// O/P Line's
l1.Display ("l1") ;
l2.Display ("l2") ;

if (l1.Intersection (l2))
    cout << "lines intersect" << endl ;
else
    cout << "lines do not intersect" << endl ;
} // main()

```

The function *CAC()* is a member function of *Point* and *Intersection()* is a member of *Line*. Apart from the member functions *CAC()* and *Intersection()* implicitly acting on the *Point* and *Line* objects that call the member functions, the functions are identical to those outlined in Chapter 6. The *Line class* declaration is:

```

class Line
{
private:
    Point p1, p2 ;
public:
    // constructors
    Line ()
        : p1 (), p2 () {}
    Line (Point p, Point q)
        : p1 (p), p2 (q) {}
    // copy constructor
    Line (const Line& l)
        : p1 (l.p1), p2 (l.p2) {}
    // member functions
    Point& P1 () { return p1 ; }
    Point& P2 () { return p2 ; }
    const Point& P1 () const { return p1 ; }
    const Point& P2 () const { return p2 ; }
    void Display (char mag[]) ;
    Boolean Intersection (const Line& l) ;
}; // class Line

```

The declaration indicates that *Line* encapsulates two *Point* data members, which represent the end-points of a line segment. The zero- and two-argument constructors initialise the *Point* data members of *Line*, while the one-argument copy constructor performs an exact copy of a *Line* object. The member functions *P1()* and *P2()* simply return a reference to the *Point* data members for both non-constant and constant *Point* objects, *Display()* displays a *Line* object and *Intersection()* tests whether two *Line* line segment objects intersect. The two-argument *Line* constructor allows an object of **class** *Line* to be defined by passing two *Point* objects:

```

Point i (1, 1), j (4, 2) ;
//...

```

---

```
Line 11 (i, j) ;
```

Two Line objects can now easily be tested to see whether they intersect or not:

```
Boolean bool = l1.Intersection (l2) ;
```

A comparison of the above program, LI\_CLASS.CPP, with the program LI.CPP of Chapter 6 clearly illustrates the benefits of adopting an object-oriented programming approach, particularly for a user of a line segment intersection routine. The Point and Line classes offer the user a more natural representation of geometric points and lines and clearly associate the clockwise/anticlockwise function *CAC()* with points and the line segment intersection function *Intersection()* with lines.

## 9.27 Summary

The **class** is the central feature of object-oriented programming in C++. A **class** encapsulates both data and functions that operate on the data. The C++ **class** enables the programmer to develop new data types and model more closely objects in the real world than a procedural style of programming.

This chapter has focused on a **class** called Point which models a point in three-dimensional space. A quick glance through this chapter will illustrate how the Point **class** has grown and developed into a powerful **class**. The final declaration of **class** Point developed in this chapter is:

```
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point () ;
    Point (double x_arg, double y_arg, double z_arg=0.0) ;
    // public member functions
    double& X () { return x ; }
    double& Y () { return y ; }
    double& Z () { return z ; }
    const double& X () const { return x ; }
    const double& Y () const { return y ; }
    const double& Z () const { return z ; }
    void Add      (const Point& p1, const Point& p2) ;
    void Subtract (const Point& p1, const Point& p2) ;
};
```

The Point **class** declaration above is just 16 lines long (discounting comment lines) and yet possesses the essential characteristics of the majority of classes that programmers develop. A **class** declaration begins with the keyword **class**, followed by the name of the **class**. The body of the **class** is within opening and closing braces and the declaration is completed with a semicolon. Class data members can be either **private** or **public**. By default, data

members of a **class** are **private**, and if **public** data members are required they must be explicitly indicated as such. The keywords **private** and **public** give the developer of a **class** complete control over the access of a **class**'s data members.

Classes have *special* member functions called *constructors* and a single *destructor*. Both constructors and the destructor have the same name as the **class** name, but the destructor is preceded by a tilde. Constructors initialise an object's data members when the object is created and are called automatically. Default, no-argument, constructors can be defined which enable a user to define an object of a given **class** without passing any arguments to the constructor and still be guaranteed of the values of the object's data members. The destructor is automatically called when an object is destroyed.

A **class** can have member functions which operate specifically on the **class**'s data members. Member functions are not *normal* or *global* functions, but are specific to a given **class**. A member function is frequently referred to as a *message*, and it can be helpful to think of member functions as sending messages to an object. Member functions are frequently used as *access* functions to access an object's data members and to ensure that an object's data members are not accidentally altered. A **class**'s member functions can also be made **inline** to eliminate calling the function for small functions. The constructors and member functions of **class** Point are:

```
// constructors:  
  
inline Point::Point ()  
: x (0.0), y (0.0), z (0.0)  
{}  
  
inline Point::Point (double x_arg, double y_arg, double z_arg)  
: x (x_arg), y (y_arg), z (z_arg)  
{}  
  
// class Point member function definitions:  
  
inline double& Point::X ()  
{ return x ; }  
  
inline double& Point::Y ()  
{ return y ; }  
  
inline double& Point::Z ()  
{ return z ; }  
  
inline const double& Point::X () const  
{ return x ; }  
  
inline const double& Point::Y () const  
{ return y ; }  
  
inline const double& Point::Z () const  
{ return z ; }  
  
inline void Point::Add (const Point& p1, const Point& p2)  
{
```

---

```

x = p1.x+p2.x ; y = p1.y+p2.y ; z = p1.z+p2.z ;
}

inline void Point::Subtract (const Point& p1, const Point& p2)
{
x = p1.x-p2.x ; y = p1.y-p2.y ; z = p1.z-p2.z ;
}

```

The above illustrates that a **class**'s constructors and member functions can be defined outside a **class** declaration. Constructors and member functions can be passed an object of a user-defined **class** just as if it were an object of a C++ integral type or **class**. In addition, member functions can return an object of a user-defined **class**, illustrating that correctly implemented user-defined classes are equivalent to C++ integral types and classes.

Later chapters will extend the **Point class** even further by making the application and manipulation of **Point** objects more natural, particularly in the next chapter, which introduces the topic of operator overloading which allows a programmer to define a set of operators that act on objects of a user-defined type.

## Exercises

- 9.1 Develop a **Window class** which models a two-dimensional rectangular window that is characterised by left, right, top and bottom integer values. Provide member functions for **Window** which return the width, height and origin of a **Window** object. It is required that the default constructor of **class Window** places a **Window** object at the origin and has width and height of 120 and 80 respectively.
- 9.2 Develop an **Int class** which mimics the C++ **int** data type. A default constructor of **Int** should initialise an object of **class Int** to zero. Provide display, increment, decrement and access member functions for **Int**.
- 9.3 Make the following **Node class** a **private** nested **class** of **LinkedList**:

```

class Node
{
public:
    int value ;
    Node* next ;
    Node (int v, Node* next_ptr)
        : value (v), next (next_ptr) {}
};

class LinkedList
{
private:
    Node* first ;
public:
    //...
};

```

- 9.4 Modify the following **class** so as to make use of the **inline** and **const** features of **class** member functions:

```

class Apple
{
private:
    Position      pos ;
    Size          siz ;
    SurfaceTexture tex ;

public:
    // constructors
    Apple () ;
    Apple (const Position& p, const Size& s,
            const SurfaceTexture& st) ;
    // member functions
    Position      Location () ;
    Size          Dimensions () ;
    SurfaceTexture Texture () ;
};

Apple::Apple ()
: pos (), siz (), tex ()
{ }

Apple::Apple (const Position& p, const Size& s,
              const SurfaceTexture& st)
: pos (p), siz (s), tex (st)
{ }

Position Apple::Location ()
{
    return pos ;
}

Size Apple::Dimensions ()
{
    return siz ;
}

SurfaceTexture Apple::Texture ()
{
    return tex ;
}

```

- 9.5 Rewrite the Point, Line and Triangle structures developed for Exercise 8.5 so that they are now classes. The perimeter, centroid and area functions should now be member functions of **class** Triangle. Test your program on a triangle whose three vertices are (1,1,0), (3,2,0) and (2.5,4,0). Compare the *main()* functions of Exercises 8.5 and 9.5.
- 9.6 Consider the development of a Triangle **class** which encapsulates three Point objects. Since several Triangle objects can share the same Point it is advantageous for Triangle to contain references to its Point objects rather than the objects themselves.
- 9.7 Develop a **class**, CartCoorSys, to represent a three-dimensional Cartesian coordinate system which is defined in terms of an origin and three coordinate axes. The origin and axes are to be modelled as three-dimensional vectors. The default constructor of **class**

CartCoorSys is to describe a coordinate system positioned at the origin, (0,0,0), with the unit vector axes (1,0,0), (0,1,0) and (0,0,1). Use your CartCoorSys **class** to declare a CartPoint **class** which now encapsulates a CartCoorSys data member as well as the **double** x,y and z data members. Consider the design of classes for cylindrical, polar and spherical coordinate systems.

# Operators and Overloading

01:10  
01/11

C++ allows operators to be overloaded specifically for a user-defined class. Such a feature offers a programmer a more natural and concise syntax for performing operations on objects of a given class. For instance, consider a *Matrix* class which encapsulates an array of values or objects in to a single entity. If we are required to perform an addition of two *Matrix* objects, *m<sub>1</sub>* and *m<sub>2</sub>*, then one solution would be for class *Matrix* to support an *Add()* member function which adds the respective data members of *Matrix* objects *m<sub>1</sub>* and *m<sub>2</sub>*:

```
Matrix m3 ;  
//...  
m3.Add (m1, m2) ;
```

Such an approach works fine, but operator overloading offers a programmer a more elegant and powerful mathematical shorthand for performing operations on *Matrix* objects:

```
Matrix m3 = m1 + m2 ;
```

Overloading the addition operator, +, specifically for class *Matrix* allows a user of the *Matrix* class to write more abstract and at the same time more natural statements.



## 10.1 Overloading the Arithmetic Binary Operators

We saw in the last chapter that the *Point* **class** contained two member functions, *Add()* and *Subtract()*, for performing the addition and subtraction of two *Point* objects:

```
//...  
// p1 + p2  
inline void Point::Add (const Point& p1, const Point& p2)  
{
```

---

```

x = p1.x+p2.x ; y = p1.y+p2.y ; z = p1.z+p2.z ;
}

// p1 - p2
inline void Point::Subtract (const Point& p1, const Point& p2)
{
x = p1.x-p2.x ; y = p1.y-p2.y ; z = p1.z-p2.z ;
}

void main ()
{
//...
p3.Add (p1, p2) ;
p4.Subtract (p1, p2) ;
//...
}

```

The `Point::Add()` member function is passed two objects, adds together their respective `x`, `y` and `z` data members, and assigns the results to the calling object's data members.

If we were performing the addition of two numbers of the integral type `int`, we could simply write:

```

int i1, i2, i3 = 0;
//...
i3 = i1 + i2 ;

```

which is clearly more intuitive than our present addition of two objects of `class Point`.

In C++, it is possible to write expressions of the form:

```

Point p1, p2, p3 ;
//...
p3 = p1 + p2 ;

```

To perform such natural mathematical expressions with objects of `class Point` requires the use of a C++ feature called *operator overloading*. Before we examine operator overloading in detail, let's take a look at a program which attempts to add two `Point` objects using the addition, `+`, operator. The following program is basically identical to the ARG\_RET2.CPP program of Chapter 9:

```

// add.cpp
// attempts to add two Point objects using
// the addition operator + before overloading
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:

```

```

// constructors
Point ()
    : x (0.0), y (0.0), z (0.0) {}
Point (double x_arg, double y_arg, double z_arg=0.0)
    : x (x_arg), y (y_arg), z (z_arg) {}
// public member functions
double& X () { return x ; }
double& Y () { return y ; }
double& Z () { return z ; }
const double& X () const { return x ; }
const double& Y () const { return y ; }
const double& Z () const { return z ; }
void Add      (const Point& p1, const Point& p2) ;
void Subtract (const Point& p1, const Point& p2) ;
} // class Point

// class Point member function
// definitions:

// p1 + p2
inline void Point::Add (const Point& p1, const Point& p2)
{
    x = p1.x+p2.x ; y = p1.y+p2.y ; z = p1.z+p2.z ;
}

// p1 - p2
inline void Point::Subtract (const Point& p1,
                           const Point& p2)
{
    x = p1.x-p2.x ; y = p1.y-p2.y ; z = p1.z-p2.z ;
}

void main ()
{
    // define objects
    Point p1 (3.0, 4.0, 5.0) ;
    Point p2 (1.0, 2.0, 3.0) ;
    Point p3, p4 ;

    // add p1 & p2
    p3 = p1 + p2 ; // error: illegal operation
}

```

The statement:

```
p3 = p1 + p2 ;
```

will generate a compilation error of the form ‘illegal operation on objects’ because we have at present not informed the compiler of how to add objects of **class** Point. Thus, we need to instruct the compiler how to add two objects of **class** Point. To do this, we require the **operator** keyword.

### 10.1.1 The **operator** Keyword

C++ provides a keyword called **operator** which overloads a given operator specifically for a user-defined **class**. Let's overload the addition and subtraction arithmetic operators for **class Point**:

```
// over_as.cpp
// overloads the arithmetic (+) and (-) operators for class
Point
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // public member functions
    double& X () { return x; }
    double& Y () { return y; }
    double& Z () { return z; }
    const double& X () const { return x; }
    const double& Y () const { return y; }
    const double& Z () const { return z; }
    // overloaded operators
    Point operator + (const Point& p) ;
    Point operator - (const Point& p) ;
}; // class Point

// overloaded operators:

// p1 + p2
inline Point Point::operator + (const Point& p)
{
    return Point (x+p.x, y+p.y, z+p.z) ;
}

// p1 - p2
inline Point Point::operator - (const Point& p)
{
    return Point (x-p.x, y-p.y, z-p.z) ;
}

void main ()
{
```

```

// define objects
Point p1 (3.0, 4.0, 5.0) ;
Point p2 (1.0, 2.0, 3.0) ;
Point p3, p4, p5 ;

// add p1 & p2
p3 = p1 + p2 ; // addition
p4 = p1 - p2 ; // subtraction
p5 = p1 - p2 + p3 - p4 ; // chained addition & subtraction

// O/P
cout << "p3 (" << p3.X () << ", " << p3.Y ()
    << ", " << p3.Z () << ")" << endl ;
cout << "p4 (" << p4.X () << ", " << p4.Y ()
    << ", " << p4.Z () << ")" << endl ;
cout << "p5 (" << p5.X () << ", " << p5.Y ()
    << ", " << p5.Z () << ")" << endl ;
}

```

with output:

```

p3 (4, 6, 8)
p4 (2, 2, 2)
p5 (4, 6, 8)

```

The definition of the overloaded addition operator for **class Point** is:

```

inline Point Point::operator + (const Point& p)
{
    return Point (x+p.x, y+p.y, z+p.z) ;
}

```

The declarator is similar to member function declarators except for the keyword **operator**, which precedes the operator (in this case +). The keyword **operator** instructs the compiler that an operator is being overloaded. Note the use of **const** and **inline**.

The general syntax of an out-of-line overloaded operator member function is:

```

typeSpecifier ClassName::operator op (parameterList)
{
    // ...
}

```

where **op** is the overloaded operator, e.g. **<**, **+**, **=** or **[ ]**. At first, overloaded operator member functions can appear a little confusing, mainly because only a single member function argument is used for binary operators which operate on two operands:

```

p3 = p1 + p2 ;

```

The object **p1**, on the left-hand side of the **+** operator is an object of **class Point** to which the overloaded operator is a member function and is thus implicitly passed. The object **p2**, on the right-hand side of the **+** operator is also an object of **class Point**, but is passed as a

member function argument and thus explicitly passed. This is reflected in the return statement of the overloaded + operator:

```
return Point (x+p.x, y+p.y, z+p.z) ;
```

which illustrates that the left-hand object's data members are directly accessed whereas the right-hand object's data members are accessed via the object function argument.

Returning a Point object in the overloaded + and - operator member functions allows a user to *chain* together operators to form complicated expressions. In the program above, OVER\_AS.CPP, an example of chaining the + and - operators is given:

```
p5 = p1 - p2 + p3 - p4 ;
```

If the overloaded operator member functions did not return a Point object such expressions would not be possible.

Note that the previous functions *Add()* and *Subtract()* have been removed from the **class Point** declaration since they are now redundant.

Overloading the multiplication (\*), division (/) and arithmetic assignment (+=, -=, \*= and /=) operators follows a similar procedure to the + and - operators. The following program code is extracted from ARITH.CPP:

```
// arith.cpp
// overloads the arithmetic and
// arithmetic assignment operators
//...
class Point
{
//...
public:
//...
    Point operator * (const Point& p) ;
    Point operator / (const Point& p) ;
    Point operator += (const Point& p) ;
    Point operator -= (const Point& p) ;
    Point operator *= (const Point& p) ;
    Point operator /= (const Point& p) ;
}; // class Point
//...
// p1 * p2
inline Point Point::operator * (const Point& p)
{
    return Point (x*p.x, y*p.y, z*p.z) ;
}

// p1 / p2
inline Point Point::operator / (const Point& p)
{
    return Point (x/p.x, y/p.y, z/p.z) ;
}

// p1 += p2
```

```

inline Point Point::operator += (const Point& p)
{
    x += p.x ; y += p.y ; z += p.z ;
    return Point (x, y, z) ;
}

// p1 -= p2
inline Point Point::operator -= (const Point& p)
{
    x -= p.x ; y -= p.y ; z -= p.z ;
    return Point (x, y, z) ;
}

// p1 *= p2
inline Point Point::operator *= (const Point& p)
{
    x *= p.x ; y *= p.y ; z *= p.z ;
    return Point (x, y, z) ;
}

// p1 /= p2
inline Point Point::operator /= (const Point& p)
{
    x /= p.x ; y /= p.y ; z /= p.z ;
    return Point (x, y, z) ;
}
//...

```

It is now possible to write expressions of the form:

```

Point p, q, r ;
//...
r = p * q ; // multiplication
//...
r = p += q ; // chained assignment and
               // arithmetic assignment
//...

```

The last expression is legal because the arithmetic assignment operators return a `Point` object and thus can be chained to form complex expressions.

## 10.2 The Unary Increment (++) and Decrement (--) Operators

When a unary operator is overloaded it is not necessary to explicitly pass an object to an overloaded operator member function because the operation is implicitly performed on the object that calls the member function. To illustrate this, let us overload the prefix and postfix increment and decrement operators for `class Point`:

```
// inc&dec.cpp
```

```
// overloads the prefix and postfix
// increment and decrement operators
#include <iostream.h> // C++ I/O

// class Point
class Point
{
public:
    //...
    Point operator ++ () ;      // prefix
    Point operator ++ (int) ;   // postfix
    Point operator -- () ;
    Point operator -- (int) ;
}; // class Point
//...

// prefix increment operator (++p)
inline Point Point::operator ++ ()
{
    x += 1.0 ; y += 1.0 ; z += 1.0 ;
    return Point (x, y, z) ;
}

// postfix increment operator (p++)
inline Point Point::operator ++ (int)
{
    x += 1.0 ; y += 1.0 ; z += 1.0 ;
    return Point (x-1.0, y-1.0, z-1.0) ;
}

// prefix decrement operator (--p)
inline Point Point::operator -- ()
{
    x -= 1.0 ; y -= 1.0 ; z -= 1.0 ;
    return Point (x, y, z) ;
}

// postfix decrement operator (p--)
inline Point Point::operator -- (int)
{
    x -= 1.0 ; y -= 1.0 ; z -= 1.0 ;
    return Point (x+1.0, y+1.0, z+1.0) ;
}

void main ()
{
    // define objects p and q
    Point p (1.0, 2.0, 3.0), q (1.0, 2.0, 3.0), r ;

    // p
    cout << "p      (" << p.X () << ",  " << p.Y ()
```

```

    << ", "     << p.Z () << ")" " << endl ;

r = ++p ; // prefix
cout << "++p (" << r.X () << ", " << r.Y ()
    << ", "     << r.Z () << ")" " << endl ;

// q
cout << "q    (" << q.X () << ", " << q.Y ()
    << ", "     << q.Z () << ")" " << endl ;

r = q++ ; // postfix
cout << "q++ (" << r.X () << ", " << r.Y ()
    << ", "     << r.Z () << ")" " << endl ;
}

```

with output:

```

p    (1, 2, 3)
++p (2, 3, 4)
q    (1, 2, 3)
q++ (1, 2, 3)

```

The prefix overloaded increment operator is defined as:

```

inline Point Point::operator ++ () // prefix
{
    x += 1.0 ; y += 1.0 ; z += 1.0 ;
    return Point (x, y, z) ;
}

```

The overloaded prefix operator increments each of the three data members by the value of 1 and then returns the incremented object. This is in agreement with the output shown above for object p (++p).

We noted in Chapter 4 that for both the increment and decrement operators there exists both a prefix and postfix version. To overload the postfix increment operator C++ adopts a similar style to the prefix operator, but uses a dummy member function argument of type **int**:

```

inline Point Point::operator ++ (int) // postfix
{
    x += 1.0 ; y += 1.0 ; z += 1.0 ;
    return Point (x-1.0, y-1.0, z-1.0) ;
}

```

Note the return values of the object's data members. Each data member is reduced by the constant 1 before it is returned, thus returning the original object before incrementing occurred. This is in agreement with the C++ definition of a postfix increment operator for integral data types.

A similar procedure is exercised for the prefix and postfix decrement operators, but note that for the postfix decrement operator the constant 1 is added to each of the object's data members before the object is returned.

## 10.3 Physical Meaning of Overloaded Point Operators

Before we go crazy and overload every conceivable operator for our **Point class**, let's just stop and examine the physical meaning behind the operators that we have overloaded so far. The **Point class** defines the  $x$ -,  $y$ - and  $z$ -coordinate values of a point in a three-dimensional space. A vector is defined by a magnitude and a direction. Vectors require a start or initial point and a terminal point, which together define a vector's magnitude and direction. If we acknowledge the distinction between a point and a vector then some interesting contradictions occur when we perform otherwise simple operations on points and vectors. Consider the following cases of operating on a point,  $p$ , and a vector,  $v$  (Bowyer and Woodwark, 1993):

$$\begin{aligned} p - p &= v \\ v + v &= v \\ v - v &= v \\ p + v &= p \\ p - v &= p \\ p + p &= ? \end{aligned}$$

The above illustrates that a point minus a point is a vector, the addition or subtraction of two vectors is another vector, and the addition or subtraction of a vector and a point is a point, but the addition of two points is undefined! At present we have overloaded the subtraction operator for **class Point** as:

```
inline Point Point::operator - (const Point& p)
{
    return Point (x-p.x, y-p.y, z-p.z) ;
```

which returns a **Point**. From the above discussion, the subtraction of two points should return a vector, and later versions of **class Point** will overload the subtraction operator to return a **Vector**:

```
inline Vector Point::operator - (const Point& p)
{
    return Vector (x-p.x, y-p.y, z-p.z) ;
}
//...
Point p, q ;
//...
Vector v = p1 - p2 ;
```

Similarly, we quickly overloaded the prefix and postfix increment and decrement operators for the **Point class**, without actually asking ourselves what does  $p++$  actually mean? Does such an operation have any physical interpretation?

These examples of overloading operators for the **Point class** should illustrate the danger of overloading operators without giving much thought to their actual application.

## 10.4 The Relational Operators

Let's now examine overloading the relational operators ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $==$  and  $\neq$ ) for **class Point**. To begin with, consider the equality operators  $==$  and  $\neq$ , which have clear physical interpretations for the **Point class**. Since a test expression involving a relational operator will be either logical-true or logical-false, it would be nice if we could return a Boolean value (TRUE or FALSE) rather than an integer. Provided the Boolean FALSE is denoted by zero and Boolean TRUE by any non-zero value we remain consistent with C++'s definition of logical true and false. It was noted in Chapter 8 that a Boolean data type is most conveniently implemented as an **enum**:

```
enum Boolean
{
    FALSE, TRUE
};
```

Thus, making use of the Boolean type, the overloaded equality operators are implemented as:

```
Boolean Point::operator == (const Point& p)
{
    return (x==p.x && y==p.y && z==p.z) ? TRUE : FALSE ;
}

Boolean Point::operator != (const Point& p)
{
    return (x!=p.x && y!=p.y && z!=p.z) ? TRUE : FALSE ;
}
```

The return type of Boolean for the above two overloaded operator member functions is clearly more informative to a user of the **Point class** than, say:

```
int Point::operator == (const Point& p)
```

which indicates that any valid integer value can be returned. Refer to program COMPARE.CPP for further details and the implementation of overloading the other relational operators  $<$ ,  $\leq$ ,  $>$  and  $\geq$  for **class Point**.

## 10.5 The Assignment Operator

### 10.5.1 Default Assignment

By default, C++ will overload the assignment operator,  $=$ , for objects of user-defined classes. For instance, the following assignment:

```
p3 = p1 ;
```

will copy *exactly* (member by member) all of the data members from p1 to p3, without having to explicitly overload the assignment operator for the `Point` **class**. This will generally be what you require of the default assignment operator (thus its inclusion), but if you require an alternative action for the assignment operator then simply overload it.

When one object is assigned to another, a member by member (memberwise) copy is made as opposed to a bitwise copy. Beware of the default memberwise copy mechanism when pointer data members are declared.

### 10.5.2 Overloaded Assignment Operator

The following program overloads the assignment operator for the `Point` **class** and simply displays a message when called:

```
// assign.cpp
// illustrates overloading the assignment operator
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // public member functions
    //...
    // overloaded operator
    Point operator = (const Point& p) ;
}; // class Point

// overloaded assignment operator
Point Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    cout << "overloaded = operator called" << endl ;
    return Point (x, y, z) ;
}

void main ()
{
    // define objects
    Point p1 (3.0, 4.0, 5.0), p2 ;

    cout << "p1(" << p1.X () << ", " << p1.Y ()
        << ", " << p1.Z () << ")" << endl ;
    cout << "p2(" << p2.X () << ", " << p2.Y ()
```

---

```

    << ", " << p2.Z () << ")" << endl ;

p2 = p1 ; // assignment

cout << "p2(" << p2.X () << ", " << p2.Y ()
    << ", " << p2.Z () << ")" << endl ;
}

```

which generates the output:

```

p1(3, 4, 5)
p2(0, 0, 0)
overloaded = operator called
p2(3, 4, 5)

```

The overloaded assignment operator member function definition is:

```

Point Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    cout << "overloaded = operator called" << endl ;
    return Point (x, y, z) ;
}

```

which indicates that the three data members of a `Point` object on the right-hand side of the `=` operator are directly assigned to the data members of the object on the left-hand side. The return type of the member function is `Point`, thus enabling chaining of the assignment operator:

```

//...
p1 = p2 = p3 = p4 ; // chained assignment

```

The overloaded assignment operator member function is passed a `const` object by reference rather than by value, and a `Point` object is returned. When we examine pointers, a better way of overloading the assignment operator will be presented which makes use of the `this` pointer. The `this` pointer enables a function to return the object which invokes the overloaded operator member function.

## 10.6 Non-Member Overloaded Operator Functions

Overloaded operator functions can be non-member functions:

```

// non_mem.cpp
// illustrates non-member overloaded operator functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{

```

```

// ...
};

// non-member overloaded operators:

// p1 + p2
inline Point operator + (const Point& p1, const Point& p2)
{
    return Point (p1.X() + p2.X(), p1.Y() + p2.Y(), p1.Z() + p2.Z());
}

// p1 - p2
inline Point operator - (const Point& p1, const Point& p2)
{
    return Point (p1.X() - p2.X(), p1.Y() - p2.Y(), p1.Z() - p2.Z());
}

void main ()
{
    // define objects
    Point p1 (3.0, 4.0, 5.0);
    Point p2 (1.0, 2.0, 3.0);
    Point p3, p4, p5;

    p3 = p1 + p2; // addition
    p4 = p1 - p2; // subtraction

    // O/P
    cout << "p3 (" << p3.X () << ", " << p3.Y ()
        << ", " << p3.Z () << ")" << endl;
    cout << "p4 (" << p4.X () << ", " << p4.Y ()
        << ", " << p4.Z () << ")" << endl;
}

```

with output:

```

p3 (4, 6, 8)
p4 (2, 2, 2)

```

Since the overloaded binary `::operator+()` and `::operator-()` functions are non-member functions, they require two arguments with object data member access via member functions. Although non-member overloaded operators are legal in C++ they are rarely used.

## 10.7 The Array Subscript Operator

In C++ it is possible to overload the array subscript operator `[]`. To illustrate overloading the `[]` operator, let us represent **class** `Point`'s three data members as an array of type **double**:

```
class Point
```

```
{
double pt[3] ;
//...
};
```

where `pt[0]`, `pt[1]` and `pt[2]` represent the *x*-, *y*- and *z*-coordinates of a point. Such a representation will assist in accessing an object's data members via the overloaded subscript operator. The following program illustrates the overloaded subscript operator:

```
// subscript.cpp
// illustrates overloading the subscript operator
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()

class Point
{
private:
    double pt[3] ;
public:
    // constructors
    Point ()
    { pt[0] = pt[1] = pt[2] = 0.0 ; }
    Point (double x_arg, double y_arg, double z_arg=0.0)
    { pt[0] = x_arg ; pt[1] = y_arg ; pt[2] = z_arg ; }
    // overloaded operator
    double& operator [] (int i) ;
}; // class Point

// overloaded subscript operator
double& Point::operator [] (int i)
{
if (i<0 || i>=3)
{
    cout << "array index " << i
        << ": out of bounds" << endl ;
    exit (EXIT_SUCCESS) ;
}
return pt[i] ;
}

void main ()
{
    Point p (1.0, 2.0, 3.0) ;

    // access
    cout << "p (" << p[0] << ", " << p[1]
        << ", " << p[2] << ")" << endl ;

    // assign
    p[2] = 0.0 ;
```

```

cout << "p( " << p[0] << ", " << p[1]
     << ", " << p[2] << ")" << endl ;

// out of bounds index
p[3] = 1.0 ;
}

```

with output:

```

p (1, 2, 3)
p (1, 2, 0)
array index 3: out of bounds

```

The definition of the overloaded subscript operator is:

```

double& Point::operator [] (int i)
{
    if (i<0 || i>=3)
    {
        cout << "array index " << i
            << ": out of bounds" << endl ;
        exit (EXIT_SUCCESS) ;
    }
    return pt[i] ;
}

```

The return type of the member function is a reference to **double**. This allows the function to return a reference to an element of the *pt* array so that the subscript operator can be used on the left-hand side of an assignment statement:

```
p[2] = 0.0 ;
```

We noted in Chapter 7 when discussing arrays that C++ does not automatically perform bound checking of an indexed array. The beauty of overloading the subscript operator is that it allows us to perform our own bound checking. The above member function definition tests whether the index value, *i*, is not equal to 0, 1 or 2. If the index value is outside the acceptable range, a run-time error message is displayed and the program is successfully terminated.

We shall revisit overloading the [ ] operator when we develop *Vector* and *Matrix* classes.

## 10.8 Composite Operators

Unfortunately, at present C++ does not support overloading of composite operators, although the usefulness of such a feature is acknowledged (Stroustrup, 1994, pp. 251–2). This does have some important consequences for indexing the elements of a user-defined two-dimensional array **class**. For instance, we observed in Chapter 8 that to index an element of a two-dimensional array in C++ we use the general syntax:

```
array_name[row_index][col_index]
```

Thus, to overload the composite operators `[] []` for integer elements of a two-dimensional array of a **class** `Array2D` we would expect to use a member function declarator of the form:

```
int& Array2D::operator [] [] (int i, int j) // error
//...
Array2D array ;
//...
array[i][j] = x ; // not possible!
```

This is illegal in C++ because composite operators cannot be overloaded.

In addition, it is not possible to use a single subscript operator in either of the two forms:

```
int& Array2D::operator [] (int i, int j) // error
//...
Array2D array ;
//...
array[i][j] = x ; // not possible!

array[i, j] = x ; // not possible!
```

The first application of `operator[]` is illegal because its left-hand side operand (`array[i]`) is not an `Array2D` object. The second application of `operator[]`, which is similar to array indexing in other languages, such as Fortran, is illegal because this operation involves the use of three operands (`array, i` and `j`). We observe from Appendix C that `[]` is a binary operator, and attempting to use `[]` with three operands is an attempt to alter the arity<sup>1</sup> of the operator, which C++ does not allow.

Alternatively, the following approach must be adopted:

```
int& Array2D::operator () (int i, int j)
//...
Array2D array ;
//...
array(i, j) = x ;
```

This approach works, but it conflicts with C++'s philosophy of indexing two-dimensional arrays for integral types. In fact, indexing the elements of an object of **class** `Array2D` via the overloaded `operator()` member function resembles a function call more than an array index – an awful notation. The following program demonstrates an implementation of a two-dimensional **class** `Array2D` which overloads the `()` operator as a two-dimensional array subscript operator:

```
// composit.cpp
// illustrates overloading an array indexing operator
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()
```

---

<sup>1</sup> For instance, the insertion operator `<<` is a binary operator, and it is illegal to alter the arity of the operator from binary to unary. For a further discussion of the precedence, associativity and arity of the C++ operators, refer to Chapter 4.

```

class Array2D
{
private:
    int a[2][2] ;
public:
    // constructor
    Array2D ()
    { a[0][0]=0 ; a[0][1]=1 ; a[1][0]=2 ; a[1][1]=3 ; }
    // overloaded operator
    int& operator () (int i, int j) ;
}; // class Array2D

// overloaded subscript operator
int& Array2D::operator () (int i, int j)
{
    if (i<0 || j>=2 || i>=2 || j>=2)
    {
        cout << "array index " << i
            << ": out of bounds" << endl ;
        exit (EXIT_SUCCESS) ;
    }
    return a[i][j] ;
}

void main ()
{
    Array2D array ;

    for (int i=0; i<2; i++)
    {
        for (int j=0; j<2; j++)
            cout << array(i,j) << " " ;
        cout << endl ;
    }
}

```

with output:

```

0 1
2 3

```

When pointers are discussed in Chapter 12 a **Matrix** **class** will be developed which allows element indexing in the C++ style:

```

Matrix m (5, 5) ; // object m of class Matrix
//...
m[0][3] = 2.12 ;

```

The problem with indexing the elements of an object of the two-dimensional array **class**, **Array2D**, does not lie with the overloaded operator function **operator[]()** but with the implementation of **class** **Array2D**.

Another frequently requested use of composite operators occurs when raising a number  $x$  to a power  $y$ ,  $x^y$ . Such an operation in Fortran could make use of the exponentiation operator, `**`:

```
v = x ** y
```

whereas Basic uses the `^` operator:

```
v = x ^ y
```

Alternatively, C++ possesses a library function called `pow()`, which conveniently performs this operation:

```
v = pow (x, y) ;
```

The number of times that  $x^y$  is required when programming does not warrant the addition of a composite exponentiation operator.

## 10.9 Conversions

Chapter 4 illustrated data type conversions of the form:

```
int i ;
//...
double d = i ;
```

which implicitly casts an integer into a floating-point `double`. C++ allows a programmer to perform casting from one type to another explicitly using either of the following:

```
int i ;
//...
double d = (int) i ; // C style
double d = int (i) ; // C++ style
```

The first approach is the C syntax, while the second is the generally adopted style in C++, for reasons that will shortly become evident.

To illustrate conversions, let us add a constructor and a *conversion member function* to the Point `class`:

```
// convert.cpp
// illustrates conversions between different types
#include <iostream.h> // C++ I/O
#include <math.h> // sqrt()

// class Point
class Point
{
private:
    // private data members
```

```

double x, y, z ;
public:
// constructors
Point ()
: x (0.0), y (0.0), z (0.0) {}
Point (double x_arg, double y_arg, double z_arg=0.0)
: x (x_arg), y (y_arg), z (z_arg) {}
Point (double d);
//...
// conversion member function
operator double ();
};

// class Point

// constructor
// converts double to Point
Point::Point (double d)
{
x = y = z = d;
}

// conversion member function
// converts Point to double
Point::operator double ()
{
double r_sq = x*x + y*y + z*z ;
return sqrt (r_sq);
}

void main ()
{
Point p1 (5.0); // double to Point

cout << "p1 (" << p1.X () << ", " << p1.Y ()
<< ", " << p1.Z () << ")" << endl ;

double distance1 = double (p1); // Point to double

cout << "distance1: " << distance1 << endl ;

Point p2 = 10.0; // double to Point

cout << "p2 (" << p2.X () << ", " << p2.Y ()
<< ", " << p2.Z () << ")" << endl ;

double distance2 = p2; // Point to double

cout << "distance2: " << distance2 << endl ;
}

```

with output:

```
p1 (5, 5, 5)
distance1: 8.66025
p2 (10, 10, 10)
distance2: 17.3205
```

To convert from a **double** to a Point, the following one-argument *converting constructor* is defined:

```
Point::Point (double d)
{
    x = y = z = d ;
}
```

This constructor simply assigns the constructor argument to each data member of an object of **class** Point. Having defined such a constructor, the following object definitions are feasible:

```
Point p1 (5.0) ;
Point p2 = 10.0 ;
```

The second statement calls the one-argument constructor because a floating-point number is being assigned to a Point object, which requires converting. Since we have not overloaded the assignment operator in the present program, the compiler will use the one-argument constructor to convert from a **double** to a Point. If the assignment operator had also been overloaded, then the overloaded assignment operator member function would be called instead of the constructor.

To perform the contrary case of converting from a Point to a **double**, the following conversion member function is defined:

```
Point::operator double ()
{
    double r_sq = x*x + y*y + z*z ;
    return sqrt (r_sq) ;
}
```

This function can be used in statements of the form:

```
double distance1 = double (p1) ;
double distance2 = p2 ;
```

The conversion member function calculates and returns the minimum distance from a Point object to the origin. The conversion member function could have taken any number of forms. I simply chose the most intuitive way of determining a Point's distance from the origin.

Alternatively, you could use the C style of casting if preferred:

```
double distance = (double) q ; // Point to double
```

Consider again the Point **class** in CONVERT.CPP, but with the **explicit** keyword specifier used in the declaration of the one-argument constructor:

```
// explicit.cpp
```

```
// illustrates the explicit specifier

class Point
{
private:
    double x, y, z ;
public:
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    explicit Point (double d) // explicit constructor
        : x (d), y (d), z (d) {}
    //...
};

void main ()
{
    Point p = 1.0 ; // error: can't convert double to Point

    Point q (2.0) ; // o.k.: explicit constructor call
}
```

A non-converting constructor declared with the **explicit** specifier prevents the constructor from being used as a converting constructor unless the conversion is explicitly indicated by a constructor call. The **explicit** specifier can only be used in constructor declarations within a **class** declaration.

## 10.10 A String **class**

This section examines a String **class**, which is an excellent **class** to illustrate the usefulness of operator overloading. It was shown in Chapter 7 that C++ represents a string of characters as an array. C++ does provide a **class** for representing strings, called **string**, whose header file is **CSTRING.H**, but rather than examining **string**'s declaration let us consider the following program. The String **class** implemented below encapsulates a string data member with a maximum length of 80 characters:

```
// string.cpp
// illustrates a String class
#include <iostream.h> // C++ I/O
#include <string.h> // strcpy()
#include <stdlib.h> // exit()

const int STR_LEN = 80 ; // maximum length of String

class String
{
private:
    char string[STR_LEN] ;
public:
    // constructor
```

```
String (char s[]{"\0"})
{ strcpy (string, s) ; }
// member function
void Show ()
{ cout << string << endl ; }
// overloaded operators
String operator = (const String& s) ;
String operator = (char s[]) ;
String operator + (const String& s) ;
String operator + (char s[]) ;
} // String class

// overloaded = operator:

// String to String
String String::operator = (const String& s)
{
strcpy (string, s.string) ;
return String (string) ;
}

// C++ string to String
String String::operator = (char s[])
{
strcpy (string, s) ;
return String (string) ;
}

// overloaded + operator:

// String + String
String String::operator + (const String& s)
{
// don't overrun
if ((strlen (string) + strlen (s.string))> STR_LEN)
{
cout << "concatenated strings too long" << endl ;
exit (EXIT_SUCCESS) ;
}
String t ;

strcpy (t.string, string) ; // string to t
strcat (t.string, s.string) ; // add two strings
return t ;
}

// C++ string + String
String String::operator + (char s[])
{
// don't overrun
if ((strlen (string) + strlen (s))> STR_LEN)
```

```

    {
    cout << "concatenated strings too long" << endl ;
    exit (EXIT_SUCCESS) ;
}
String t ;

strcpy (t.string, string) ;      // string to t
strcat (t.string, s) ;          // add two strings
return t ;
}

void main ()
{
String s1 ;                      // null String
s1.Show () ;

String s2 ("operator ") ;        // initialised String
s2.Show () ;

String s3 = "overloading " ; // C++ string to String
s3.Show () ;

String s4 = s2 + s3 + "is cool " ; // concatenate
s4.Show () ;
}

```

with output:

```

operator
overloading
operator overloading is cool

```

The one-argument constructor converts a C++ string to a String and assigns a null string as default:

```

String (char s[] = "\0")
{ strcpy (string, s) ; }

```

The String **class** overloads the assignment operator so that one String object can be assigned to another String object or a C++ string can be assigned to a String. The String::**operator=()** declarations are:

```

String operator = (const String& s) ;
String operator = (char s[]) ;

```

which are overloaded, overloaded = operator member functions (i.e. doubly overloaded). Thus, C++ allows overloaded operator member functions to be overloaded just as if they were *normal* member functions.

With the above overloaded assignment operator member functions, statements of the form below are perfectly valid:

---

```
String s3 = "overloading " ; // C++ string to String
```

This statement converts a C++ string to a `String` object via the overloaded assignment operator rather than the one-argument constructor. Similarly, the addition operator is overloaded to enable the concatenation either of two objects of `class String` or of a C++ string and a `String` object:

```
String s4 = s2 + s3 + "is cool " ; // concatenate
```

Since the overloaded `+` operator returns a `String`, it is possible to use chained `+` operators.

## 10.11 Overloading the C++ Input Extraction (`>>`) and Output Insertion (`<<`) Operators

All of the classes discussed so far have displayed the data members of a `class` either by using access member functions or a member function specifically designed for outputting an object's data members. For instance, the data members of an object `p` of `class Point` have typically been displayed using the following:

```
Point p ;
//...
cout << "Point p (" << p.X () << ", " << p.Y () << ", "
<< p.Z () << ")" << endl ;
```

Alternatively, a member function could be defined which performs a similar task:

```
class Point
{
//...
void Display () ;
//...
};

void Point::Display ()
{
cout << "(" << x << ", " << y << ", " << z << ")" << endl ;
}

void main ()
{
Point p ;
//...
p.Display () ;
//...
}
```

Such approaches work fine, but we have seen throughout the text that C++ offers a more attractive form of output for integral types by using the insertion operator (`<<`):

```

char    ch ;
int     i ;
double d ;
//...
cout << "character ch: " << ch ;
cout << "integer i : " << i ;
cout << "double d : " << d ;

```

cout is an object of **class** ostream, which provides standard output. The insertion operator is sufficiently overloaded to output all of C++'s integral types so that a user of the ostream **class** does not have to specify the type of identifier or object when performing output. Typically, the form of the IOSTREAM.H header file for the overloaded << operator is:

```

class ostream : virtual public ios
{
//...
public:
//...
// character
ostream& operator << (char) ;
ostream& operator << (signed char) ;
ostream& operator << (unsigned char) ;

// integer and floating-point
ostream& operator << (short) ;
ostream& operator << (unsigned short) ;
ostream& operator << (int) ;
ostream& operator << (unsigned int) ;
ostream& operator << (long) ;
ostream& operator << (unsigned long) ;
ostream& operator << (float) ;
ostream& operator << (double) ;
ostream& operator << (long double) ;
//...
};

```

which illustrates that the << operator is overloaded for all of the integral data types. If the << operator is not overloaded for a given **class**, T, it is the responsibility of the designer or user of **class** T to overload the << operator. For example, let's examine overloading the insertion and extraction operators for **class** Point:

```

// over_io.cpp
// overloads C++'s insertion, <<, and extraction, >>,
operators
#include <iostream.h> // C++ I/O

// class Point
class Point
{
public:
// public data members

```

```

double x, y, z ;
// constructors
Point ()
    : x (0.0), y (0.0), z (0.0) {}
Point (double x_arg, double y_arg, double z_arg=0.0)
    : x (x_arg), y (y_arg), z (z_arg) {}
}; // class Point

// overloaded operators:

inline ostream& operator << (ostream& s, const Point& p)
{
    s << "(" << p.x << ", " << p.y << ", " << p.z
        << ")" << endl ;
    return s ;
}

inline istream& operator >> (istream& s, Point& p)
{
    cout << "enter x, y and z coordinates of a Point: " ;
    s >> p.x >> p.y >> p.z ;
    return s ;
}

void main ()
{
    Point p (1.0, 2.0, 3.0), q ;

    // O/P p
    cout << p ;

    // I/P q
    cin >> q ;

    // O/P q
    cout << q ;
}

```

Some user interaction is:

```

(1, 2, 3)
enter x, y and z coordinates of a Point: 4 5 6
(4, 5, 6)

```

It is worth noting that the overloaded `<<` and `>>` operator functions are not member functions of **class** `Point` and that the `x`, `y` and `z` data members are declared **public**. `Point`'s data members are made **public** because the overloaded insertion and extraction operator functions are member functions of classes `ostream` and `istream`, respectively. As a result, the object on the left-hand side of the `<<` or `>>` operators is implicitly assumed to be of **class** `ostream` or `istream`, respectively, so overloaded insertion and extraction functions must be non-member functions of a user-defined **class**.

Let's examine the overloaded insertion operator:

```
inline ostream& operator << (ostream& s, const Point& p)
{
    s << "(" << p.x << ", " << p.y << ", " << p.z
        << ")" << endl ;
    return s ;
}
```

Two objects are passed as arguments to the function. The first argument is a reference to the output stream which occurs on the left-hand side of the `<<` operator. The second argument is a `const` reference (since the function does not alter the data members of the object) to the object on the right-hand side of the `<<` operator. The function returns a reference to an object of `class ostream`, which allows the overloaded operator to be chained.

Within the body of the above overloaded operator function the three data members of an object of `class Point` are output using the argument object `s`. The insertion operator, `<<`, can be applied to any specified stream, and thus using the argument stream object, `s`, instead of the `cout` object makes the overloaded insertion operator function more general. When we discuss file streams we shall see that the argument stream object can be used to direct output to the screen or a file without modification of the overloaded `operator<<()` member function.

A similar approach is taken when overloading the extraction operator, `>>`:

```
inline istream& operator >> (istream& s, Point& p)
{
    cout << "enter x, y and z coordinates of a Point: " ;
    s >> p.x >> p.y >> p.z ;
    return s ;
}
```

A user is first prompted to enter a `Point`'s `x`, `y` and `z` data members, after which `double` values are assigned to the object's appropriate data members. The second argument, `p`, is not a `const` reference because `p`'s data members are assigned values, and hence changed by the function. Unfortunately, in defining the overloaded `operator>>()` function we have inserted the standard output `cout` stream object within the function definition. This is a common mistake to make and greatly reduces the generality of the overloaded `operator>>()` function by restricting its use to standard input and output. A better definition is the simpler:

```
inline istream& operator >> (istream& s, Point& p)
{
    return s >> p.x >> p.y >> p.z ;
}
```

Rather than declaring the data members of `class Point` `public` in `OVER_IO.CPP` the `X()`, `Y()` and `Z()` member functions can be used, provided they each return a reference to a `double`:

```
// over_iol.cpp
// further illustrates overloading
// the insertion and extraction
```

```

// operators
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    //...
    // member functions
    double& X () { return x; }
    //...
}; // class Point

// overloaded operators:

inline ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.X() << ", " << p.Y() << ", " << p.Z()
        << ")" << endl;
}

inline istream& operator >> (istream& s, Point& p)
{
    return s >> p.X() >> p.Y() >> p.Z();
}

void main ()
{
    Point p (1.0, 2.0, 3.0), q;

    // O/P p
    cout << p;

    // I/P q
    cout << "enter x, y and z coordinates of a Point: ";
    cin >> q;

    // O/P q
    cout << q;
}

```

If you prefer not to return a reference to a **double** for member functions *X()*, *Y()* and *Z()* while maintaining the **private** declaration of *x*, *y* and *z*, the *Point* three-argument constructor can be used in the definition of **operator>>()**:

```

// over_io2.cpp
// another approach to
// overloading the insertion

```

```

// and extraction operators
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // member functions
    double X () const { return x; }
    //...
}; // class Point

// overloaded operators:

inline ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.X() << ", " << p.Y() << ", " << p.Z()
        << ")" << endl;
}

inline istream& operator >> (istream& s, Point& p)
{
    double x_temp, y_temp, z_temp;

    s >> x_temp >> y_temp >> z_temp;

    // use Point 3-arg. constructor
    p = Point (x_temp, y_temp, z_temp);

    return s;
}

void main ()
{
    //... as before
}

```

The overloaded extraction operator first places the x,y and z data members of a Point object into temporary **double** variables and then uses the three-argument constructor of Point to assign a Point object to argument p.

When we discuss **friends** in the next chapter we shall revisit the overloaded insertion and extraction operators, so that the above techniques of declaring a Point's data members **public**, using member functions and a constructor, are removed.

## 10.12 Operators Which Cannot be Overloaded

C++ is very generous as to which of the available operators can be overloaded. However, there are a few exceptions. Operators that cannot be overloaded are:

::	scope resolution
.	direct member access
<b>sizeof</b>	size, in bytes, of an object
.*	direct member pointer access
?:	conditional

An attempt to overload either of the above operators will result in a compilation error of the form ‘Operator cannot be overloaded’.

## 10.13 Restrictions Attached to Operator Overloading

When a given operator is overloaded, the precedence, associativity and arity of the operator cannot be altered. Therefore, the usual rules attached to the use of operators in expressions are maintained:

```
//...
10 - 4 * 3 ;      // -2
//...
p1 - p2 * p3 ;  // objects of class Point
```

## 10.14 Choose Carefully Which Operators to Overload

When a programmer writes the following expression, which adds two variables of the integral type **int** and then proceeds to assign the result to another **int**:

```
int a, b, c ;
//..
c = a + b ;
```

the meaning attached to the addition and assignment operators is clear. However, if you overload the addition operator for the **Point class**, a user of the **class** will probably expect a similar operation of two **Point** objects as an addition of two integral types:

```
Point p, q, r ;
//...
r = p + q ;
```

Operator overloading, if used wisely, can give a **class** a more natural application. However, attaching an abstract meaning to an operator can be very confusing. For instance, if the % operator is overloaded for the **Point class** to return the distance between two **Point** objects, we could write:

```
double operator % (const Point& p)
{
    return sqrt ((x-p.x)*(x-p.x) +
                 (y-p.y)*(y-p.y) +
                 (z-p.z)*(z-p.z)) ;
}
//...
Point p1, p2 ;
//...
double distance = p1 % p2 ;
```

The usual meaning attached to the `%` operator is the arithmetic modulus. A user of the `Point` **class** would probably find the overloaded `%` operator confusing because it conflicts with the original meaning attached to the modulus operator. Therefore, when overloading operators try to retain an operator's original meaning. For this reason, it would be better to define a member function instead of attaching an abstract meaning to an operator:

```
//...
double distance = p1.DistanceBetweenPoints (p2) ;
```

The ultimate test of whether overloading a given operator is worthwhile is how much the overloaded operator is actually used. If the operator is rarely used or frequently used incorrectly then I would seriously re-evaluate the decisions made for overloading an operator.

## 10.15 Summary

Operator overloading is not an essential feature in the definition of an object-oriented programming language and is not supported by object-oriented languages such as Eiffel and Oberon-2. Nevertheless, operator overloading is a powerful feature of C++ programming, particularly when manipulating objects of user-defined numeric classes, such as `Point`, `Vector` and `Matrix`.

Member functions provide a means of accessing and sending messages to a specific object's data members. Overloaded operators similarly allow a programmer to operate on an object's data members, but in a more natural manner. An overloaded operator member function declaration of a given **class** is similar to the **class**'s member functions except for the keyword `operator`, which indicates to the compiler that a given operator will have a specific, user-defined meaning when applied to objects of the **class**.

Both unary and binary operators can be overloaded, and all of the available operators can be overloaded except the following five: direct member access `(.)`, scope resolution `(::)`, direct member pointer access `(.* )`, conditional operator `(?:)` and the `sizeof` operator. The assignment operator is overloaded by default for objects of a user-defined **class**. The default assignment operator will copy all of the data members from one object exactly into another object.

At present, composite operators cannot be overloaded in C++.

One-argument constructors and conversion member functions can be designed to enable both implicit and explicit conversions from one type or **class** to another.

The C++ insertion and extraction operators can be overloaded for performing output and input, respectively, on objects of user-defined classes.

When we examine pointers it will be shown that in certain circumstances the **this** pointer offers us a favourable alternative for returning an object from an overloaded operator member function. The **this** pointer is a pointer to the object that invoked the overloaded operator member function and is passed implicitly when the function is called.

## Exercises

- 10.1 Overload the `+=` operator for the String **class** presented in STRING.CPP.
- 10.2 Further develop the `Int` **class** of Exercise 9.2 so that `Int` supports an overloaded modulus operator `%`. In your implementation of `operator%` () use the C++ library function `div()`:

```
div_t div (int numer, int denom) ; // STDLIB.H

struct div_t
{
    int quot ; // quotient
    int rem ; // remainder
};
```

`div()` returns `numer` divided by `denom`, placing the quotient and remainder in the structure `div_t`.

- 10.3 Design a **class** called `Tensor` which encapsulates the properties of a mathematical tensor of rank two,  $a_{ij}$  ( $i,j=x,y,z$ ). Define for `Tensor` constructors and overloaded `+`, `-`, `<<` and `>>` operators. Also design `SymmetricTensor` and `AntiSymmetricTensor` tensor classes which encapsulate a symmetric tensor,  $a_{ij}=a_{ji}$ , and antisymmetric tensor,  $a_{ij}=-a_{ji}$ , respectively. Provide adequate conversion functions for each of the three classes `Tensor`, `SymmetricTensor` and `AntiSymmetricTensor`, so that objects are implicitly converted from one **class** to another.

- 10.4 Provide overloaded equality (`==`) and inequality (`!=`) operators for the following **class**:

```
class Vector3D
{
    private:
        double a, b, c ;
    public:
        Vector3D (double _a, double _b, double _c)
            : a (_a), b (_b), c (_c) {}
};
```

- 10.5 Provide a type **double** one-argument non-converting constructor for the **class** `Vector3D` of Exercise 10.4 using the **explicit** keyword.
- 10.6 Develop a `Time` **class** which encapsulates `hour`, `minute` and `second` data members. Provide overloaded `+`, `-`, `++` and `--` overloaded operators for `Time`.
- 10.7 For the `Vector3D` **class** of Exercise 10.4 overload the `<<` and `>>` operators while maintaining that `a`, `b` and `c` are **private** members of `Vector`.

# Friends

e2:e4

*Friends are so important in C++ that they deserve a chapter all to themselves. Both classes and functions can be made friends of a given class. A friend has access to a class's private data members, although it is not a member of the class.*

*This chapter discusses friend functions, classes and overloaded operators. Friend functions are particularly useful for overloading binary operators where the left-hand operand object is of a class other than the class in which the overloaded operator function is a member. An entire class can be made a friend of another class and consequently has access to the friend's private data members. The C++ insertion, <<, and extraction, >>, overloaded operator functions are frequently made friends to allow the C++ stream objects access to the data members of an object of a user-defined class.*



## 11.1 friend Functions

To date, a **class**'s data members have been designated as either **private** or **public**. A **private** data member can only be accessed directly within the **class** or indirectly via a member function, whereas a **public** data member can be accessed directly by an object of the data member's respective **class**. However, a function which is made a **friend** of a **class** is allowed direct access to a **class**'s **private** data members. Let us consider the following program, which declares a **class** **Circle** and defines a function **Area()** that is a **friend** of **class** **Circle**:

```
// fri_func.cpp
// illustrates friend functions
#include <iostream.h> // C++ I/O

class Circle
{
private:
    double radius;
```

```

public:
    // constructors
    Circle ()
        : radius (0.0) {}
    Circle (double r)
        : radius (r) {}
    // friend
    friend double Area (const Circle& c) ;
}; // class Circle

// friend function Area() of class Circle
double Area (const Circle& c)
{
    // area=pi*r^2
    return 3.141593 * c.radius * c.radius ;
}

void main ()
{
    Circle circle (5.0) ;

    double area = Area (circle) ;

    cout << "area of circle: " << area << endl ;
}

```

with output:

```
area of circle: 78.5398
```

The function `Area()` is not a member function of `class Circle` and yet has access to the `radius` data member of an object of `class Circle` that is passed to the function as an argument. `Area()` is declared within the body of `class Circle`'s declaration as:

```
friend double Area (const Circle& c) ;
```

The keyword `friend` precedes the return type of the function. In fact, it is irrelevant whether the keyword precedes the return type or vice versa, but `friend` is conventionally placed before the return type. In addition, it is irrelevant whether the `friend` function declaration is placed in the `private` or `public` sections of a `class` declaration, although `friends` are generally placed in the `public` section. I always place `friend` declarations at the end of the `public` section (if one and only one exists!) of a `class` declaration. The general syntax of a `friend` function, `FunctionName()`, declaration is:

```

class ClassName
{
    //...
    friend return_type FunctionName (parameter_list) ;
    //...
};
```

The keyword **friend** is only used in a member function declaration and not in the respective declarator of the member function definition. A **friend** function has access to the **private** data members of the **class** in which the friendship is made. In the above program, **Area ()** has access to the **radius** data member of an object of **class Circle**:

```
double Area (const Circle& c)
{
    // area=pi*r^2
    return 3.141593 * c.radius * c.radius ;
}
```

The **radius** data member of an object **c** is accessed via the member access operator (.) in the usual manner.

**Area ()** is called with the statement:

```
double area = Area (circle) ;
```

illustrating that the member access operator (.) is not required for function calls since **Area ()** is a *normal* global function. The **friend** function call is more in line with the calling of other global functions rather than the alternative **class** member function call:

```
double area = circle.Area () ;
```

Although the **friend** function **Area ()** is in agreement with procedural function calls, it conflicts with an object-oriented approach, which clearly associates an area operation with **class Circle**.

### 11.1.1 So Why Use **friend** Functions?

In the preceding example, **class Circle** could clearly have been designed to include an **Area ()** member function instead of a **friend** function. There are, however, circumstances in which **friend** functions can be very useful. We shall see later that **friend** functions can be useful for overloading operators and for giving a more general application to overloaded operators. In particular, **friends** are important for overloading C++'s input and output stream insertion (<<) and extraction (>>) operators for use with user-defined **class** objects.

As an illustration of the usefulness of **friends**, consider the following program. The program consists of three **class** declarations A, B and C. It is required that the data members of each **class** are simultaneously displayed. We could simply incorporate a member function in each **class** which appropriately displayed an object's data member:

```
class A
{
private:
    int a ;
public:
    void Display ()
        { cout << "a: " << a << endl ; }
    //...
};

class B
```

```

{
private:
    int b ;
public:
    void Display ()
    { cout << "b: " << b << endl ; }
    //...
};

//...

```

However, such an approach requires three different and separate member functions. If this overhead is unacceptable and the access speed of an object's data member is important, a better approach is to have a single *Display()* function which is a **friend** to each of the three classes A, B and C:

```

// abc.cpp
// illustrates a function which is a friend
// of three classes A, B and C
#include <iostream.h> // C++ I/O

// class declarations
class B ;
class C ;

class A
{
private:
    int a ;
public:
    A (int a_arg) { a = a_arg ; }
    // friend
    friend void Display (A a_disp, B b_disp, C c_disp) ;
};

class B
{
private:
    int b ;
public:
    B (int b_arg) { b = b_arg ; }
    // friend
    friend void Display (A a_disp, B b_disp, C c_disp) ;
};

class C
{
private:
    int c ;
public:
    C (int c_arg) { c = c_arg ; }
    // friend

```

---

```

friend void Display (A a_disp, B b_disp, C c_disp) ;
};

// friend function to classes A, B and C
void Display (A a_disp, B b_disp, C c_disp)
{
    cout << "A::a = " << a_disp.a << "; B::b = " << b_disp.b
        << "; C::c = " << c_disp.c << endl ;
}

void main ()
{
    A a_main (1) ;
    B b_main (2) ;
    C c_main (3) ;

    Display (a_main, b_main, c_main) ;
}

```

with output:

```
A::a = 1; B::b = 2; C::c = 3
```

Thus, a single function is called in *main()* which has access to the data members of classes A, B and C.

Note the forward declarations of classes B and C before the **class** declaration of A:

```

class B ;
class C ;

class A
{
    //...

```

We have seen that C++ is strict concerning the definition of a variable or object before it is used and calling functions before they are defined. Similarly, in C++ a **class** must be declared before it is referenced. Within the declaration of **class** A the **friend** function *Display()* references classes B and C before they are declared:

```

class A
{
    //...
friend void Display (A a_disp, B b_disp, C c_disp) ;
};

class B
{
    //...
};

//...

```

Therefore, it is necessary to indicate to the compiler that the **class** declarations B and C exist within the program.

### 11.1.2 **friend** Functions: Good, Bad or Ugly?

At first, **friend** functions seem to contradict completely C++'s philosophy of **private** data member access by allowing access to any old function. However, a closer look at **friend** functions illustrates that this is not the case. In the above program, FRI\_FUNC.CPP, a **class** Circle was declared as follows:

```
class Circle
{
private:
    double radius ;
public:
    //...
    // friend
    friend double Area (const Circle& c) ;
};
```

The declaration indicates that the function *Area ()* is a **friend** of **class** Circle. Therefore, the designer of **class** Circle specifically made *Area ()* a **friend** and thus allowed *Area ()* access to the radius data member of an object of **class** Circle. A user of **class** Circle who does not have access to Circle's declaration cannot define arbitrary functions which have access to Circle's **private** data members.

However, **friend** functions should be used with caution and reserved for special cases in which there is no other alternative or when the temptation of using a **friend** cannot be resisted. As we shall shortly see, **friends** can give increased functionality to overloaded operators in certain cases.

## 11.2 **friend** Classes

It is also possible to make an entire **class** a **friend** of another **class**. A **friend class** has full access to **private** data members of a **class** without being a member of that **class**. The following program helps illustrate **friend** classes:

```
// fr_class.cpp
// illustrates friend classes
#include <iostream.h> // C++ I/O

// forward declare Vector class
// class Vector ;

// class Point
class Point
{
private:
    // private data members
```

```

    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // public member functions
    double& X () { return x ; }
    double& Y () { return y ; }
    double& Z () { return z ; }
    const double& X () const { return x ; }
    const double& Y () const { return y ; }
    const double& Z () const { return z ; }
    // overloaded operator
    // Point operator + (const Vector& v) ;
    // friend class
    friend class Vector ;
}; // class Point

// Vector class
class Vector
{
private:
    // private data members
    double a, b, c ;
public:
    // constructors
    Vector ()
        : a (0.0), b (0.0), c (0.0) {}
    Vector (double a_arg, double b_arg, double c_arg=0.0)
        : a (a_arg), b (b_arg), c (c_arg) {}
    // public member functions
    double& A () { return a ; }
    double& B () { return b ; }
    double& C () { return c ; }
    const double& A () const { return a ; }
    const double& B () const { return b ; }
    const double& C () const { return c ; }
    // overloaded operator
    Point operator + (const Point& p) ;
    //friend class Point ;
}; // class Vector

// v + p
Point Vector::operator + (const Point& p)
{
    return Point (a+p.x, b+p.y, c+p.z) ;
}

/*

```

```

// p + v
Point Point::operator + (const Vector& v)
{
    return Point (x+v.a, y+v.b, z+v.c) ;
}
*/
void main ()
{
    Point p (1.0, 1.0) ;
    Vector v (2.0, 2.0) ;

    Point q = v + p ;

    cout << "Point q (" << q.X () << ", " << q.Y () 
        << ", "           << q.Z () << ")" << endl ;
}

```

This program declares two classes, `Point` and `Vector`. We shall briefly describe the fundamentals of a vector shortly, but for the moment let us concentrate on **friend** classes. The declaration of **class** `Point` indicates that a `Vector` **class** is made a **friend** of `Point`:

```

class Point
{
    //...
public:
    //...
    friend class Vector ;
};

```

The declaration:

```
friend class Vector ;
```

can be placed anywhere within **class** `Point`'s declaration, in either the **private** or **public** sections. Such a declaration allows `Vector` full access to `Point`'s data members and member functions. Thus, we are able to define the overloaded `Vector::operator+()` member function as:

```

Point Vector::operator + (const Point& p)
{
    return Point (a+p.x, b+p.y, c+p.z) ;
}

```

which accesses object `p`'s **private** data members directly.

**friend** classes are strange things. We went to all the trouble of defining `Point`'s data members as **private** and providing adequate access member functions `X()`, `Y()` and `Z()`, and now a **friend class** has direct access to data members! If a **class**, such as `Point`, has implemented adequate member functions to access `Point`'s data members, there is simply no need for a **friend class**. The above overloaded `+ operator` member function could easily have been written as:

---

```
Point Vector::operator + (const Point& p)
{
    return Point (a+p.X (), b+p.Y (), c+p.Z ());
}
```

eliminating the need to make **class** Vector a **friend** of **class** Point.

The above program, FR\_CLASS.CPP, also includes (commented out) a **friend class** declaration within Vector's declaration, which makes Point a **friend** of Vector:

```
class Vector
{
//...
public:
//...
    friend class Point ;
};
```

Similarly, an overloaded Point::operator+() member function definition is commented out which has direct access to Vector's **private** data members. Note that if a **class** is forward declared it is possible to declare a **friend class** without the keyword **class**:

```
// forward declare
class Vector ;

class Point
{
//...
public:
//...
    friend Vector ; // no class keyword required
};
```

It is worth mentioning that **class** friendship is not transmitted through different classes:

```
// fr_tran.cpp
// illustrates that friendship is not transitive
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    double x, y, z ;
public:
    // friend class
    friend class Vector ;
};

// Vector class
class Vector
{
```

```

private:
    double a, b, c ;
public:
    void PointFunction (Point& p)
    { p.x = p.y = p.z = 1.0 ; } // o.k.: accessible
    // friend class
    friend class Matrix ;
};

// Matrix class
class Matrix
{
private:
    Vector array[3] ; // array of Vector's
public:
    void VectorFunction (Vector& v)
    { v.a = v.b = v.c = 1.0 ; } // o.k.: accessible
    void PointFunction (Point& p)
    { p.x = p.y = p.z = 1.0 ; } // error: not accessible
};

void main ()
{
}

```

The above program illustrates three classes: Point, Vector and Matrix. Vector is a **friend** of Point and Matrix is a **friend** of Vector, and consequently **class** Vector has access to Point's **private** x, y and z data members and **class** Matrix has access to Vector's **private** a, b and c data members. However, just because Vector is a **friend** of Point and Matrix is a **friend** of Vector does not mean that Matrix has access to Point's **private** data members.

In addition note that friendship is not inherited, as we shall see in Chapter 15.

### 11.2.1 Vectors

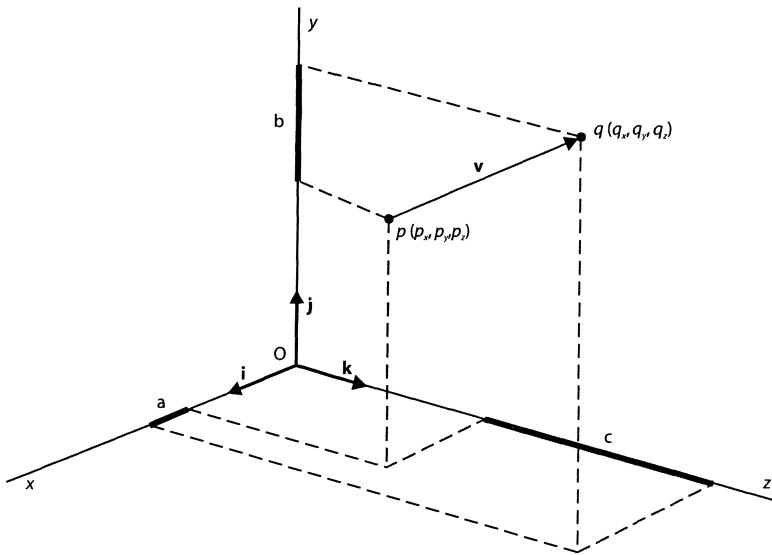
Let us now return to vectors. Figure 11.1 illustrates two points  $p$  and  $q$  in a three-dimensional Cartesian coordinate system with origin O. Consider a vector  $\mathbf{v}$  with initial point  $p$  and terminal point  $q$ . Let  $(p_x, p_y, p_z)$  and  $(q_x, q_y, q_z)$  be the coordinates of points  $p$  and  $q$  respectively. The scalar components of vector  $\mathbf{v}(a, b, c)$  are then defined as:

$$a = q_x - p_x, \quad b = q_y - p_y, \quad c = q_z - p_z$$

Vectors are frequently written in terms of the triple orthogonal unit vectors  $\mathbf{i}, \mathbf{j}$  and  $\mathbf{k}$ ; refer to Fig. 11.1. The vectors  $\mathbf{i}, \mathbf{j}$  and  $\mathbf{k}$  each have positive directions in line with the positive directions of the coordinate axes. By making use of the unit vectors, vector  $\mathbf{v}$  can be written:

$$\mathbf{v} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$$

The declaration of **class** Vector has three data members, a, b and c, which represent the three components of a vector:



**Fig. 11.1** A vector  $\mathbf{v}$  in a three-dimensional Cartesian coordinate system with an initial point  $p$  and terminal point  $q$ . Also shown are the orthogonal unit vectors  $\mathbf{i}$ ,  $\mathbf{j}$  and  $\mathbf{k}$ .

```
class Vector
{
private:
    double a, b, c ;
//...
};
```

The form of **Vector** is similar to that of **Point**, except that the **a**, **b** and **c** data members represent the components of a vector, whereas the **x**, **y** and **z** data members of **class Point** represent a point.

## 11.3 Friends and Overloaded Operators

**friends** can be used in conjunction with overloading an operator. Let's overload the addition operator **+** for **class Point** using a **friend** function:

```
// fr_op.cpp
// illustrates operator overloading
// using friend functions
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
```

```

double x, y, z ;
public:
// constructors
Point ()
: x (0.0), y (0.0), z (0.0) {}
Point (double x_arg, double y_arg, double z_arg=0.0)
: x (x_arg), y (y_arg), z (z_arg) {}
// public member functions
//...
// friend
friend Point operator + (const Point& p1,
                           const Point& p2) ;
}; // class Point

// Point friend function (p + p)
inline Point operator + (const Point& p1, const Point& p2)
{
return Point (p1.x+p2.x, p1.y+p2.y, p1.z+p2.z) ;
}

void main ()
{
    Point p (1.0, 1.0), q (-1.0, -1.0) ;

    Point r = p + q ;

    cout << "r (" << r.X () << ", " << r.Y ()
         << ", " << r.Z () << ")" << endl ;
}

```

Note that **friend** functions can also be defined as **inline** and that no scope resolution operator (**:**) is required for **friend** function declarators. Since a **friend** function does not operate implicitly on an object, both **Point** operands of a typical expression involving the binary operator **+** must be explicitly passed to the **friend** function. The overloaded **+** operator is declared a **friend of class** **Point** by the following declaration within the **Point class** declaration:

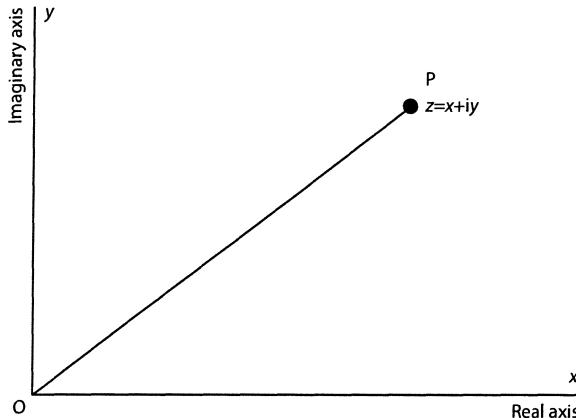
```
friend Point operator + (const Point& p1, const Point& p2) ;
```

Thus little seems to have been gained by using a **friend** overloaded operator. However, there is an important special case in operator overloading which makes **friends** very useful. To illustrate this special case, let us consider a complex number **class**, **Complex**. Firstly, let us have a brief reminder of what a complex number is.

### 11.3.1 Complex Numbers

There are equations which cannot be solved purely in terms of real numbers:

$$x^2 + 2 = 0$$



**Fig. 11.2** The complex plane.

A complex number,  $z$ , is a pair of real numbers  $(x, y)$  in which  $x$  is the *real part of  $z$*  and  $y$  is the *imaginary part of  $z$* :  $\text{Re } z=x$  and  $\text{Im } z=y$ . Complex numbers are generally represented in terms of the *imaginary unit*,  $i$  ( $0, 1$ ):

$$z = x + iy$$

An important characteristic of  $i$  is that  $i^2=-1$ :  $((0,1)(0,1)=(-1,0)=-1)$ .

Complex numbers are frequently represented as points in the *complex plane* or *Argand*<sup>1</sup> diagram. Figure 11.2 illustrates the complex plane, in which the  $x$ -axis is called the *real axis* and the  $y$ -axis is the *imaginary axis*.

The four basic arithmetical operations on two complex numbers,  $z_1$  ( $x_1+iy_1$ ) and  $z_2$  ( $x_2+iy_2$ ), are fairly straightforward and are given below.

*Addition:*

$$z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$$

*Subtraction:*

$$z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$$

*Multiplication:*

$$z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1)$$

*Division:*

$$\frac{z_1}{z_2} = \frac{(x_1 + iy_1)(x_2 - iy_2)}{(x_2 + iy_2)(x_2 - iy_2)} = \frac{x_1 x_2 + y_1 y_2}{x_2^2 + y_2^2} + i \frac{x_2 y_1 - x_1 y_2}{x_2^2 + y_2^2}$$

<sup>1</sup> Jean Robert Argand (1768–1822)

The *conjugate* of a complex number  $z (=x+iy)$  is denoted by  $\bar{z} (=x-iy)$ .

### 11.3.2 Complex class

From the above discussion, a complex number **class** Complex should encapsulate both real and imaginary numbers. Therefore, the implementation of **class** Complex below contains two data members, **re** and **im**, which represent the real and imaginary parts of a complex number:

```
// fr_cplx0.cpp
// illustrates overloaded operators using
// friend functions via a Complex class
#include <iostream.h> // C++ I/O

// Complex number class
class Complex
{
private:
    double re, im ;
public:
    // constructors
    Complex ()
        : re (0.0), im (0.0) {}
    Complex (double r, double i)
        : re (r), im (i) {}
    // access member functions
    double& Real () ;
    double& Imag () ;
    const double& Real () const ;
    const double& Imag () const ;
    // overloaded operators
    Complex operator + (const Complex& c) ;
    Complex operator + (double d) ;
    // overloaded operator/friend
    friend Complex operator + (double d, const Complex& c) ;
};

// access member functions:

// non-const objects
inline double& Complex::Real ()
{
    return re ;
}

inline double& Complex::Imag ()
{
    return im ;
}
```

```

// const objects
inline const double& Complex::Real () const
{
    return re ;
}

inline const double& Complex::Imag () const
{
    return im ;
}

// overloaded operators:

// (a+bi)+(c+di)=(a+c)+(b+d)i
inline Complex Complex::operator + (const Complex& c)
{
    return Complex (re + c.re, im + c.im) ;
}

// (a+bi)+d=(a+d)+(b)i
inline Complex Complex::operator + (double d)
{
    return Complex (re + d, im) ;
}

// overloaded operators/friends:

// d+(a+bi)=(a+d)+(b)i
inline Complex operator + (double d, const Complex& c)
{
    return Complex (d + c.re, c.im) ;
}

void main ()
{
    Complex c1 (1.0, 2.0), c2 (3.0, 4.0) ; // define & initialise

    Complex c3 = c1 + c2 ;
    Complex c4 = c1 + 10.0 ;
    Complex c5 = 10.0 + c1 ;

    cout << "c3 (" << c3.Real () << ", "
          << c3.Imag () << ")" << endl ;
    cout << "c4 (" << c4.Real () << ", "
          << c4.Imag () << ")" << endl ;
    cout << "c5 (" << c5.Real () << ", "
          << c5.Imag () << ")" << endl ;
}

```

The default and one-argument constructors both initialise the `re` and `im` data members of `Complex`. Two access member functions, `Real()` and `Imag()`, are defined which return, by

reference, the `re` and `im` data members for non-constant and constant objects, respectively. The last three functions in the declaration of `class Complex` are functions that overload the `+` operator. The first of the three overloaded `+` operator functions is:

```
Complex operator + (const Complex& c) ;
```

which allows a user of `class Complex` to add together two `Complex` objects:

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

When an overloaded operator is used in an expression, the object on the left-hand side of the operator must be a member of the `class` that overloads the operator. Therefore the above expression is valid, since object `c1` is an object of `class Complex` and object `c2` is passed as an argument to `Complex::operator+()`. The next overloaded operator function is:

```
Complex operator + (double d) ;
```

which allows a user of `Complex` to add a `Complex` object and a `double` floating-point number:

```
Complex c4 = c1 + 10.0 ;
```

Again, the operand on the left-hand side of the operator is an object of `class Complex`. However, what happens if a user attempts the following expression?

```
Complex c5 = 10.0 + c1 ;
```

The object on the left-hand side of the `+` operator is now of type `double`, not `Complex`. Type `double` was not defined with an operation involving the addition of a `double` and a `Complex`, and thus a compilation error will result. If a `friend` function is used in conjunction with the overloaded `+` operator and passed two arguments (instead of just the one, as above), a solution is found to the problem of adding a `double` and a `Complex`:

```
friend Complex operator + (double d, const Complex& c) ;
```

In this case, both operands are passed as arguments to the `friend` overloaded `+` operator function.

Occasionally in the literature and examination of the COMPLEX.H header file supplied with commercial C++ compilers you will see that the above three `operator+()` functions are declared as:

```
friend Complex operator + (const Complex& c1,
                           const Complex& c2) ;
friend Complex operator + (const Complex& c, double d) ;
friend Complex operator + (double d, const Complex& c) ;
```

It is unnecessary to declare all three functions as `friend` functions, since the first two functions implicitly act on a `Complex` object.

### 11.3.3 The Use of Conversion Functions

In the previous section, we had to explicitly define an overloaded operator function for the three cases `Complex+Complex`; `Complex+double` and `double+Complex`. Thus, for the four arithmetic operators (+, -, \* and /) we would need 12 overloaded operator functions to perform basic arithmetical operations on `Complex` and `double` objects. It was noted in Chapter 10 that in C++ it is possible to define conversion member functions which enable conversion from one specified type or `class` to another. A conversion function which converts from `double` to `Complex` could be implemented as a one-argument constructor:

```
Complex::Complex (double d)
{ re = d ; im = 0.0 ; }
```

This conversion function simply assigns the `double` argument, `d`, to the `re` data member of `class Complex` and sets the `im` part to zero.

The conversion function now allows us to define `class Complex` with a single `friend` overloaded + operator function:

```
class Complex
{
//...
public:
    // constructors
    Complex () ;
    Complex (double r, double i) ;
    // conversion func. (1 arg. constructor)
    Complex (double d) { re=d; im=0.0 ; }
    //...
    // overloaded operator/friend
    friend Complex operator + (const Complex& c1,
                               const Complex& c2) ;
};
```

However, we can use a little trick in the case of the `Complex` `class`. At present, the conversion function is merely a one-argument constructor which assigns the constructor argument to the `re` data member and sets the `im` data member to zero. Thus, by assigning a default value to `im` in the two-argument constructor we are able to eliminate the one-argument constructor. This is illustrated in the program below, which also defines the overloaded operators -, \*, and /:

```
// fr_cplx1.cpp
// illustrates overloaded operators using friend functions
// via a Complex class and a conversion function
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()

const double TOLERANCE = 1e-06 ;

// Complex number class
class Complex
{
private:
```

```

    double re, im ;
public:
    // constructors
    Complex () ;           // default argument
    Complex (double r, double i=0.0) ;
    // access member functions
    double& Real () { return re ; }
    double& Imag () { return im ; }
    const double& Real () const { return re ; }
    const double& Imag () const { return im ; }
    // overloaded operators/friends
    friend Complex operator + (const Complex& c1,
                                const Complex& c2) ;
    friend Complex operator - (const Complex& c1,
                                const Complex& c2) ;
    friend Complex operator * (const Complex& c1,
                                const Complex& c2) ;
    friend Complex operator / (const Complex& c1,
                                const Complex& c2) ;
};

// constructors:

// no. arg. constructor:
// initialise Re & Im components of Complex no. to 0
inline Complex::Complex ()
{
    re = 0.0 ;
    im = 0.0 ;
}

// 1 arg. constructor: set Re & Im components
inline Complex::Complex (double r, double i)
{
    re = r ;
    im = i ;
}

// overloaded operators/friends:

// (a+bi)+(c+di)=(a+c)+(b+d)i
inline Complex operator + (const Complex& c1, const Complex& c2)
{
    return Complex (c1.re + c2.re, c1.im + c2.im) ;
}

// (a+bi)-(c+di)=(a-c)+(b-d)i
inline Complex operator - (const Complex& c1, const Complex& c2)
{
    return Complex (c1.re - c2.re, c1.im - c2.im) ;
}

```

```

// (a+bi) * (c+di) = (ac-bd) + (ad+bc)i
inline Complex operator * (const Complex& c1, const Complex& c2)
{
    return Complex (c1.re*c2.re - c1.im*c2.im,
                    c1.re*c2.im + c1.im*c2.re) ;
}

// (a+bi)/(c+di) = ((ac+bd)/(c^2+d^2)) + ((bc-ad)/(c^2+d^2))i
inline Complex operator / (const Complex& c1, const Complex& c2)
{
    double denom = c2.re*c2.re + c2.im*c2.im ;
    if (denom < TOLERANCE)
    {
        cout << "Complex:operator / :zero divide" << endl ;
        exit (EXIT_SUCCESS) ;
    }
    return Complex ((c1.re*c2.re + c1.im*c2.im)/denom,
                    (c1.im*c2.re - c1.re*c2.im)/denom) ;
}

void main ()
{
    Complex c1 (1.0, 2.0), c2 (3.0, 4.0) ; // define &
                                              // initialise

    Complex c3 = c1 + c2 ;
    Complex c4 = c1 - c2 ;
    Complex c5 = c1 * c2 ;
    Complex c6 = c1 / c2 ;

    cout << "c3 (" << c3.Real () << ", "
          << c3.Imag () << ")" << endl ;
    cout << "c4 (" << c4.Real () << ", "
          << c4.Imag () << ")" << endl ;
    cout << "c5 (" << c5.Real () << ", "
          << c5.Imag () << ")" << endl ;
    cout << "c6 (" << c6.Real () << ", "
          << c6.Imag () << ")" << endl ;
}

```

with output:

```

c3 (4, 6)
c4 (-2, -2)
c5 (-5, 10)
c6 (0.44, 0.08)

```

For further details of the declaration of the C++ library **class** `complex`, based on type **double**, the reader should refer to the header file `COMPLEX.H`. Note that the C++ draft standard defines the following three floating-point complex number classes for types **float**,

**double** and **long double**, respectively: **float\_complex**, **double\_complex** and **long\_double\_complex**.

## 11.4 Overloading the C++ Input Extraction (>>) and Output Insertion (<<) Operators

The previous chapter illustrated that it is possible to overload the insertion, <<, and extraction, >>, operators so that we can output the data members of an object of a user-defined **class** in a manner similar to that of C++'s integral types:

```
Complex c ;  
//...  
cout << c ;
```

However, it was noted in Chapter 10 that the insertion and extraction operator functions are non-member functions of a user-defined **class**, and it is therefore necessary to make a **class**'s data members **public** so that the insertion and extraction operator functions have access to an object's data members. This goes deeply against the grain of programming in C++ in an object-oriented fashion.

**friend** functions can be used to allow the insertion and extraction operator functions to have access to an object's data members while specifying that an object's data is kept **private**. The following program illustrates the use of **friend** overloaded insertion and extraction functions for **class** Complex:

```
// fr_io.cpp  
// illustrates overloading C++'s insertion  
// and extraction operators for class Complex  
//...  
// Complex number class  
class Complex  
{  
    private:  
        double re, im ;  
    public:  
        //...  
        // overloaded operators/friends  
        //...  
        friend ostream& operator << (ostream& s,  
                                         const Complex& c) ;  
        friend istream& operator >> (istream& s,  
                                         Complex& c) ;  
};  
//...  
// overloaded operator/friend:  
//...  
// insertion operator  
inline ostream& operator << (ostream& s, const Complex& c)  
{
```

---

```

return s << "(" << c.re << ", " << c.im << ")" << endl ;
}

// extraction operator
inline istream& operator >> (istream& s, Complex& c)
{
return s >> c.re >> c.im ;
}

void main ()
{
Complex c1 (1.0, 2.0), c2 (3.0, 4.0) ; // define & initialise

Complex c3 = c1 + c2 ;
Complex c4 = c1 - c2 ;
Complex c5 = c1 * c2 ;
Complex c6 = c1 / c2 ;

cout << "c3 " << c3 ;
cout << "c4 " << c4 ;
cout << "c5 " << c5 ;
cout << "c6 " << c6 ;
}

```

with output:

```

c3 (4, 6)
c4 (-2, -2)
c5 (-5, 10)
c6 (0.44, 0.08)

```

## 11.5 Summary

A **friend** function or **class** has access to the **private** data members of an object of a given **class**, although it is not a member of that **class**. Functions or classes are made **friends** of a given **class** by use of the **friend** keyword in a **class**'s declaration. **friends** can be very useful in certain cases, particularly when overloading binary operators in which the left-hand operand is not related to the **class** which overloaded the operator and when overloading the C++ insertion and extraction operators.

## Exercises

- 11.1 For the following complex number **class**, **Complex**, define a function *Sqrt ()* which returns the square root of a **Complex** object and is a **friend** of **Complex**:

```

class Complex
{

```

```

private:
    double re, im ;
public:
    Complex (double real, double imag=0.0)
        : re (real), im (imag) {}
    };
```

- 11.2 Provide a single overloaded + operator, using **friend**, for the String **class** of Chapter 10, STRING.CPP.
- 11.3 Extend the Tensor, SymmetricTensor and AntiSymmetricTensor classes developed in Exercise 10.3 so that the overloaded +, -, << and >> operators are overloaded as **friend** functions.
- 11.4 For the Int **class** of Exercise 9.2 provide an overloaded + operator. Overload + as member, **friend**, and non-member and non-**friend** functions of **class** Int.
- 11.5 The following code compiles without errors, but results in a run-time error. Find the error. Why does it occur?

```

class Int
{
private:
    int data ;
public:
    Int (int value)
        : data (value) {}
    //...
    friend Int operator + (const Int& i1, const Int& i2) ;
};

Int operator + (const Int& i1, const Int& i2)
{
    return Int (i1+i2) ;
}

void main ()
{
    Int i1 (1), i2 (2) ;

    Int i3 = i1 + i2 ;
}
```

- 11.6 Overload the insertion and extraction operators for **class** Point without the use of **friend** functions.
- 11.7 The following program code lists two classes called LinkedList and LinkedListIterator:

```

class LinkedList
{
private:
    class Node
    {
        //
```

```
    } * first ;
public:
    LinkedList ()
        : first (NULL) {}
    //...
};

class LinkedListIterator
{
private:
    LinkedList& list ;
    LinkedList::Node* current ;
public:
    LinkedListIterator (LinkedList& l)
        : list (l), current (l.first) {}
    //...
};
```

It is observed that `LinkedListIterator` requires direct access to `LinkedList`'s **private** data member, `first`, and nested **class** `Node`. How can this access problem be resolved using **friend** classes? Linked lists and their respective iterators will be discussed in more detail in Chapters 12 and 13.

# Pointers

*it's good to know the future's Orange*

*Pointers are a feature that C++ inherited from the C programming language. Although not specifically an object-oriented feature of C++, pointers are nevertheless extremely powerful and form a large part of the language. In several cases, such as the implementation of data structures, pointers are essential for good programming, and give a programmer more control over the manipulation of objects.*

*In contrast with all of the objects that we have seen so far, a pointer holds the memory address of another object rather than holding the object itself. A pointer is a variable that can point to any object of a specified type. The actual object pointed to by a pointer can be accessed indirectly by a process known as indirection or dereferencing. Pointers can be used in arithmetic expressions, declared as const, passed as arguments to a function and returned by a function. A pointer is not restricted to pointing to a single object, but can point to an array of objects stored contiguously in memory. It is this property of pointers that reveals their similarity to C++ static arrays. Furthermore, C++ allows a pointer to point to another pointer and even to an entire function.*

*Dynamic memory allocation is made possible with the help of pointers. We will examine both the C malloc()/free() functional and the C++ new and delete operator approaches to dynamic memory allocation. The new and delete operators allocate and deallocate, respectively, a requested block of memory from the operating system and correspondingly allow a programmer to have more control over the allocation and deallocation of memory for objects and arrays of objects. C++ allows the new and delete operators to be overloaded for user-defined classes.*

*String, Vector and Matrix classes are developed in the present chapter and serve as typical applications of pointers in the implementation of C++ data structures.*



## 12.1 A First Look at Pointers

So far, we have managed to introduce 11 chapters without discussing pointers – but they can wait no more! In theory, pointers are a beautifully simple concept, but in practice they are the root of much confusion, particularly with programmers who are new to C++. I have not met anyone yet who naturally took to pointers; hence the reason for waiting 11 chapters before

discussing them. However, without sounding too pessimistic, and bearing in mind that you are probably a very good C++ programmer by now, let's kick off our tour of pointers.

Simply stated:

*A pointer is a variable that contains the memory address of another object.*

If you remember, at all times, that (1) a pointer is a *variable* and (2) a pointer contains the memory address of *another object*, and not the object itself, the notion of a pointer will be easier to grasp. Note the emphasis placed on 'another object'. A pointer does not contain the memory address of itself.

The general syntax of a pointer definition is:

```
typeSpecifier* pointerName ;
```

This statement defines a pointer variable, `pointer_name`, which points to objects of a C++ integral or user-defined type or `class`, `typeSpecifier`. The asterisk, `*`, is the *indirection operator*, which indicates to the compiler that the defined variable is a pointer and not a *normal object*. For instance:

```
int* i_ptr ; // pointer to int
double* d_ptr ; // pointer to double
Point* p_ptr ; // pointer to Point
Complex* c_ptr ; // pointer to Complex
```

defines variables `i_ptr`, `d_ptr`, `p_ptr` and `c_ptr` to be pointers to the C++ integral types `int` and `double` and user-defined classes `Point` and `Complex` respectively. Frequently, the type that a pointer points to is referred to as the *base type*.

Let's examine a program to help clarify pointers:

```
// ptr_1st.cpp
// illustrates pointer variables
#include <iostream.h> // C++ I/O

void main ()
{
    int i = 0 ; // integer variables i & j
    int j = 9 ;

    int* i_ptr ; // pointer variables i_ptr & j_ptr
    int* j_ptr ;

    // O/P
    cout << "i: " << i << endl ;
    cout << "j: " << j << endl ;

    cout << "i_ptr: " << i_ptr << endl ;
    cout << "j_ptr: " << j_ptr << endl ;
}
```

with output:

```
i: 0
j: 9
i_ptr: 0x1f5f
j_ptr: 0xffff
```

The `main()` function begins by defining and initialising two `int` variables `i` and `j`:

```
int i = 0 ;    // integer variables i & j
int j = 9 ;
```

which, by now, should be familiar. Next, two pointer variables `i_ptr` and `j_ptr` to the base type `int` are defined in `main()`:

```
int* i_ptr ;    // pointer variables i_ptr & j_ptr
int* j_ptr ;
```

The asterisk in the definition indicates to the compiler that `i_ptr` and `j_ptr` are not normal variables of type `int` but pointers to type `int`. So what's the difference? Examining the program output we find that the variables `i` and `j` are displayed, as initialised, with the values 0 and 9 respectively. The pointers `i_ptr` and `j_ptr` are displayed as the memory addresses `0x1f5f` and `0xffff` respectively (these memory addresses will probably be different when you run the above program). Thus the output illustrates that the compiler treats a pointer variable as containing the memory address of a variable and not the value of a variable. To illustrate this important point further we require the use of the *address-of operator*, `&`.

## 12.2 The Address-of Operator

The address-of operator, `&`, is a unary operator that returns the memory address of the operand that it acts upon<sup>1</sup>. Consider the program:

```
// aoo.cpp
// illustrates the address-of operator &
#include <iostream.h>    // C++ I/O

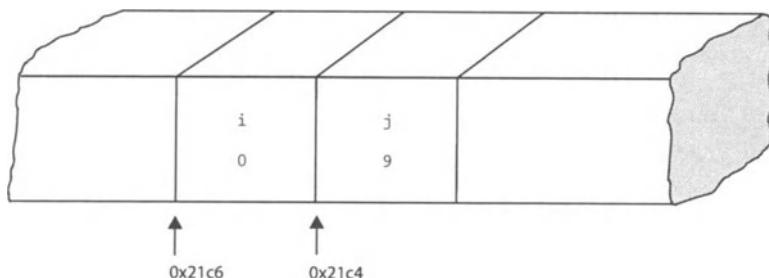
void main ()
{
    int i = 0 ;    // integer variables i & j
    int j = 9 ;
    // O/P
    cout << "i: " << i << endl ;
    cout << "j: " << j << endl ;

    cout << "&i: " << &i << endl ;    // O/P addresses of i & j
    cout << "&j: " << &j << endl ;
}
```

with output:

---

<sup>1</sup> Refer to Appendix C for a complete list of the available operators in the C++ programming language.



**Fig. 12.1** Schematic representation of a segment of the computer's memory which contains the variables *i* and *j*.

```
i: 0
j: 9
&i: 0x21c6
&j: 0x21c4
```

which illustrates that the address-of operator, `&`, accesses the memory addresses of variables *i* and *j*. Figure 12.1 schematically illustrates the memory addresses of variables *i* and *j*.

Do not confuse the address-of operator with the reference operator, which was introduced in Chapter 6 when we discussed the passing and returning of objects via function arguments by reference rather than by value. The address-of operator precedes a variable or object name, whereas the reference operator follows the variable type:

```
cout << &i ; // address-of operator
//...
void Function (int& i) ; // reference operator
//...
```

We can now combine the above two programs, PTR\_1ST.CPP and AOO.CPP, to assign the memory addresses of the variables *i* and *j* to a pointer variable, *ptr*, of base type **int**:

```
// ptr.cpp
// illustrates the pointer variables
// and the address-of operator &
#include <iostream.h> // C++ I/O

void main ()
{
    int i = 0 ; // integer variables i & j
    int j = 9 ;
    // O/P
    cout << "i: " << i << endl ;
    cout << "j: " << j << endl ;

    int* ptr ; // pointer to int
    ptr = &i ; // ptr points to i
    cout << "(ptr=&i); ptr: " << ptr
```

---

```

    << ", *ptr: " << *ptr << endl ;

ptr = &j ; // ptr points to j

cout << "(ptr=&j); ptr: " << ptr
    << ", *ptr: " << *ptr << endl ;
}

```

with output:

```

i: 0
j: 9
(ptr=&i); ptr: 0x21ee, *ptr: 0
(ptr=&j); ptr: 0x21ec, *ptr: 9

```

The following statement assigns the memory address of variable *i* to the pointer *ptr*:

```
ptr = &i ; // ptr points to i
```

This statement does not assign, by copying, the value of variable *i* to the pointer variable *ptr*, but rather assigns the memory address of *i* to *ptr*. Later in the program, the pointer *ptr* points to the memory address of variable *j*:

```
ptr = &j ; // ptr points to j
```

which illustrates that the pointer *ptr* is indeed a variable and can be assigned the memory address of any variable of type **int**.

If the value of a variable of type **int** is required rather than its memory address, the unary operator **\*** precedes the pointer:

```
cout << *ptr ; // access value and not memory address
```

*\*ptr* accesses the value held at the memory address pointed to by the operand *ptr*. Alternatively, the **int** value held at the memory address pointed to by *ptr* could be assigned to a variable or constant of type **int**:

```

int i ;
int* ptr ;
//...
ptr = &i ;
//...
int value = *ptr ;           // assign value of i to a variable
//...
int const VALUE = *ptr ;    // assign value of i to a constant

```

## 12.3 Multiple Pointer Definitions in a Single Statement

We have seen with normal variables that C++ allows us to define multiple uninitialised variables in a single statement of the form:

```
int i, j, k ;
```

Similarly, it is possible to define multiple uninitialised pointers in a single statement:

```
int* i_ptr, * j_ptr, * k_ptr ;
```

Note that this is not the same as:

```
int* i_ptr, j_ptr, k_ptr ;
```

which is a common mistake. In this case, `i_ptr` is a pointer to `int`, but `j_ptr` and `k_ptr` are normal variables of type `int` and not pointers.

## 12.4 A Note on Pointer Definitions

There are generally two styles of defining a pointer variable:

```
int* ptr1 ;  
int *ptr2 ;
```

The first attaches the asterisk to the type specifier, while the second attaches the asterisk to the variable name. Which style you adopt makes no difference to the compiler, but the style that I adopt is to attach the asterisk to the type specifier. This convention helps to clarify that the asterisk is part of the type specification and not the variable name, i.e. `int*` is a pointer to `int` in the above example.

## 12.5 Another Look at Pointers

Let's now take another look at pointers from an lvalue/rvalue perspective. Remember that an object's lvalue is its memory address, whereas the rvalue is the actual data held by the object. You may have noticed from the above discussion that the rvalue of a pointer is in fact the lvalue of a different object. Thus we can now extend our definition of a pointer:

*A pointer is a variable whose rvalue contains the lvalue or memory address of another object.*

To help illustrate this definition, let us consider another program example on pointers:

```
// ptr_next.cpp  
// another look at pointers  
#include <iostream.h> // C++ I/O  
  
void main ()  
{  
    int j = 9 ; // integer variable j
```

```

int* ptr ; // pointer to int
cout << "lvalue of j : " << &j <<
      ", rvalue of j : " << j << endl ;

cout << "lvalue of ptr: " << &ptr <<
      ", rvalue of ptr: " << ptr << endl ;

ptr = &j ; // ptr points to j

cout << "lvalue of j : " << &j <<
      ", rvalue of j : " << j << endl ;
cout << "lvalue of ptr: " << &ptr <<
      ", rvalue of ptr: " << ptr << endl ;

*ptr = 10 ; // assign 10 to rvalue of ptr (j)

cout << "lvalue of j : " << &j <<
      ", rvalue of j : " << j << endl ;
cout << "lvalue of ptr: " << &ptr <<
      ", rvalue of ptr: " << ptr << endl ;

int k = *ptr ; // assign rvalue of ptr to k

cout << "lvalue of k : " << &k <<
      ", rvalue of k : " << k << endl ;
}

```

with output:

```

lvalue of j : 0x45ff2282, rvalue of j : 9
lvalue of ptr: 0x45ff227e, rvalue of ptr: 0xfffff0b22
lvalue of j : 0x45ff2282, rvalue of j : 9
lvalue of ptr: 0x45ff227e, rvalue of ptr: 0x45ff2282
lvalue of j : 0x45ff2282, rvalue of j : 10
lvalue of ptr: 0x45ff227e, rvalue of ptr: 0x45ff2282
lvalue of k : 0x45ff2246, rvalue of k : 10

```

The program begins by defining and initialising a variable, *j*, of type **int** and a pointer variable to **int**, *ptr*:

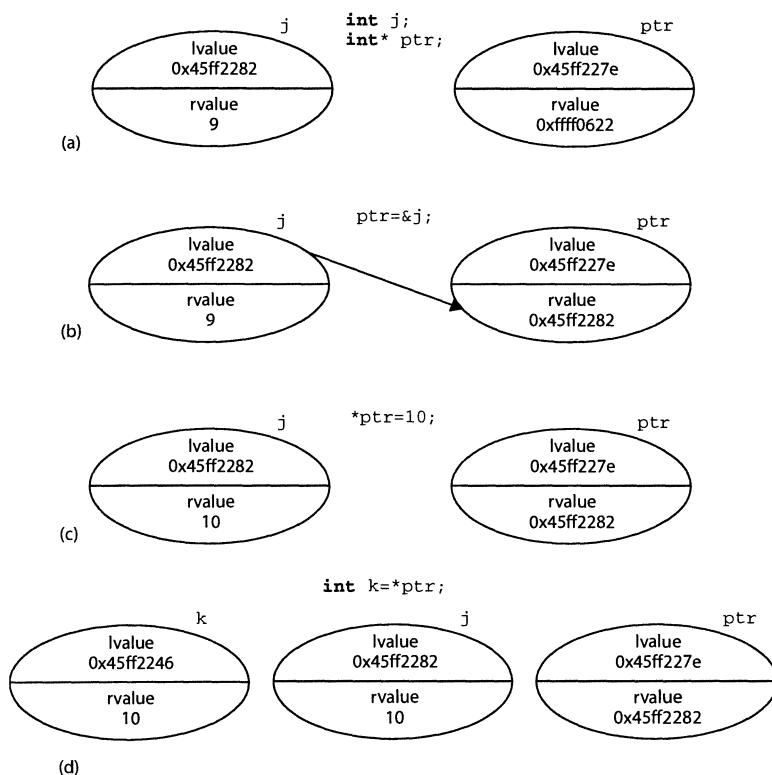
```

int j = 9 ; // integer variable j
int* ptr ; // pointer to int

```

Directly following these definitions the lvalues and rvalues of *j* and *ptr* are displayed on the screen. Both *j* and *ptr* have their own unique lvalues – remember that a pointer is a variable and has to be stored somewhere in memory! The rvalue of *j* is equal to the integer constant 9 while *ptr*'s rvalue is garbage because at this point the rvalue of *ptr* has not been assigned the memory address of an identifier of type **int**; see Fig. 12.2(a).

Next, the following statement assigns the lvalue of *j* to the rvalue of *ptr* by means of the address-of operator, &:

**Fig. 12.2** Playing with pointers.

```
ptr = &j ; // ptr points to j
```

Observe from the program output that the rvalue of *ptr* is now equal to the lvalue of *j* (Fig. 12.2(b)).

The statement:

```
*ptr = 10 ; // assign 10 to rvalue of ptr (j)
```

uses the process of indirection (i.e. indirect access) to change the rvalue of *j* via the rvalue of the pointer *ptr*. This statement alters the rvalue of *j* because the previous statement, *ptr=&j*, assigned the lvalue or memory address of *j* to the rvalue of *ptr*. Therefore, whatever happens to the rvalue of *ptr* will indirectly happen to *j* (Fig. 12.2(c)). Indirection is frequently referred to as *dereferencing*.

The final assignment statement in program PTR\_NEXT.CPP is:

```
int k = *ptr ; // assign rvalue of ptr to k
```

This statement uses indirection to assign the rvalue of *ptr*, which is the lvalue of *j*, to the rvalue of variable *k* (Fig. 12.2(d)).

Confused? Don't worry – manipulating pointers does take some getting use to.

## 12.6 Valid Pointers

Just as C++ does not guarantee the value held by an uninitialised variable, similarly C++ does not guarantee the memory address held by the rvalue of a pointer variable. So always ensure that before the rvalue of a pointer variable is accessed that it has been initialised to a valid memory address.

Beware of uninitialised pointers. The following program defines a variable `i` and a pointer `ptr`. The variable `i` is then assigned the constant value of 1. Next, pointer `ptr` is assigned the value of `i` without `ptr` first being initialised:

```
// nonvalid.cpp
// illustrates an uninitialised pointer
#include <iostream.h> // C++ I/O

void main ()
{
    int i ;
    int* ptr ;

    i = 1 ;

    *ptr = i ; // error: ptr doesn't point to i
}
```

The compiler will issue a compilation warning of the form ‘use of pointer `ptr` before definition’ for the above program. The program will compile and link correctly, but will generally generate a run-time application error. Therefore, always ensure that a pointer is defined and initialised to a valid memory address before it is used, e.g. `ptr=&i`.

## 12.7 The NULL Pointer

C++ defines<sup>2</sup> an identifier called `NULL` which is referred to as ‘the NULL pointer’. The NULL pointer is guaranteed not to point to valid data and will always be interpreted as logical-false (i.e. 0). The NULL pointer can be useful for string manipulation since the value of the null terminator character ‘\0’ is not fixed and can be machine-dependent. Therefore, try to use `NULL` or 0 instead of ‘\0’ whenever possible in your programs.

Also, it is good programming practice to initialise a pointer to the NULL pointer when a pointer is defined. This convention can eliminate the common programming mistake of accidentally using an uninitialised pointer in a test expression. For example:

```
// null.cpp
// illustrates the NULL pointer
#include <iostream.h> // C++ I/O

void main ()
```

<sup>2</sup> The majority of compilers #define the NULL pointer in the `STDIO.H` header file. The Borland C++ (version 5.0) compiler defines `NULL` in the `_NULL.H` header file.

```

{
int* ptr1 ;           // uninitialised pointer
int* ptr2 = NULL ;   // pointer initialised to NULL

if (ptr1)
    cout << "uninitialised pointer tested logical-true"
    << endl ;

if (ptr2)
    cout << "initialised pointer to NULL" << endl ;
}

```

with output:

```
uninitialised pointer tested logical-true
```

Commercial compilers are generally very good at detecting uninitialised pointers and will generally issue a compilation warning for the above program of the form ‘possible use of pointer “ptr1” before definition’.

## 12.8 Don't Mix Types

Consider the program:

```

// mix.cpp
// illustrates the mixing of types with pointers
#include <iostream.h> // C++ I/O

void main ()
{
    double d = 1.25 ;
    int* ptr ;

    ptr = &d ;           // error: can't convert a double pointer
                         // to an int pointer

    ptr = (int*) &d ;   // cast double pointer to an int pointer

    double e = *ptr ;
    cout << "e: " << e << endl ;
}
```

The statement:

```
ptr = &d ;
```

attempts to assign a **double** pointer to an **int** pointer. Such an assignment will generate a compilation error of the form ‘cannot convert a double pointer to an integer pointer’. It is possible to cast the **double** pointer into an **int** pointer:

---

```
ptr = (int*) &d ;
```

However, such a conversion<sup>3</sup> of one pointer type to another can result in very strange behaviour. Pointer `ptr` is of base type `int` and `d` is of type `double`. Thus, a `double` memory address has been assigned in to an `int` memory address. Hence the statement:

```
double e = *ptr ;
```

assigns only two bytes of data to the `double` eight-byte variable `e`. It is shown in the next section that a pointer to `void` can be defined which allows the mixing of types.

## 12.9 void Pointers

The previous section illustrated that statements of the form:

```
int* ptr = &d ;
```

are illegal, where `d` is of type `double`. For such cases C++ allows us to define a pointer to `void`:

```
void* ptr ;
```

which at first appears contradictory. A pointer to `void` allows any type of pointer to be assigned to itself. This is illustrated in the following program:

```
// ptr_void.cpp
// illustrates pointers to void

#include <iostream.h> // C++ I/O
#include <stdlib.h> // qsort()
#include <string.h> // strcmp()

// sorts two arguments
int Sort (const void* a, const void* b)
{
    return strcmp ((char*)a, (char*)b) ;
}

void main ()
{
    int i, * i_ptr ; // int variable & pointer
    double d, * d_ptr ; // double variable & pointer
    void* v_ptr ; // void pointer

    i_ptr = &i ; // o.k.: int* to int*
```

---

<sup>3</sup> It was noted in Chapter 4 that a cast of the form `(int*)&d` is a C-style cast. Unfortunately a C++-style pointer cast `int*(&d)` is not valid.

```

// i_ptr = &d ; // error: double* to int*

d_ptr = &d ;      // o.k.: double* to double*
// d_ptr = &i ; // error: int* to double*

v_ptr = &i ;      // o.k.: int* to void*
v_ptr = &d ;      // o.k.: double* to void*

// strings:
char names[][][5] = { "Jean", "John", "Jane", "Jack" } ;

qsort ((void*) names, 4 , sizeof (names[0]), Sort) ;

for (int j = 0; j < 4; j++)
    cout << names[j] << " " ;
cout<< endl ;

// integers:
int lottery_numbers[] = { 30, 3, 5, 44, 14, 22 } ;

qsort ((void*) lottery_numbers, 6 ,
       sizeof (lottery_numbers[0]), Sort) ;

for (int k = 0; k < 6; k++)
    cout << lottery_numbers[k] << " " ;
}

```

with sorted output:

```

Jack Jane Jean John
3 5 14 22 30 44

```

The statements:

```

v_ptr = &i ;      // o.k.: int* to void*
v_ptr = &d ;      // o.k.: double* to void*

```

illustrate that pointers to **int** and **double** can be assigned to a pointer to **void**.

To illustrate the usefulness of pointers to **void**, the above program also sorts an array of strings and an array of integers. The C++ library function *qsort()* is used for performing the sorting of both strings and integers. The signature of the *qsort()* function is:

```

void qsort (void* base, size_t nelem, size_t width,
           int (*fcmp) (const void*, const void*)) ;

```

*qsort()*<sup>4</sup> is declared in the header file **STDLIB.H**. Although *qsort()*'s function declarator may appear slightly confusing at the moment, it is an excellent function to illustrate the usefulness of pointers to **void**.

---

<sup>4</sup> Refer to your compiler's documentation for a more complete description of the *qsort()* function.

The `qsort()` function sorts an array pointed to by the `base` argument using the *median of three* variant of the quicksort algorithm for sorting. When `qsort()` returns, the sorted array is the array pointed to by `base`. The argument `nElem` is the number of elements in the array, with each element of the array occupying `width` number of bytes. The `fcmp` argument is a pointer to a user-defined comparison function which is called by `qsort()` to sort the elements of the array. In the above program the comparison function is called `Sort()` and defined as:

```
int Sort (const void* a, const void* b)
{
    return strcmp ((char*)a, (char*)b) ;
}
```

Note the casting performed on the arguments `a` and `b` of the `Sort()` function from `void*` to `char*`, which is necessary because the function declaration of the string compare function, `strcmp()`, is:

```
int strcmp (const char* s1, const char* s2) ;
```

Both function arguments, `a` and `b`, of `Sort()` are of type `const void*`, which allows us to define a single `Sort()` function which will deal with a variety of argument types. This eliminates the necessity of defining a comparison function for each possible type:

```
int SortChar (const char* a, const char* b)
//...
int SortInt (const int* a, const int* b)
//...
int SortDouble (const double* a, const double* b)
//...
```

In addition, the first argument to the `qsort()` function is a pointer to `void` which similarly allows the function to sort a variety of types. The function call:

```
qsort ((void*) names, 4 , sizeof (names[0]), Sort) ;
```

sorts an array, `names`, of characters, whereas the function call:

```
qsort ((void*) lottery_numbers, 6 ,
       sizeof (lottery_numbers[0]), Sort) ;
```

sorts an array, `lottery_numbers`, of integers.

The use of pointers as function arguments is discussed in more detail shortly.

## 12.10 `const` Pointers

The previous section illustrated the use of the `const` modifier with pointers. When dealing with pointers and the `const` modifier it is possible to specify whether the pointer is `const`, the object that it points to is `const` or both. The following program illustrates the use of `const` with pointers:

```
// const.cpp
```

```
// illustrates the use of const with pointers
#include <iostream.h> // C++ I/O

void main ()
{
    int i = 1, j = 2, k = 3, l = 4 ;

    int* ncp_nco ;           // non-const pointer, non-const object

    const int* ncp_co ;      // non-const pointer, const object

    int* const cp_nco = &i ; // const pointer, non-const object

    const int* const cp_co = &j ; // const pointer, const object

    // initialise uninitialised non-const pointers
    ncp_nco = &k ;
    ncp_co = &l ;

    *ncp_nco = 10 ;
    *ncp_co = 20 ; // error: const object
    *cp_nco = 30 ;
    *cp_co = 40 ; // error: const object

    ncp_nco++ ;
    ncp_co++ ;
    cp_nco++ ; // error: const pointer
    cp_co++ ; // error: const pointer
}
```

The following statement defines a **const** object but not a **const** pointer:

```
const int* ncp_co ; // pointer to const object
```

whereas:

```
int* const cp_nco = &i ; // const pointer to object
```

defines a **const** pointer and not a **const** object. Thus, the rule is that if the **const** modifier appears on the left-hand side of the asterisk, \*, then the object is constant; else if **const** appears on the right-hand side of \* then the pointer is constant. Note that a **const** pointer must be initialised when defined.

## 12.11 Pointers and Function Arguments

When functions were discussed in Chapter 6 it was noted that there are three ways in which an argument can be passed to a function in C++: by value, by reference or by pointer. This section will now discuss the passing of an argument to a function by pointer.

With the exception of arrays, objects are (by default) passed as arguments to a function by value. This guarantees that the function cannot alter in any way the object in the calling function, because when an object is passed by value the function automatically makes its own copy of the object. However, there are occasions when it is required that the original object in the calling function is modified. In such situations Chapter 6 showed that a function argument can be passed by reference and consequently acts as an alias for the calling function object. In principle, the passing of function arguments by pointer or by reference are similar (with some minor syntactical differences) in that they both allow data with local scope to a function to be altered. The following program (based on Kernighan and Ritchie (1978, pp. 91–3)) compares the three methods of passing function arguments:

```
// func_arg.cpp
// illustrates passing function arguments by pointer
#include <iostream.h> // C++ I/O

// pass by value
void ValueSwap (int x, int y)
{
    int temp ; // temporary variable to hold x

    temp = x ;
    x    = y ;
    y    = temp ;
}

// pass by reference
void ReferenceSwap (int& x, int& y)
{
    int temp ; // temporary variable to hold x

    temp = x ;
    x    = y ;
    y    = temp ;
}

// pass by pointer
void PointerSwap (int* ptr_x, int* ptr_y)
{
    int temp ; // temporary variable to hold x

    temp   = *ptr_x ;
    *ptr_x = *ptr_y ;
    *ptr_y = temp ;
}

void main ()
{
    int a = 0, b = 1 ;
    cout << "a: " << a << ", b: " << b << endl ;

    ValueSwap (a, b) ;
```

```

cout << "ValueSwap ():      a: " << a << ", b: " << b << endl ;
ReferenceSwap (a, b) ;
cout << "ReferenceSwap (): a: " << a << ", b: " << b << endl ;

PointerSwap (&a, &b) ;
cout << "PointerSwap ():   a: " << a << ", b: " << b << endl ;
}

```

with output:

```

a: 0, b: 1
ValueSwap ():      a: 0, b: 1
ReferenceSwap (): a: 1, b: 0
PointerSwap ():   a: 0, b: 1

```

The first of the three functions passes two arguments by value:

```
void ValueSwap (int x, int y) ;
```

and the program output illustrates that this function does not alter the integers a and b from the *main()* calling function. The next function passes two arguments by reference:

```
void ReferenceSwap (int& x, int& y) ;
```

The ampersand indicates to the compiler that x and y are aliases for a and b in the *main()* calling function:

```
ReferenceSwap (a, b) ;
```

The last of the three functions passes pointers to the two integers to be swapped:

```
void PointerSwap (int* ptr_x, int* ptr_y)
{
    int temp ; // temporary variable to hold x

    temp = *ptr_x ;
    *ptr_x = *ptr_y ;
    *ptr_y = temp ;
}
```

Since the *PointerSwap()* function is passed the memory address of two integers it must dereference the arguments to access their values. The statement:

```
temp = *ptr_x ;
```

assigns the integer value held at the memory address pointed to by *ptr\_x* to the temporary variable *temp*, and similarly for the other two function body assignment statements.

*PointerSwap()* is called in *main()* as:

```
PointerSwap (&a, &b) ;
```

Notice the use of the address-of operator, `&`, since `PointerSwap()` requires that a variable's memory address is passed, not its value.

The key difference between passing by reference and passing by pointer is that a reference is an alias for the calling function variable, whereas a pointer is the actual memory address of the calling function variable. Comparing the two definitions and calls of the functions `ReferenceSwap()` and `PointerSwap()` illustrates two main advantages of passing function arguments by reference rather than by pointer: (1) the function body statements do not require the use of the indirection operator and (2) a user of the function is relieved of the burden of having to provide the function with the memory address of a variable explicitly.

## 12.12 Pointer Arithmetic

Just as with normal variables, pointers can be used in expressions, provided that certain restrictions are observed. Pointer arithmetic is restricted to the addition (+), subtraction (-), increment (++) and decrement (--) operators. Let us first consider the addition operator.

### 12.12.1 Pointer Addition

The following program illustrates pointer addition:

```
// p_add.cpp
// illustrates pointer addition
#include <iostream.h> // C++ I/O

const int STR_LEN = 80 ;

void main ()
{
    char string[STR_LEN] ;

    cout << "enter a string: " ;
    cin.get (string, STR_LEN) ;

    int position = 0 ;
    char* p_str = string ;

    cout << "enter a position from the start of the string: " ;
    cin >> position ;

    if (position >= 0 && position < 80)
        cout << "the character at position " << position
        << " is: " << *(p_str + position) ;
    else
        cout << "position " << position
        << " is out of range-try again! " ;
}
```

Some user interaction is:

```
enter a string: while you were out
enter a position from the start of the string: 6
the character at position 6 is: y
```

The statement:

```
char* p_str = string ;
```

defines a pointer, `p_str`, to `char` and simultaneously assigns the memory address of a user-entered string to the rvalue of the pointer `p_str`. The statement:

```
* (p_str + position) ;
```

accesses the character held at the memory location (`p_str+position`).

Beware of making the common mistake:

```
*p_str + position ;
```

which would return the value 125 for the string ‘while you were out’ with `position` equal to 6 because the ASCII decimal equivalent of the letter ‘w’ of the word ‘while’ is the integer 119.

The above program illustrates adding an integer, or an *offset*, to a pointer. It is not possible to add pointers in C++. For example, try adding the following two pointers;

```
* (p_str + string) ;
```

This statement will not compile and will issue a compilation warning of the form ‘invalid pointer addition’. It makes no sense to add two pointers together, and only an offset can be added to a pointer.

### 12.12.2 Pointer Subtraction

We saw in the last subsection that pointers cannot be added. Provided that two pointers are of the same base type, one pointer can be subtracted from another pointer. The difference is simply the scalar number of elements of the pointer’s base type. To help illustrate this, consider the program:

```
// p_sub.cpp
// illustrates pointer subtraction
#include <iostream.h> // C++ I/O

// returns the length of a string
int StrLength (char* string)
{
    char* p_str = string ;

    while (*p_str)
        p_str++ ;
    return p_str - string ;
}

void main ()
```

---

```

{
char string[] = "while you were out" ;

cout << "length of \"while you were out\" is: "
     << StrLength (string) ;
}

```

Within the body of the *StrLength()* function the pointer *p\_str* is initialised to the start of the string array. While *p\_str* does not point to the NULL pointer, the **while**-loop continues to increment the pointer *p\_str* to the next character in the string. When the end of the string is reached and the **while**-loop terminates, the length of the string is obtained by subtracting the memory address of the first character of the string from the memory address of the last character in the string. The use of the increment operator with the *p\_str* pointer will become clearer from the discussion in the next section.

### 12.12.3 Increment and Decrement Operators and Pointers

The program STR\_ARG.CPP of Chapter 7 examined two functions: *StrCompare()*, which compares two strings, and *StrCopy()* which copies one string into another string. Both functions were passed two arrays of characters using array notation. Let's now modify these functions in terms of pointer notation and simultaneously illustrate the use of the postfix increment operator with pointers:

```

// p_inc.cpp
// illustrates the increment operator and pointers
#include <iostream.h> // C++ I/O

// compares two strings
// returns 1 if str1==str2, 0 if str1!=str2
int StrCompare (const char* str1, const char* str2)
{
    while (*str1++ == *str2++)
    {
        if (*str1 == NULL)
            return 1 ;
    }
    return 0 ;
}

// copies str2 into str1
void StrCopy (char* str1, const char* str2)
{
    while (*str1++ = *str2++)
    ;
}

void main ()
{
    char string1[] = "this is a string" ;
    char string2[] = "this is a string" ;
    char string3[] = "another string" ;
}

```

```

// compare string1 and string2
if (StrCompare (string1, string2) == 1)
    cout << "string1 == string2" << endl ;
else
    cout << "string1 != string2" << endl ;

// compare string2 and string3
if (StrCompare (string2, string3) == 1)
    cout << "string2 == string3" << endl ;
else
    cout << "string2 != string3" << endl ;

// copy string3 to string1
StrCopy (string1, string3) ;

cout << endl
    << "string1: " << string1 << endl ;
}

```

*StrCompare()* performs the comparison of two strings by comparing each character of the two strings that have the same array index:

```

int StrCompare (const char* str1, const char* str2)
{
    while (*str1++ == *str2++)
    {
        if (*str1 == NULL)
            return 1 ;
    }
    return 0 ;
}

```

Since the function argument **str1** is a pointer to **char**, **str1++** advances **str1** to the next character. **\*str1++** is the actual character pointed to before **str1** is incremented, since the postfix operator is used. The test condition of the **while**-loop tests each character with the same array index of both strings and then proceeds to increment to the next character. Within the body of the **while**-loop each character of the string pointed to by **str1** is tested against the **NULL** pointer.

How is the expression **\*str1++** to be interpreted? It could be read as **\*(str1++)** or **(\*str1)++**. As always, to resolve this issue we need to refer to the precedence and associativity of the operators **\*** and **++**. Appendix C informs us that the binary operators **\*** and **++** both have the same high precedence level of 15 and are right-associative. Hence the expression **\*str1++** is interpreted as **\*(str1++)** with **str1** incremented first, followed by pointer dereferencing.

*StrCopy()*<sup>5</sup> copies one string, character by character and including the null terminator, into another string:

```

void StrCopy (char* str1, const char* str2)

```

---

<sup>5</sup> This function definition may issue a compilation warning ‘possible incorrect assignment’.

---

```
{
while (*str1++ = *str2++)
;
}
```

Note that we could equally have written the **while**-loop as:

```
while ((*str1++ = str2++) != NULL)
```

The **while**-loop tests whether the condition within the inner parentheses is not equal to the NULL pointer. However, since the NULL pointer is equivalent to the ASCII character 0 (logical-false in C++), the explicit test using the relational **!=** operator is unnecessary.

It is worth mentioning that the `str1` pointer to **char** of function *StrCopy()* cannot be **const** modified because the string that `str1` points to is altered, whereas the string that `str2` points to is not altered.

The increment and decrement operators, when applied to a pointer, increase or decrease the pointer by a scalar according to what the pointer points to. For instance, the following illustrates incrementing **int**, **float** and **double** pointers and decrementing an **int** pointer:

```
int* i_ptr ; float* f_ptr ; double* d_ptr ;
//...
i_ptr++ ; // point to next int
f_ptr++ ; // point to next float
d_ptr++ ; // point to next double
//...
i_ptr-- ; // point to previous int
```

Each statement increments or decrements the rvalue (memory address) of its respective pointer by the scalar value of the pointer's base type.

It is also possible to use the prefix increment and decrement operators with pointers. For example:

```
int* ptr ;
//...
++ptr ; // increment ptr then access memory address
*++ptr ; // increment ptr then access value
--ptr ; // decrement ptr then access memory address
*--ptr ; // decrement ptr then access value
```

## 12.13 Pointers and Relational Operators

The previous section illustrated that relational operators and pointers can be used in conjunction in expressions. Pointers can be compared using the **<**, **<=**, **>**, **>=**, **==** and **!=** relational operators, but be sure that you know what you mean! If pointers point to different unrelated arrays or objects, then a pointer comparison will generally be meaningless. If, however, pointers are related somehow pointer comparisons can be useful. For example, we saw in the *StrCompare()* function of the P\_INC.CPP program the combined use of the **==** equality operator with pointers to **char**:

---

```

int StrCompare (const char* str1, const char* str2)
{
    while (*str1++ == *str2++)
    {
        if (*str1 == NULL)
            return 1 ;
    }
    return 0 ;
}

```

## 12.14 Pointers are Quicker

Aside from the obvious benefits, pointers can offer a more efficient way of manipulating data. For example, compare the *StrCompare()* function discussed above for both array indexing and pointer notation. Examining the array notation version first:

```

int StrCompare (char str1[], char str2[])
{
    int i = 0 ;
    while (str1[i] == str2[i])
    {
        if (str1[i++] == NULL)
        //...
    }
}

```

we observe that a variable *i* is required to index through the two string arrays. Considering only the *str1* character array, the lvalue of *str1[0]* must be loaded, then the value of the variable *i* must be loaded, then the value of *i* must be added to the lvalue of *str1[0]* followed by the equality comparison of *str1[i]* and *str2[i]*, and so on.

The relevant code of the pointer version of *StrCompare()* is:

```

int StrCompare (const char* str1, const char* str2)
{
    while (*str1++ == *str2++)
    //...
}

```

which illustrates that the use of the index variable, *i*, and hence the subscript operator, [ ], are eliminated.

## 12.15 Pointers and Arrays

When arrays were discussed in Chapter 7, the similarity between arrays and pointers was highlighted. This section now discusses in detail the close relationship between arrays and pointers by first examining one-dimensional arrays.

### 12.15.1 One-Dimensional Arrays

Consider the following program:

```
// p&array.cpp
// illustrates pointers and arrays
#include <iostream.h> // C++ I/O

void main ()
{
    const int SIZE = 5 ;

    int array[SIZE] = { 1, 2, 3, 4, 5 } ;

    // array notation
    cout << "array[i] : " ;
    for (int i=0; i<SIZE; i++)
        cout << array[i] << " " ;
    cout << endl ;

    // pointer notation
    cout << "*array+j: " ;
    for (int j=0; j<SIZE; j++)
        cout << *(array + j) << " " ;
}
```

with output:

```
array[i] : 1 2 3 4 5
*(array+i): 1 2 3 4 5
```

Within `main()`, an array of `SIZE` elements, each of type `int`, is defined and initialised:

```
int array[SIZE] = { 1, 2, 3, 4, 5 } ;
```

The elements of the array are stored consecutively in memory and are indexed using array notation via the subscript operator, `[]`, as `array[0]`, `array[1]`, ..., `array[SIZE-1]`.

Alternatively, and equivalently, the elements of the array can be accessed using pointer notation with the expression:

```
* (array + i)
```

which illustrates that the expression `array[i]` of an arbitrary array `array` is directly equivalent to `* (array+i)`, where `i` is an array index variable. The expression `(array+i)` is the memory address of the `i`th integer element in the array from the start of the array, whereas `* (array+i)` is the value held at that memory address. Note that it is not possible to use the increment operator in conjunction with the array name, `*array++`, although the elements of the array are accessed consecutively. To see why such an expression is not possible, let us first of all remind ourselves that an array name is in fact a synonym for the memory address of the array's zeroth element.

If `ptr` is a pointer to type `int` (i.e. `int* ptr`), then the statement:

```
ptr = &array[0] ;
```

assigns the memory address, using the address-of operator, of the first element of array (`array[0]`) to the rvalue of pointer `ptr`. However, since the name of an array is a synonym for the memory address of the first or zeroth element of the array, the previous assignment is equivalent to:

```
ptr = array ;
```

Therefore, returning to our expression `*array++`, we now see that such an expression is illegal because the name of an array is a constant, and it is not possible to increment a constant. The name of an array is constant and not a variable simply because the memory address of the array has to be placed somewhere in memory, and if this memory position were to change continually we would be unable to access the array elements<sup>6</sup>. Also, the assignments `array=ptr` and `ptr=&array` are illegal.

However, it is possible to increment a pointer: `ptr++`. The following program illustrates accessing the elements of an array using the increment operator in conjunction with a pointer:

```
// p_inc1.cpp
// illustrates incrementing a pointer
#include <iostream.h> // C++ I/O

void main ()
{
    const int SIZE = 5 ;

    int array[SIZE] = { 1, 2, 3, 4, 5 } ;

    // define & initialise a pointer, ptr, to point to array
    int* ptr = array ;

    for (int i=0; i<SIZE; i++)
        cout << *(ptr++) << " " ;
}
```

## 12.15.2 Two-Dimensional Arrays

Let us now examine two-dimensional arrays:

```
// p_a_2d.cpp
// illustrates pointers and two-dimensional arrays
#include <iostream.h> // C++ I/O

void main ()
{
    const int ROW = 2, COL = 3 ;
```

---

<sup>6</sup> A multitasking operating system, such as Microsoft Windows or IBM's OS/2, allows variable or moveable memory addresses of arrays and objects, but such a process is generally beyond our control.

```

int array[ROW][COL] = { {1, 2, 3}, {4, 5, 6} } ;

// array notation
cout << "array[i][j]: " << endl ;
for (int i=0; i<ROW; i++)
{
    for (int j=0; j<COL; j++)
        cout << array[i][j] << " " ;
    cout << endl ;
}

// pointer notation
cout << *(*(array+i)+j): " << endl ;
for (int k=0; k<ROW; k++)
{
    for (int l=0; l<COL; l++)
        cout << *(*(array + k) + l) << " " ;
    cout << endl ;
}
}

```

with output:

```

array[i][j]:
1 2 3
4 5 6
*(*(array+i)+j):
1 2 3
4 5 6

```

We saw in Chapter 7 that an element of a two-dimensional array at row position *i* and column position *j* can be accessed using the compound subscript operators:

`array[i][j]`

The equivalent pointer notation is:

`*(*(array + i) + j)`

## 12.16 Pointers, Arrays and Function Arguments

The topic of passing arrays as function arguments was covered in Chapter 7, but without reference to pointers. Consider the following program:

```

// paafaa.cpp
// illustrates pointers, arrays and function arguments
#include <iostream.h> // C++ I/O

// returns the sum of an array of integers

```

```

// access array elements by subscript operator []
double SubscriptSum (const double array[], int SIZE)
{
    double sum = 0 ;
    for (int i=0; i< SIZE; i++)
        sum += array[i] ;
    return sum ;
}

// returns the sum of an array of integers
// access array elements by pointer and increment operator ++
double PointerSum (const double* array, int SIZE)
{
    double sum = 0 ;
    for (int i=0; i< SIZE; i++)
        sum += *array++ ;
    return sum ;
}

void main ()
{
    double a_main[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 } ;

    cout << "SubscriptSum (a_main): "
         << SubscriptSum (a_main, 5) << endl ;
    cout << "PointerSum (a_main) : "
         << PointerSum (a_main, 5) << endl ;
}

```

First of all, let's remind ourselves of the technique used in Chapter 7 of passing arrays to functions and indexing an array's elements:

```

double SubscriptSum (const double array[], int SIZE)
{
    //...
    sum += array[i] ;
    //...
}

```

Both the array name and subscript operator, [ ], are used within the function parentheses to indicate to the compiler that an array is being passed to the function. The elements of the array are accessed using the array subscript operator.

However, it is common practice when passing arrays as function arguments to use pointers rather than the name of an array and the subscript operator. For example, the C++ library function *strcmp()* is passed two constant arrays of characters as pointers:

```
int strcmp (const char* s1, const char* s2) ;
```

The function *PointerSum()* in program PAAFA.CPP uses pointer notation:

```
double PointerSum (const double* array, int SIZE)
```

---

```
{
//...
sum += *array++ ;
//...
}
```

The values of the elements of an array are accessed by dereferencing the `array` pointer function argument and using the postfix increment operator, as outlined in the previous sections.

It was mentioned in Chapter 7, with reference to program ARG.CPP and function `Display3()`, that it is not possible to pass a two-dimensional array to a function in the general form:

```
void Display3 (int array[][], int row, int col)
{
//...
}
```

This function declarator is illegal because the compiler must know the column dimension of an array so that it can find the memory address of the array. Pointers offer a neat solution to the problem of passing a two-dimensional array to a function whose size is determined by the `row` and `col` arguments:

```
// f_p_a_2d.cpp
// illustrates passing a general
// two-dimensional array to a function
#include <iostream.h> // C++ I/O

void Display (int* array, int row, int col)
{
    for (int i=0; i<row; i++)
    {
        for (int j=0; j<col; j++)
            cout << array[i*col + j] << " ";
        cout << endl ;
    }
}

void main ()
{
    const int ROW = 2, COL = 3 ;

    int array[ROW][COL] = { {1, 2, 3}, {4, 5, 6} } ;

    Display (&array[0][0], ROW, COL) ;
}
```

with output:

```
1 2 3
4 5 6
```

The function *Display()* is passed a pointer to **int** which points to the start of a two-dimensional array. Note the use of the address-of operator in the *Display()* function call. Within the function body of *Display()* the elements of an array are consecutively accessed using the expression:

```
array[i*col + j]
```

demonstrating that the elements of a two-dimensional array are in fact contiguously stored in memory as a one-dimensional array.

## 12.17 Pointers to Pointers

We have seen that a pointer can point to the memory address of another object:

```
int* ptr ; // rvalue of ptr holds lvalue of object of type int
```

C++ allows us to define a pointer to a pointer. The first pointer points to the memory address of the second pointer which, as we have seen previously, points to the memory address of an object; see Fig. 12.3. The following program helps illustrate a pointer to a pointer:

```
// ptr_ptr.cpp
// illustrates a pointer to a pointer
#include <iostream.h> // C++ I/O

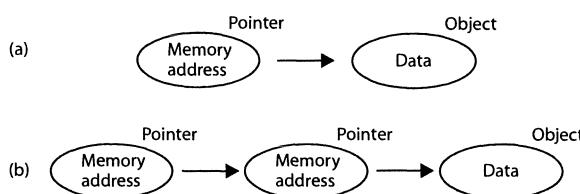
void main ()
{
    int a = 1 ;

    int* ptr1 ; // pointer to type int
    int** ptr2 ; // pointer to an int pointer

    ptr1 = &a ; // assign memory addresses
    ptr2 = &ptr1 ;

    cout << "&a : " << &a << endl ;
    cout << "a : " << a << endl ;

    cout << "&ptr1 : " << &ptr1 << endl ;
    cout << "ptr1 : " << ptr1 << endl ;
    cout << "*ptr1 : " << *ptr1 << endl ;
```



**Fig. 12.3** (a) Pointer to an object. (b) Pointer to a pointer, which in turn points to an object.

---

```

cout << "&ptr2 : " << &ptr2 << endl ;
cout << "ptr2 : " << ptr2 << endl ;
cout << "*ptr2 : " << *ptr2 << endl ;
cout << "**ptr2: " << **ptr2 << endl ;
}

```

with output:

```

&a      : 0x450721e6 // lvalue of a
a       : 1           // rvalue of a
&ptr1   : 0x450721e2 // lvalue of ptr1
ptr1    : 0x450721e6 // rvalue of ptr1 (loc. of a)
*ptr1   : 1           // rvalue of a
&ptr2   : 0x450721de // lvalue of ptr2
ptr2    : 0x450721e2 // lvalue of ptr1
*ptr2   : 0x450721e6 // rvalue of ptr1 (loc. of a)
**ptr2: 1           // rvalue of a

```

The output (with added comments) illustrates that `ptr1` points to the memory address of `int` variable `a` and `*ptr1` is the actual value of `a`. Pointer `ptr2` points to the memory address of `ptr1`, `*ptr2` is the rvalue of `ptr1`, which in turn is the memory address of `a`, and `**ptr2` is the value of `a`. Note that `ptr2` is a pointer to an `int` pointer and not a pointer to `int`.

## 12.18 Arrays of Pointers and Pointers to Arrays

C++ allows arrays of pointers:

```

// array_p.cpp
// illustrates an array of pointers
#include <iostream.h> // C++ I/O

const int N_ELEMENTS = 3 ;
const int N_ARRAYS = 2 ;

// sums elements of a 1D array
int SumArray (int* p, int n_elements)
{
    int sum = 0 ;
    for (int i=0; i<n_elements; i++)
    {
        sum += *p++ ;
    }
    return sum ;
}

// sum elements of a list of arrays
int Sum (int** pp, int n_list)
{

```

```

int sum = 0 ;
for (int i=0; i<n_list; i++)
{
    sum += SumArray (* (pp+i), N_ELEMENTS) ;
}
return sum ;
}

void main ()
{
    int* array_p[N_ARRAYS] ; // array of pointers to
                           // type int

    int array1[N_ELEMENTS] = {1, 2, 3} ;
    int array2[N_ELEMENTS] = {4, 5, 6} ;

    array_p[0] = array1 ;
    array_p[1] = array2 ;

    cout << "sum: " << Sum (array_p, N_ARRAYS) ;
}

```

Within *main()*, an array of pointers to **int** is defined:

```
int* array_p[N_ARRAYS] ;
```

*array\_p* is an array of *N\_ARRAYS* integer pointers. The above program defines two arrays, *array1* and *array2*, of type **int** and then proceeds to assign the address of each array to a pointer array:

```
array_p[0] = array1 ;
array_p[1] = array2 ;
```

Alternatively, we could have used the address-of and subscript operators explicitly:

```
array_p[0] = &array1[0] ;
array_p[1] = &array2[0] ;
```

The first argument of the *Sum()* function declarator is a pointer to an **int** pointer:

```
int Sum (int** pp, int n_list)
```

The pointer to an **int** pointer allows us to pass to the *Sum()* function an array of pointers, which in the present case is a list of one-dimensional **int** arrays.

A popular use of arrays of pointers is with an array of strings. We saw in Chapter 7 (program INIT\_STR.CPP and Fig. 7.6) that a two-dimensional array of constant strings wastes memory space if all the strings are not of the same length. An array of pointers to strings wastes no memory space:

```
// a_p_str.cpp
// illustrates arrays of pointers to strings
```

```
#include <iostream.h> // C++ I/O

void main ()
{
const int MAX_STR_LEN = 9 ;
const int NUM = 5 ;

// array notation (wastes 16 of the 45 bytes allocated)
char names[NUM][MAX_STR_LEN] =
{ "Graham", "Ian", "Jim", "Jonathan", "Tony" } ;

// pointer notation
char* p_names[] =
{ "Graham", "Ian", "Jim", "Jonathan", "Tony" } ;

// O/P names
for (int i=0; i< NUM; i++)
{
    cout << names[i] << " " ;
}
cout << endl ;
// O/P (backwards) p_names
for (int j=NUM-1; j>= 0; j--)
{
    cout << p_names[j] << " " ;
}
}
```

Within `main()`, an array of pointers to strings is defined and initialised as:

```
char* p_names[] =
{ "Graham", "Ian", "Jim", "Jonathan", "Tony" } ;
```

Since the strings do not form part of an array the compiler will store them contiguously in memory, thus wasting no space. Refer to Fig. 12.4 for a comparison between a two-dimensional array of strings and an array of pointers to strings.

Note that arrays of pointers can be initialised when defined, just like *normal arrays*.

The above program illustrated an array of pointers. C++ also allows us to define a pointer to an array. The following program illustrates a pointer to an array:

```
// p_array.cpp
// illustrates pointers to arrays
#include <iostream.h> // C++ I/O

const int SIZE = 2 ;

void main ()
{
    int (*p_array)[SIZE] ;           // pointer to an array

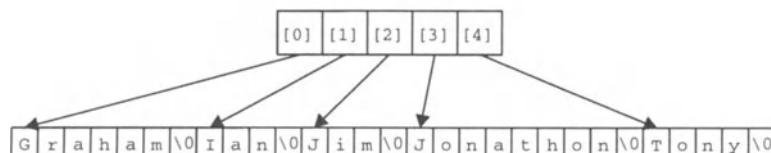
    int array[SIZE] = {1, 2} ;     // an array
```

```
char names[5][9]={"Graham", "Ian", "Jim", "Jonathon", "Tony"};
```

names[0]	G	r	a	h	a	m	\0		
	I	a	n	\0					
	J	i	m	\0					
	J	o	n	a	t	h	o	n	\0
names[4]	T	o	n	y	\0				

(a)

```
char* p_names[5]={"Graham", "Ian", ...};
```



(b)

**Fig. 12.4** (a) A two-dimensional array of strings which wastes 16 of the 45 bytes allocated. (b) An array of pointers to strings.

```
// assign memory address
// of array to pointer
p_array = &array ;

cout << "&array[0]: " << &array[0] << endl ;
cout << "p_array : " << p_array << endl ;
}
```

with output:

```
&array[0]: 0x4e0721aa
p_array : 0x4e0721aa
```

which illustrates that the `p_array` pointer holds the memory address of `array`.

The statement:

```
int (*p_array) [SIZE] ;
```

defines a pointer, `p_array`, to an array with `SIZE` elements. The memory address of an array, `array`, is assigned to the rvalue of the pointer to an array with the help of the address-of operator:

```
p_array = &array ;
```

## 12.19 Pointers to String Constants

The previous section illustrated the use of pointers with string constants. Consider the following program:

```
// ptr_str.cpp
// illustrates pointers to string constants
#include <iostream.h> // C++ I/O
void main ()
{
    char string[] = "array string constant" ;

    char* str = "pointer string constant" ;

    cout << string << endl ;
    cout << str << endl ;
}
```

This program is perfectly legal, although memory has not explicitly been requested in the above program for the string constant ‘pointer string constant’. This is because the string constant is placed in a *string table* which is generated by the compiler. The pointer `str` is a pointer to the string in the string table. However, I do not recommend using the pointer-to-string-constant approach if the string is to be modified, because commercial C++ compilers generally generate optimised string tables which allow a string constant to be accessed from different parts of a program. When the `new` and `delete` operators are introduced later we shall see a better method of defining a pointer to a string, allocating sufficient memory for the string and finally deleting the memory allocated to hold the string when we are finished with it.

## 12.20 Pointers to Functions

This section examines the use of pointers to point to the memory address of a function. This is a powerful use of pointers in C++ which allows a general function to be invoked simply by an application of a pointer.

To date we have seen functions that have been passed objects by value, reference or pointer:

```
double      sqrt  (double x) ;           // value (MATH.H)
friend complex sqrt  (complex& c) ;       // reference
                                         // (COMPLEX.H)
size_t      strlen (const char* s) ;     // pointer
                                         // (STRING.H)
```

In program PTR\_VOID.CPP the use of the C++ library function `qsort()` illustrated that a pointer to a function can also be passed as a function argument:

```
void qsort (void* base, size_t nelem, size_t width,
           int (*fcmp) (const void*, const void*)) ;
```

`fcmp` is a pointer to a comparison function.

To illustrate pointers to functions, let's examine the Newton–Raphson iterative method for solving equations of the form  $f(x)=0$ , where  $f$  is an arbitrary function with a continuous derivative  $f'$ . The Newton–Raphson method is a popular technique for determining the roots of an equation because it is simple, efficient and has a graphical interpretation. Figure 12.5 illustrates the graph of  $y=f(x)$  for the range  $a \leq x \leq b$ , where  $a$  and  $b$  are constants. Given an initial approximate value for  $x$ ,  $x_0$ , we let  $x_1$  be the point of intersection of the tangent to the curve of  $f$  at  $x_0$  with the  $x$ -axis:

$$\tan\alpha = f'(x_0) = \frac{f(x_0)}{x_0 - x_1}, \quad \text{or} \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

and similarly for point  $x_1$  and so on until a required degree of convergence has been achieved. Thus, the general iterative Newton–Raphson formula is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \dots$$

In the program below the termination criteria adopted is that convergence has been achieved when the following condition is satisfied:

$$|x_{n+1} - x_n| \leq \epsilon$$

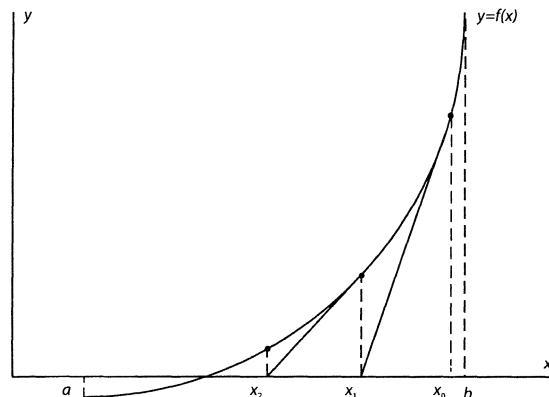
where  $\epsilon$  is a small non-zero constant.

To help emphasise the generality of pointers to functions, the function which implements the Newton–Raphson method, *NewtonRaphson()*, is placed in the file NR.CPP. The *NewtonRaphson()* function is developed with no prior knowledge of the functions that it will operate on, only that they will be of the form:

```
double arbitrary_function (double) ;
```

The function declaration of *NewtonRaphson()* is given in the NR.H header file:

```
// nr.h
// header file for Newton-Raphson method
```



**Fig. 12.5** The Newton–Raphson method.

---

```
#ifndef _NR_H // prevent multiple includes
#define _NR_H

#include <math.h> // fabs()

enum Boolean {FALSE, TRUE} ;

void NewtonRaphson (double (*f_ptr)(double),
                     double (*df_ptr)(double),
                     int n_iterations, double tolerance,
                     double x0, double& x_new, int& count,
                     Boolean& converged) ;

#endif // _NR_H
```

The implementation file is NR.CPP:

```
// nr.cpp
// implementation of Newton_Raphson method

#include "nr.h" // header

// performs Newton-Raphson iteration
void NewtonRaphson (double (*f_ptr)(double),
                     double (*df_ptr)(double),
                     int n_iterations, double tolerance,
                     double x0, double& x_new,
                     int& count, Boolean& converged)
{
    double x_old = x0 ; // initialise x_old to initial guess

    count = 0 ; // no. of iterations required for convergence
    converged = FALSE ;

    for (int i=0; i<n_iterations; i++)
    {
        count++ ;
        // Newton-Raphson
        x_new = x_old - (f_ptr (x_old) / df_ptr (x_old)) ;

        // if converged
        if (fabs(x_new - x_old) < tolerance)
        {
            converged = TRUE ;
            break ;
        }
        // else continue using new estimate
        else
            x_old = x_new ;
    }
} // NewtonRaphson()
```

The program which tests the *NewtonRaphson()* function is P\_FUNC.CPP:

```
// p_func.cpp
// illustrates pointers to functions
#include <iostream.h> // C++ I/O
#include <math.h> // fabs()
#include <iomanip.h> // setprecision()

#include "nr.h" // NewtonRaphson()

const int MAX_ITERATIONS = 100 ;
const double TOLERANCE = 1e-06 ;

// function prototypes
double Function (double x) ;
double DFunction (double x) ;

void main ()
{
    double x_initial, x_solution = 0.0 ;
    int iter_to_converge = 0 ;
    Boolean converge_or_not = FALSE ;

    cout << endl << "enter an initial value of x
                      for the solution of f(x)=x^2-2=0: " ;
    cin >> x_initial ;

    NewtonRaphson (Function, DFunction, MAX_ITERATIONS,
                    TOLERANCE, x_initial, x_solution,
                    iter_to_converge, converge_or_not) ;

    if (converge_or_not)
    {
        cout << endl
            << "converged after "
            << iter_to_converge << " iterations" << endl
            << "solution of f(x)=x^2-2=0: "
            << setprecision (7) << x_solution ;
    }
    else
        cout << "sorry-no convergence" ;
} // main()

// function
double Function (double x)
{
    // f(x)=x^2-2
    double fx = x * x - 2.0 ;
    return fx ;
}
```

---

```
// derivative of function
double DFunction (double x)
{
    // f'(x)=2x
    double deriv = 2.0 * x ;
    return deriv ;
}
```

with some user interaction:

```
enter an initial value of x for the solution of
f(x)=x^2-2=0: 10

converged after 7 iterations
solution of f(x)=x^2-2=0: 1.414214
```

As an example, the function  $f(x)=x^2-2=0$ ,  $f'(x)=2x$  is chosen to test the Newton–Raphson method for computing the square root of 2. The output illustrates that the *NewtonRaphson()* function computes  $\sqrt{2}$  to an accuracy of six decimal places in only seven iterations given an initial estimate of  $x_0=10$ .

The function declarator of *NewtonRaphson()* is:

```
void NewtonRaphson (double (*f_ptr)(double),
                     double (*df_ptr)(double), int
                     n_iterations, double tolerance,
                     double x0, double& x_new,
                     int& count, Boolean& converged) ;
```

The first two arguments are pointers to functions, which we shall examine in more detail shortly. The third and fourth arguments are the maximum number of iterations of the Newton–Raphson method and the specified level of accuracy (i.e.  $\epsilon$ ), respectively. An upper limit of iterations must be specified so that the function does not iterate indefinitely. Argument number five is the initial guess,  $x_0$ , and the sixth argument is the solution to the specified tolerance if convergence is achieved. The last two arguments specify the number of iterations required to reach the specified level of solution accuracy and whether or not convergence was achieved.

It was noted in Chapter 7 that an array name is in fact a synonym for the memory address of the zeroth element of the array. Similarly, C++ adopts the rule that a function's name is the memory address of the function. In C++, a pointer to a function, *f\_ptr*, is defined as:

```
return_typeSpecifier (*f_ptr) (typeSpecifier1, ...);
```

For example, given an arbitrary function of the form:

```
double Function (double) ;
```

a pointer to this function is defined as:

```
double (*f_ptr) (double) ;
```

The memory address of function *Function()* can then be assigned to the pointer *f\_ptr*:

```
f_ptr = Function ;
```

Note that only the function name is used in the assignment.

Since the rvalue of `f_ptr` contains the memory address of `Function()`, dereferencing `f_ptr` using the indirection operator `*` will simply call `Function()`:

```
double d ;
//...
(*f_ptr) (d) ; // calls the function pointed to by f_ptr
```

The parentheses encompassing `*f_ptr` are required because the precedence level of the indirection operator is less than that of the function call operator `()`. Actually, the indirection operator is not required, since attaching the list of arguments to `f_ptr` dereferences `f_ptr` by default:

```
double d ;
//...
f_ptr (d) ; // calls the function pointed to by f_ptr
```

A pointer to a function can also be initialised when it is defined:

```
double (*f_ptr) (double) = Function ;
```

These features of pointers to functions are illustrated in the implementation of the `NewtonRaphson()` function:

```
void NewtonRaphson (double (*f_ptr)(double),
                     double (*df_ptr)(double), /*...*/
                     //...
                     {
                     for (int i=0; i<n_iterations; i++)
                     {
                     //...
                     x_new = x_old - (f_ptr (x_old) / df_ptr (x_old)) ;
                     //...
                     }
```

which defines two pointers to functions `f_ptr` and `df_ptr` that point to  $f(x)$  and  $f'(x)$  respectively.

You may think that the use of pointers to functions makes the `NewtonRaphson()` function declarator look a bit messy. If so, then a `FunctionPointer` **typedef** can be defined as a pointer to a function whose return type is `double` and has one `double` function argument:

```
typedef double (*FunctionPointer) (double) ;
```

`typedef` `FunctionPointer` allows us to rewrite the `NewtonRaphson()` declarator as:

```
void NewtonRaphson (FunctionPointer f_ptr,
                     FunctionPointer df_ptr, int n_iterations,
                     double tolerance, double x0,
```

---

```
double& x_new, int& count,
Boolean& converge_flag) ;
```

Finally, the equality (`==`) and inequality (`!=`) relational operators can be applied to pointers to functions. This can be useful to test whether two or more function pointers point to the same function:

```
void NewtonRaphson (FunctionPointer f_ptr,
                      FunctionPointer df_ptr, /*...*/
//...
if (f_ptr == df_ptr) // pointers point to same function
{
    cout << "pointers to same function" << endl ;
    exit (EXIT_SUCCESS) ;
}
else // o.k.
{
//...
}
```

## 12.21 Returning a Pointer From a Function

We have seen functions that return an object and a reference to an object. Functions can also return a pointer. Clearly, to return a pointer a function must declare the return type specifier to be a pointer. As an example, the C++ `strcpy()` library function returns a pointer to `char`:

```
char* strcpy (char* dest, const char* src) ;
```

The pointer returned from `strcpy()` is in fact a pointer to the destination string, `dest`. The following program defines a function `StrCopy()` which mimics `strcpy()`:

```
// f_ret_p.cpp
// illustrates functions that return a pointer
#include <iostream.h> // C++ I/O

// copies a string, src, and returns a pointer to dest
char* StrCopy (char* dest, const char* src)
{
    char* start_dest = dest ; // record start of dest

    while (*dest++ = *src++) // copy src to dest
        ;
    return start_dest ; // return pointer to char
}

void main ()
{
    char string1[] = "string1" ;
    char string2[] = "string2" ;
```

```
char* dest = StrCopy (string1, string2) ;
cout << dest ;
}
```

The **return** statement returns a pointer to the destination string.

A function can also return a pointer to **void**. For instance, the C++ library function *memcpy()* returns a **void\*** and has the following signature:

```
void* memcpy (void* dest, const void* src, size_t n) ; // MEM.H
```

*memcpy()* copies *n* bytes from the source block, *src*, to the destination block, *dest*, and returns a pointer to **void** to the destination block. Receiving and returning pointers to **void** allows *memcpy()* to deal with a variety of different data types.

## 12.22 Casting

Pointers can be cast from one base type to another. Consider the following program, which prompts a user to enter the size of a square array and then proceeds to display an array of pseudo-random numbers of the requested size:

```
// cast_p.cpp
// illustrates casting pointers
#include <iostream.h>    // C++ I/O
#include <stdlib.h>      // malloc(), exit(), rand(), srand()
#include <time.h>        // time(), time_t
#include <iomanip.h>      // setw()

// allocate sufficient memory & return pointer to memory
// block
int* Allocate (int n)
{
    int* i_ptr = (int*) malloc ((size_t)(n*n*sizeof(int))) ;

    if (i_ptr == NULL) // unsuccessful request
    {
        cout << "unsuccessful memory allocation" ;
        exit (EXIT_SUCCESS) ;
    }
    return i_ptr ;
}

// frees allocated memory
void DeAllocate (int* array)
{
    if (array != NULL) // prevent freeing NULL pointer
        free ((void*) array) ;
}

// gets a size
```

```

int GetSize ()
{
    int size = 0 ;

    do
    {
        cout << "enter size of a square array: " ;
        cin >> size ;
    } while (size <= 0) ;
    return size ;
}

// fills an array with random numbers (0:99)
void FillArray (int* array, int size)
{
    // initialise random number generator
    time_t t ;
    srand (unsigned (time(&t))) ;

    for (int i=0; i<size; i++)
        for (int j=0; j<size; j++, array++)
            *array = rand () % 100 ;
}

// displays an array
void DisplayArray (int* array, int size)
{
    for (int i=0; i<size; i++)
    {
        for (int j=0; j<size; j++, array++)
            cout << setw (2) << *array << " " ;
        cout << endl ;
    }
}

void main ()
{
    int* array = NULL, size = 0 ;

    size = GetSize () ;           // get size of array
    array = Allocate (size) ;     // allocate sufficient memory

    FillArray (array, size) ;    // fill array with numbers
    DisplayArray (array, size) ; // display array

    DeAllocate (array) ;         // free memory
}

```

Some user interaction is:

```
enter size of a square array: 5
```

```
27 59 40 24 59
66 83 64 73 89
81 20 79 4 14
65 52 79 56 71
71 57 89 85 84
```

C++ includes a standard library function called *malloc()* which allocates memory from the heap. The signature of *malloc()* is:

```
void* malloc (size_t size) ;
```

and a program using *malloc()* should include either of the header files STDLIB.H or ALLOC.H. *size* is the requested size of memory allocation in bytes. The return type of *malloc()* is a pointer to **void** and the argument type is **size\_t**, which is declared as:

```
typedef unsigned size_t ;
```

Returning a pointer to **void** greatly increases the generality of the *malloc()* function. Within the function body of *Allocate()* the *malloc()* function is used to allocate sufficient memory for an array of integers of size *n*×*n*:

```
int* Allocate (int n)
{
    int* i_ptr = (int*) malloc ((size_t) (n*n*sizeof(int))) ;
    //...
}
```

Since the return type of *malloc()* is a pointer to **void** we must cast the pointer returned from *malloc()* to be a pointer to **int**. The type cast takes the form of:

```
(int*)
```

If *malloc()* successfully allocates the amount of memory requested, a pointer is returned to the start of the memory block allocated from the heap. If there is insufficient memory available to satisfy the memory request, *malloc()* returns the NULL pointer. In the above program, if the memory request is successful the memory address of the block of memory is assigned to the rvalue of *i\_ptr*.

If we specifically request a block of memory from the heap, it is our responsibility to ensure that this memory is deleted when we are finished with it. If we do not delete the block of allocated memory when we are finished using it we are clearly relying on the operating system to delete the memory for us. What the operating system does with memory is generally beyond our control – so always ensure that a memory allocation is accompanied by a deallocation. A C++ library function that frees an allocated block of memory is *free()*:

```
void free (void* block) ;
```

The single function argument of *free()* is a pointer to **void**. The arrays that program CAST\_P.CPP deals with are of type **int**, and it is therefore necessary to type cast an **int** array, array, to a **void** pointer before *free()* is called:

```
void DeAllocate (int* array)
```

---

```

{
if (array != NULL) // prevent freeing NULL pointer
    free ((void*) array) ;
}

```

Although the **if**-statement is not essential, it does remove the possibility of deleting a pointer that was not allocated memory. Freeing a NULL pointer using *free()* can cause some serious problems.

The above program is an example of dynamic memory allocation. It was mentioned, in connection with program STATIC.CPP of Chapter 7, that the size of an array has to be a constant fixed at the time of compilation. The following code is an extract from STATIC.CPP:

```

void main ()
{
    int array_size ;

    cout << "enter an array size: " ;
    cin >> array_size ;

    // define a 1D array of a size determined by a user
    int array[array_size] ; // error: const. expression
                           // required.
    //...
}

```

The program CAST\_P.CPP, through the use of pointers, memory allocation and deallocation functions allows us to allocate memory dynamically at run-time according to the requirements of a program. Use of the functions *malloc()*, *calloc()*, *realloc()* and *free()* to allocate and deallocate memory dynamically is the generally adopted style when programming in C. We shall see shortly that although C++ still supports this C style of dynamic memory allocation, the generally adopted C++ approach to dynamic allocation is through the use of the **new** and **delete** operators.

## 12.23 Recap

To date we have seen a number of applications of pointers, so before introducing pointers to objects and the **new** and **delete** operators, Table 12.1 summarises several declarations that we've seen so far.

## 12.24 Pointers to Objects

Pointers would be of limited use if they could not point to objects of user-defined structures and classes. To illustrate pointers to objects we shall examine the *Point* **class** introduced in Chapter 9. An object of **class** *Point* is defined with the statement:

```
Point p ; // object p
```

**Table 12.1** Declarations.

Declaration	Example	Description
type name ;	int var ;	Variable
type name[] ;	int array[] ;	Array
type* name ;	int* i_ptr ;	Pointer variable
type* name[] ;	int* array_ptr[] ;	Array of pointers
type (*name)[] ;	int (*ptr_array)[] ;	Pointer to an array
type& name ;	int& i_ref ;	Reference
type name () ;	int Func () ;	Function
type& name () ;	int& Func () ;	Function returning a reference
type* name () ;	int* Func () ;	Function returning a pointer
type (*name) () ;	int (*f_ptr) () ;	Pointer to a function

The syntax for defining a pointer to **class Point** is identical to that of defining a pointer variable to an integral type:

```
Point* p_ptr ; // pointer to class Point
```

The following program examines the above two approaches:

```
// p_obj.cpp
// illustrates pointers to objects
#include <iostream.h> // C++ I/O

// class Point
class Point
{
private:
    // private data members
    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (double d)
        : x (d), y (d), z (d) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    double& X () { return x ; }
    double& Y () { return y ; }
    double& Z () { return z ; }
    const double& X () const { return x ; }
    const double& Y () const { return y ; }
    const double& Z () const { return z ; }
    // overloaded operators
    Point operator = (const Point& p) ;
```

```

// friends
friend ostream& operator << (ostream& s,
                                const Point& p) ;
friend istream& operator >> (istream& s, Point& p) ;
}; // class Point

// overloaded operators:

// assignment operator
inline Point Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return Point (x, y, z) ;
}

// overloaded operators/friends:

// overloaded insertion operator (object)
inline ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
}

// overloaded extraction operator
inline istream& operator >> (istream& s, Point& p)
{
    return s >> p.x >> p.y >> p.z ;
}

void main ()
{
    Point p (1.0, 2.0, 3.0) ; // object
    Point* p_ptr (NULL); // pointer to Point
    p_ptr = &p ; // p_ptr points to p

    cout << "p      : " << p      << endl
        << "&p     : " << &p     << endl
        << "p_ptr  : " << p_ptr  << endl
        << "*p_ptr: " << *p_ptr << endl ;

    cout << endl
        << "(*p_ptr).X(): " << (*p_ptr).X () << endl
        << "p_ptr->X()  : " << p_ptr->X () << endl ;
}

```

with output:

```

p      : (1, 2, 3)
&p     : 0x442f23a8
p_ptr  : 0x442f23a8
*p_ptr: (1, 2, 3)

```

```
(*p_ptr).X() : 1  
p_ptr->X()   : 1
```

The statement:

```
Point* p_ptr (NULL) ;
```

defines a pointer to `Point` and simultaneously initialises the pointer to the `NULL` pointer. Then the rvalue of `p_ptr` is assigned the memory address of object `p`. The output illustrates that `p_ptr` points to the memory address of object `p` and that dereferencing `p_ptr` (i.e. `*p_ptr`) accesses the `Point` object pointed to by `p_ptr`.

The final statement within the `main()` function is:

```
cout << endl  
<< "(*p_ptr).X() : " << (*p_ptr).X () << endl  
<< "p_ptr->X()   : " << p_ptr->X ()   << endl ;
```

and illustrates calling the member function `Point::X()` using the pointer `p_ptr` rather than the object `p`. The *direct member access* operator (`.`) requires that the left-hand operand acted on by the operator is an object of a **class** in which the right-hand operand is a member. Hence the following is legal:

```
p.X () ; // o.k.: p is an object
```

However, the following statement is illegal:

```
p_ptr.X () ; // error: p_ptr is not an object
```

Since the direct member access operator must act on an object, we must deference the pointer `p_ptr` to enable us to call the member function `Point::X()`:

```
(*p_ptr).X () ; // o.k.: *p_ptr refers to the  
//          object pointed to by p_ptr
```

This syntax works fine, but C++ offers a more attractive syntax, via the *indirect member access* operator (`->`), which specifically acts on pointers to objects:

```
p_ptr->X () ; // o.k.: -> acts on pointers to objects
```

All of the above observations equally apply to structures.

## 12.25 Pointers to Data Members and Member Functions

Previous sections have illustrated pointers to functions and pointers to objects. The C++ programming language also allows a programmer to define pointers to the data members and member functions of a **class**. There exist in the C++ programming language two binary operators that we have not discussed yet, namely the *pointer to member* operators `.` `*` and `->`. The pointer to member operators provide a means of accessing a **class**'s data member or member function via a pointer. To illustrate **class** member pointers, consider the program:

```

// ptr_mem.cpp
// illustrates the pointer-to-member
// operators .* and ->*
#include <iostream.h> // C++ I/O

class X
{
public:
    int data ;

    X () { data = 0 ; }
    void Set (int i) { data = i ; }
    void Display () { cout << "X::data: " << data << endl ; }
};

void main ()
{
    // define & initialise pointer to class X int data member
    int X::*pidm      = &X::data ;
    // define & initialise pointers to class X member
    // functions
    void (X::*pimf)(int) = &X::Set ;
    void (X::*pvmf)()   = &X::Display ;

    X x ; // define object x

    (x.*pvmf)() ; // call X::Display()
    (x.*pimf)(1) ; // call X::Set()
    (x.*pvmf)() ; // call X::Display()

    x.*pidm = 2 ;
    cout << "X::data: " << x.*pidm << endl ;

    X* y = &x ; // define and initialise pointer to an object

    (y->*pvmf)() ; // call X::Display()
    (y->*pimf)(1) ; // call X::Set()
    (y->*pvmf)() ; // call X::Display()

    y->*pidm = 0 ;
    cout << "X::data: " << y->*pidm << endl ;
}

```

Within `main()`, three pointers (`pidm`, `pimf` and `pvmf`) are defined and simultaneously initialised:

```

// define & initialise pointer to class X int data member
int X::*pidm      = &X::data ;
// define & initialise pointers to class X member functions
void (X::*pimf)(int) = &X::Set ;
void (X::*pvmf)()   = &X::Display ;

```

pidm is a pointer to the **int** data data member of **class X**, while pimf and pvmf are pointers to member functions of **class X**. The scope resolution operator (`::`) associates each pointer with **class X**. An object, x, is then defined of **class X**, the `X::Set()` member function is called and the `X::data` member is assigned the constant 2 through the use of the pointers and the `.*` pointer to member operator:

```
X x ;  
//...  
(x.*pimf)(1) ; // call X::Set()  
//...  
x.*pidm = 2 ; // assign 2 to X::data
```

Clearly, the `X::data` data member must be accessible in order to perform the above assignment.

Next, a pointer to **class X** is defined and initialised to the object x. The `->*` operator is then used to call `X::Set()` and assign a constant to `X::data`:

```
X* y = &x ;  
  
(y->*pimf)(1) ; // call X::Set()  
//...  
y->*pidm = 0 ; // assign 0 to X::data
```

Why do we need the `.*` and `->*` operators? It turns out that the above pointers, pidm, pimf and pvmf, are in fact offsets and do not define the actual memory addresses of data members or member functions. The offset defines a position in the object that identifies a member of the object.

The `.*` operator cannot be overloaded, whereas the `->*` operator can be overloaded.

## 12.26 The **new** and **delete** Operators

The CAST\_P.CPP program illustrated the method generally adopted in C for allocating and deallocating memory dynamically via the `malloc()` and `free()` functions. C++ contains two operators, **new** and **delete**, which perform similar operations to `malloc()` and `free()` through the use of pointers. There exist small syntactical differences between the application of **new** and **delete** for objects and arrays of objects, so we shall begin by examining objects.

### 12.26.1 Objects

The general syntax of the **new** and **delete** unary operators for objects is:

```
pointer_variable = new class_name (optional_parameter_list) ;  
  
delete pointer_variable ;
```

`pointer_variable` is a pointer to **class** or type `class_name`. When the **new** operator is invoked a block of memory is requested from the operating system which is sufficient in size

to hold an object of **class** class\_name. If the memory request is successful the **new** operator returns a pointer to the allocated block of memory. If the memory request is unsuccessful **new** returns the NULL pointer. Depending on a **class**'s overloaded constructors (assuming they are defined), an optional list of parameters may accompany class\_name.

The **delete** operator frees or deallocates memory allocated using the **new** operator and pointed to by pointer\_variable. Do not attempt to use the **delete** operator to free memory allocated by a means other than by the **new** operator.

Let's examine a program which uses the **new** and **delete** operators with objects of **class** Point:

```
// nd_obj.cpp
// illustrates the new and delete operators applied to
// objects
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()

// class Point
class Point
{
    //
    //...
};

//...
void main ()
{
    // pointers p_ptr and q_ptr to Point
    Point* p_ptr (NULL), * q_ptr (NULL) ;

    // allocate memory
    p_ptr = new Point ; // default constructor
    q_ptr = new Point (1.0, 2.0) ; // 3 arg. constructor

    if (!p_ptr || !q_ptr)
    {
        cout << "unsuccessful memory allocation" ;
        exit (EXIT_SUCCESS) ;
    }

    cout << "*p_ptr: " << *p_ptr ;
    cout << "*q_ptr: " << *q_ptr ;

    // free allocated memory
    delete p_ptr ;
    delete q_ptr ;
}
```

The two statements:

```
p_ptr = new Point ;
q_ptr = new Point (1.0, 2.0) ;
```

request sufficient memory from the heap to hold two objects of **class** Point and assign to **p\_ptr** and **q\_ptr** the memory addresses where the objects will be held in memory. The above two statements make use of Point's default and three-argument constructors, respectively, to initialise the memory area pointed to by **p\_ptr** and **q\_ptr**. Remember that an object's constructor is called automatically when an object is created. Similarly, constructors are called when memory is allocated dynamically using the **new** operator. When we are finished with the pointers **p\_ptr** and **q\_ptr** the **delete** operator is used to free the memory pointed to by **p\_ptr** and **q\_ptr**. Note that although the memory pointed to by **p\_ptr** and **q\_ptr** is deleted, the pointers themselves are not deleted. Beware of using a pointer that points to a block of memory that has been deleted.

The use of the **new** operator is similar in principle to the **malloc()** function, but does not require casting the pointer to **void** returned from **malloc()** to the appropriate **class**:

```
Point* p_ptr = (Point*) malloc((size_t) sizeof(Point)) ;
```

and similarly for the **free()** function.

When using the **delete** operator you have to know what you are doing. The following program examines three uses of **delete**:

```
// delete.cpp
// illustrates the delete operator
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()

void main ()
{
    // 1: define, allocate, test, delete
    int* i_ptr = new int ; // allocate

    if (!i_ptr)
    {
        cout << "unsuccessful memory allocation" ;
        exit (EXIT_SUCCESS) ;
    }
    delete i_ptr ; // deallocate

    // 2: define initialised pointer to NULL, delete
    int* j_ptr (NULL) ;
    delete j_ptr ;

    // 3: define uninitialised pointer, delete
    int* k_ptr ;
    delete k_ptr ; // runtime error
}
```

The first case defines a pointer to the base type **int**, allocates sufficient memory for an **int** and initialises the pointer to the memory address of the block of allocated memory. If the operating system is unable to allocate the requested memory the program terminates. Finally, the **i\_ptr** pointer is deleted. The second case uses the **delete** operator to delete a pointer that is initialised to the **NULL** pointer. C++ allows us to **delete** a **NULL** pointer. The third

case deletes an uninitialised pointer. This is a very severe error and will generally generate a run-time application error, because an unknown random block of memory will be deleted.

Just as the use of the **new** operator causes a **class**'s constructor to be called automatically, the **delete** operator automatically executes a **class**'s destructor. The **Point** **class** above did not define a destructor, so let's now examine a **String** **class** which incorporates both constructor and destructor member functions. The following **String** **class** also introduces the use of the **new** operator with arrays, which we shall examine in more detail in the next subsection.

A **String** **class** was introduced in Chapter 10, program STRING.CPP, with reference to operator overloading:

```
// string.cpp
// illustrates a String class
//...
const int STR_LEN = 80 ; // maximum length of String

class String
{
private:
    char string[STR_LEN] ;
public:
    // constructor
    String (char s[]={"\0"})
        { strcpy (string, s) ; }
    //...
};
```

The **class** encapsulates a string as an array of 80 characters independent of the actual string length. The string length cannot exceed 80 characters, and if a user of the **class** defines a String object of length less than 80 characters memory space is wasted. The **new** and **delete** operators can make the **String** **class** more dynamic:

```
// nd_str.cpp
// illustrates the use of the new and delete
// operators with a String class
#include <iostream.h> // C++ I/O
#include <string.h> // strlen(), strcpy()
#include <assert.h> // assert()

class String
{
private:
    char* string ; // pointer to char
    int length ; // length of string
public:
    // constructor
    String (const char str[]={}) ;
    // destructor
    ~String () ;
    // friend
    friend ostream& operator << (ostream& s,
```

```

                                const String& str) ;
}; // String class

// 1 arg. constructor
String::String (const char str[])
{
if (str == NULL) // NULL string
{
    length = 0 ;
    string = NULL ;
}
else           // get length, allocate, test, copy
{
    length = strlen (str) ;
    string = new char[length+1] ;
    assert (string != NULL) ;
    strcpy (string, str) ;
}
cout << "constructor called!" << endl ;
}

// destructor
String::~String ()
{
delete [] string ; // deallocate memory

string = NULL ;      // set string pointer to NULL
length = 0 ;         // set string length to zero

cout << endl << "destructor called!" ;
}

// friend:

// overloaded insertion operator
inline ostream& operator << (ostream& s, const String& str)
{
s << str.string ;
return s ;
}

void main ()
{
String s = "string" ;

cout << s ;
}

```

with output:

constructor called!

---

```

string
destructor called!

class String has two private data members, string and length:
```

```

class String
{
private:
    char* string ; // pointer to char
    int length ; // length of string
//...
};
```

The data member `string` is a pointer to `char` and will point to an array of characters held by an object of `class String`. If a user-specified string is not of zero length then `String`'s one-argument constructor first determines the length of the string, requests sufficient memory to hold the string plus the null terminator, verifies that the pointer returned from the `new` operator is not the NULL pointer and finally copies the user-specified string to the `string` data member. Note the declaration of the `String::String()` constructor in the `class` declaration of `String`:

```
String (const char str[] = 0) ;
```

The argument is passed using array notation, rather than pointer notation, so that uninitialised `String` objects (null strings) can be defined without having to define a zero argument default `String` constructor explicitly.

`String`'s destructor simply deallocates the memory allocated from `String`'s constructor and sets the `string` and `length` data members to the NULL pointer and zero respectively.

Note the use of the `assert()` macro in the constructor:

```
assert (string != NULL) ;
```

Programs `CAST_P.CPP`, `ND_OBJ.CPP` and `DELETE.CPP` tested, using an `if`-statement, the pointer returned from the `new` operator against the NULL pointer. If the memory request was unsuccessful then the programs would first display an error message and then abort. Alternatively, C++ provides a library macro called `assert()` which performs a similar task:

```
void assert (int test) ; // ASSERT.H
```

The `assert()` macro tests the specified condition in the `test` argument, and if the `test` expression evaluates to zero the macro displays the error message 'Assertion failed: test, file filename, line linenum' on `stderr`<sup>7</sup> and aborts the program by calling the `abort()` function, where `filename` and `linenum` are the filename of the source code and line number where the `assert()` macro appears, respectively. If you place the `#define NDEBUG` directive (i.e. 'no debugging') in the source code before the `#include <assert.h>` preprocessor directive, the `assert()` macro statement is effectively commented out.

The `abort()` function signature is:

---

<sup>7</sup> `stderr` is the standard error output device and is #defined in the `STDIO.H` header file.

```
void abort () ; // STDLIB.H
```

`abort()` terminates a program in an abnormal way and displays the run-time message 'Program Aborted' on `stderr`.

Earlier we saw arrays of pointers to C++ integral types. We can also define arrays of pointers to objects:

```
// a_p_str1.cpp
// illustrates arrays of pointers to String objects
//...
class String
{
private:
    char* string ; // pointer to char
    int length ; // length of string
public:
    //...
    friend istream& operator >> (istream& s, String& str) ;
}; // String class
//...
// overloaded extraction operator
istream& operator >> (istream& s, String& str)
{
    char buffer[80] ; // temporary buffer

    s >> buffer ; // get string

    str.length = strlen (buffer) ;
    delete [] str.string ;
    str.string = new char[str.length+1] ;
    assert (str.string != NULL) ;
    strcpy (str.string, buffer) ;
    return s ;
}

void main ()
{
    const int SIZE = 5 ;

    String* p_str[SIZE] ; // array of pointers to String

    // I/P
    for (int i=0; i<SIZE; i++)
    {
        // allocate memory for new object
        p_str[i] = new String ;
        assert (p_str[i] != NULL) ;
        cout << "enter a string: " ;
        cin >> *p_str[i] ;
    }
    // O/P
```

---

```
for (int j=0; j<SIZE; j++)
    cout << *p_str[j] << endl ;
}
```

Note the addition of the overloaded extraction operator (`>>`) to the String **class**. The statement:

```
String* p_str[SIZE] ;
```

defines an array of pointers, `p_str`, to **class** String with SIZE elements. The statement:

```
p_str[i] = new String ;
```

requests sufficient memory for a String object and assigns the address of the object to the `i`th element of array `p_str`.

## 12.26.2 Arrays

The previous subsection demonstrated that C++ allows an array of objects to be allocated using the **new** operator. The general syntax is of the form:

```
pointer_variable = new class_name [ARRAY_SIZE] ;
```

This statement requests a block of memory sufficient in size to hold an array of ARRAY\_SIZE elements of **class** or type `class_name` and assigns the memory address of the block of memory to the pointer `pointer_variable`. Unlike the object form of the **new** operator, the array form does not allow initialisation arguments to be supplied when an array of objects is created using **new**. As a consequence, arrays of objects can only be created using the **new** operator with a default constructor.

To free an array that was allocated dynamically the **delete** operator adopts the syntax:

```
delete [] pointer_variable ;
```

The subscript operator, `[]`, informs the compiler that a block of memory associated with an array is to be deallocated.

To illustrate the use of the **new** and **delete** operators with arrays let's design a Vector **class**. Vectors, consisting of three components, are frequently used in computer graphics for modelling objects in a three-dimensional space. However, the term vector can be applied more generally to describe a list of elements of arbitrary size. For instance, the following illustrates an array or matrix of elements  $a_{ij}$  ( $i=1,2,\dots,m$ ;  $j=1,2,\dots,n$ ):

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

The horizontal lines of the matrix are referred to as rows, or *row vectors*, and the vertical lines are columns, or *column vectors*, and are conventionally written:

$$[a_1, a_2, \dots, a_n], \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

Thus, a Vector **class** must be sufficiently dynamic to cater for an arbitrary size of row or column vectors. The following program presents a first implementation of a Vector **class**:

```
// vect.cpp
// a Vector class that illustrates the use of the
// new and delete operators for dynamically allocating
// and deallocating memory for arrays
#include <iostream.h> // C++ I/O
#include <assert.h> // assert()
#include <mem.h> // memset()
#include <stdlib.h> // exit()

class Vector
{
private:
    // data members
    double* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector ()
        : array (NULL), n_elements (0) {}
    Vector (int n) { Allocate (n) ; }
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                  n_elements = 0 ; }
    // overloaded operators
    double& operator [] (int index) ;
    const double& operator [] (int index) const ;
    // friends
    friend ostream& operator << (ostream& s,
                                      const Vector& v) ;
}; // class Vector

// private member function:

// allocates memory for elements of vector
void Vector::Allocate (int n)
{
    if (n <= 0)
    {
        cout << "zero or negative number of elements" ;
    }
}
```

```
    exit (EXIT_SUCCESS) ;
}
array = new double[n] ;
assert (array != NULL) ;
n_elements = n ;
// zero array
memset ((void*)array, 0, (size_t)n*sizeof(double)) ;
}

// overloaded operators:

// subscript operator [] (non-const objects)
double& Vector::operator [] (int index)
{
if (index < 0 || index >= n_elements)
{
cout << "indexing Vector element out of range" ;
exit (EXIT_SUCCESS) ;
}
return array[index] ;
}

// subscript operator [] (const objects)
const double& Vector::operator [] (int index) const
{
if (index < 0 || index >= n_elements)
{
cout << "indexing Vector element out of range" ;
exit (EXIT_SUCCESS) ;
}
return array[index] ;
}

// friend:

// overloaded insertion operator <<
ostream& operator << (ostream& s, const Vector& v)
{
s << "[" ;
for (int i=0; i<v.n_elements; i++)
{
s << v.array[i] ;
i!=v.n_elements-1 ? s << ", " : s << "]" ;
}
return s ;
}

void main ()
{
Vector a (3), b (5) ;
```

```

cout << "a: " << a << endl ;
cout << "b: " << b << endl ;

a[0] = 1.0 ; a[1] = 2.0 ; a[2] = 3.0 ;

cout << "a: " << a << endl ;
}

```

with output:

```

a: [0, 0, 0]
b: [0, 0, 0, 0, 0]
a: [1, 2, 3]

```

The Vector **class** encapsulates two **private** data members:

```

class Vector
{
private:
    // data members
    double* array ;
    int      n_elements ;
    //...
};

```

array is a pointer to **double** which will be assigned the memory address of an array of elements of type **double** that refer to the n\_elements elements of a vector. Vector's one-argument constructor calls the *Allocate()* **private** member function:

```
Vector (int n) { Allocate (n) ; }
```

The default no-argument constructor simply sets array to NULL and n\_elements to zero. The one-argument constructor allocates sufficient memory for n elements. Memory is allocated dynamically by calling the **private** *Allocate()* function:

```

void Vector::Allocate (int n)
{
if (n <= 0)
{
    cout << "zero or negative number of elements" ;
    exit (EXIT_SUCCESS) ;
}
array = new double[n] ;
assert (array != NULL) ;
n_elements = n ;
// zero array
memset ((void*)array, 0, (size_t)n*sizeof(double)) ;
}

```

Note that *Allocate()* is declared a **private** member function of **class** Vector to prevent a user from accidentally calling the function. Memory is allocated for an array of n

**double** elements. The pointer returned from **new** is tested against the NULL pointer, and if the memory request is successful the number of elements, **n\_elements**, in the array is set.

Since C++ does not guarantee that the elements of an array are set to zero (or any other value or object) it is necessary to *zero-out* the array. The obvious way to zero-out an array is to simply use a **for**-loop:

```
for (int i=0; i<n_elements; i++)
    array[i] = 0.0 ;
```

However, this is not very efficient. A faster way is to call the *memset ()* function:

```
void* memset (void* s, int c, int n) ; // MEM.H
```

*memset ()* sets the first **n** bytes of memory pointed to by **s** to the character **c**. The return value of *memset ()* is **s**. The **Vector** : : *Allocate ()* function calls *memset ()* to initialise the elements of array to zero.

Pay particular attention to the above declaration of the **Vector** : : *array* data member and memory allocation using the **new** operator:

```
class Vector
{
private:
    double* array ;
    //...
};

//...
void Allocate (int n)
{
    //...
    array = new double[n] ;
    //...
}
```

Alternatively, the following is a common mistake for which the compiler will not issue a compilation or linker warning or error message:

```
//...
void Allocate (int n)
{
    //...
    double* array = new double[n] ;
    //...
}
```

which redefines **array**. This will result in a disastrous run-time error when an attempt is made to access the unallocated elements of a **Vector** object.

**Vector**'s destructor frees the memory allocated to an object, assigns the NULL pointer to the **array** data member and sets the **n\_elements** data member to zero:

```
~Vector () { delete [] array ; array = NULL ; n_elements = 0 ; }
```

The subscript, [ ], and insertion, <<, operators are overloaded for **class** Vector so as to enable natural indexing and output of a Vector's elements. The return type of both of the overloaded **operator[]()** member functions is a reference to **double**. Returning a reference allows the subscript operator to appear on either the left- or right-hand side of the assignment operator:

```
Vector a ;
//...
double d = a[0] ;
a[1] = 2.0 ;
```

Now suppose that we would like to increase or decrease the number of elements of a Vector object once it has been defined. The following program adds a *New()* member function to **class** Vector which performs just such a task:

```
// vect1.cpp
// a Vector class that caters for an
// increasing/decreasing number of elements
//...
class Vector
{
public:
    //...
    // member function
    void New (int new_n) ;
    //...
}; // class Vector

// change number of elements
void Vector::New (int new_n)
{
    delete [] array ;
    array      = NULL ;
    n_elements = 0 ;

    Allocate (new_n) ;
}
//...
void main ()
{
    Vector a ;
    cout << "original a: " << a << endl ;
    a.New (4) ; // add an element
    a[0] = 1.0 ; a[1] = 2.0 ; a[2] = 3.0 ; a[3] = 4.0 ;
    cout << "new a      : " << a << endl ;
}
```

Firstly, *New()* **Deletes** the memory allocation associated with an object's original array of elements and then sets the object's **array** and **n\_elements** data members to **NULL** and **0** respectively. Then *New()* simply calls *Allocate()* to allocate sufficient memory for the new size of a Vector object. At present, the *New()* member function is not very smart, since it

deletes the existing array of elements of the `Vector` object that calls the function. However, we shall refrain from continuing to extend `class Vector` until we discuss templates in Chapter 13.

`VECT1.CPP` contains some additional features, not shown above, which will be discussed later in connection with returning ‘local’ objects from member functions.

### 12.26.3 Objects, Arrays and `delete`

The previous two subsections have illustrated separately the use of the `delete` operator for both objects and arrays of objects, but under general circumstances how does a compiler or programmer know whether a pointer points to an object or an array of objects? Consider the following program code:

```
//...
void Function (X* obj) // obj may point to an object or an
                      // array
{
//...
delete obj ;      // ?
delete [] obj ;   // ?
}
```

It is not possible to determine from a pointer whether it points to an object or an array of objects. It is the programmer’s responsibility to ensure that the correct syntax is used when using the `delete` operator – getting it wrong can have disastrous results!

### 12.26.4 Data Member Initialisation Lists and `new`

Previous sections have demonstrated the use of the `new` operator for allocating memory for data member objects and arrays of objects:

```
class X
{
private:
    int* data ;
    int* array ;
public:
    X ()
        { data = new int ; array = new int[5] ; }
//...
};
```

Alternatively, the `new` operator can be placed in a `class`’s constructor data member initialisation list:

```
// new_init.cpp
// illustrates the combined use of the new operator
// with a class's constructor data member initialisation
// list
#include <iostream.h> // C++ I/O
```

```

class X
{
private:
    int* data ;
    int* array ;
public:
    X ()
        : data (new int) , array (new int[5]) {}
    X (int n)
        : data (new int(n)) , array (new int[n]) {}
    ~X ()
        { delete data ; delete [] array ; }
};

void main ()
{
}

```

### 12.26.5 Overloaded **new** and **delete** Operators

Since **new** and **delete** are operators it is possible to overload them specifically for a given **class**. The general syntax of overloading<sup>8</sup> the **new** and **delete** operators for objects and arrays of objects is:

```

void* operator new (size_t size)
{
//...
}
void* operator new[] (size_t size)
{
//...
}
void operator delete (void* ptr)
{
//...
}
void operator delete[] (void* ptr)
{
//...
}

```

The overloaded **new** operator function must return a pointer to the memory address of the block of memory that it allocates if allocation is successful or the NULL pointer if the memory allocation is unsuccessful. The overloaded **delete** operator function is passed a pointer to the memory address of the block of memory to be deallocated.

To see overloading of the **new** and **delete** operators in action, let's revisit the **Vector** **class** discussed above:

---

<sup>8</sup> You may want to refer to Chapter 10 to remind yourself of operator overloading.

```
// over_nd.cpp
// illustrates overloading the new and delete operators
//...
class Vector
{
//...
public:
    // constructors
    //...
    // destructor
    ~Vector () { delete [] array ;
        array = NULL,
        n_elements = 0 ;
        cout << "destructor called" << endl ; }

    //...
    // overloaded operators
    //...
    void* operator new (size_t size) ;
    void operator delete (void* ptr) ;
    void* operator new[] (size_t size) ;
    void operator delete[] (void* ptr) ;
    //...
}; // class Vector

// private member function:

// allocates memory for elements of vector
void Vector::Allocate (int n)
{
    cout << "constructor called" << endl ;
    //...
}
//...
// overloaded new operator for object
void* Vector::operator new (size_t size)
{
    cout << "object overloaded new operator called" << endl ;
    return malloc (size) ;
}

// overloaded delete operator for object
void Vector::operator delete (void* ptr)
{
    cout << "object overloaded delete operator called" << endl ;
    free (ptr) ;
}

// overloaded new operator for an array of objects
void* Vector::operator new[] (size_t size)
{
    cout << "array overloaded new operator called" << endl ;
```

```

return malloc (size) ;
}

// overloaded delete operator for an array of objects
void Vector::operator delete[] (void* ptr)
{
    cout << "array overloaded delete operator called" << endl ;
    free (ptr) ;
}
//...
void main ()
{
    Vector* v_ptr (NULL) ;
    Vector* array_v ;

    // allocate object and array of objects
    v_ptr = new Vector ;
    array_v = new Vector[2] ;

    cout << *v_ptr << endl ;

    // deallocate object and array of objects
    delete v_ptr ;
    delete [] array_v ;

    cout << endl ;

    Vector* v1_ptr = ::new Vector (2) ; // global allocate
    cout << *v1_ptr << endl ;
    ::delete v1_ptr ; // global deallocate
}

```

with output:

```

object overloaded new operator called
constructor called
array overloaded new operator called
constructor called
constructor called
[0, 0, 0]
destructor called
object overloaded delete operator called
destructor called
destructor called
array overloaded delete operator called

constructor called
[0, 0]
destructor called

```

The **new** and **delete** operators are overloaded for objects of the Vector **class** as follows:

---

```

void* Vector::operator new (size_t size)
{
    cout << "object overloaded new operator called" << endl ;
    return malloc (size) ;
}

void Vector::operator delete (void* ptr)
{
    cout << "object overloaded delete operator called" << endl ;
    free (ptr) ;
}

```

For illustration purposes the **new** and **delete** operators, for both objects and an array of objects, simply call the C++ memory manipulation library functions *malloc()* and *free()*, respectively. Note that if a memory allocation request is unsuccessful or the **size** argument is zero the *malloc()* function returns the NULL pointer.

The program output illustrates that the statement:

```
v_ptr = new Vector ;
```

first calls the object overloaded **new** operator and then calls Vector's default constructor. It is important to note that use of the **new** operator will result in an object's appropriate constructor being called automatically. Similarly, but in reverse order to **new**, the statement:

```
delete v_ptr ;
```

will first cause Vector's destructor to be called automatically and then the object's overloaded **delete** operator. Also, the program output illustrates that a **class**'s constructor and destructor are called automatically for each object of an array of objects.

The final three statements within the *main()* function of program OVER\_ND.CPP are:

```

Vector* v1_ptr = ::new Vector (2) ; // global allocate
cout << *v1_ptr << endl ;
::delete v1_ptr ;                      // global deallocate

```

If the **new** and **delete** operators are specifically overloaded for a given **class**, use of **new** and **delete** will call the overloaded member functions **operator new()** and **operator delete()**, respectively. If you require that the global **::operator new()** and **::operator delete()** operators be used then precede the operator by the scope resolution operator **(::)**. The program output illustrates that **::new** and **::delete** automatically call an object's constructor and destructor (as usual), but do not call the overloaded **operator new()** and **operator delete()** operators.

An important point to note when overloading the **new** and **delete** operators is that, if they are overloaded for a given **class**, use of **new** and **delete** on any other type or **class** will result in the default or global **new** and **delete** operators being called. Thus, in the above program, OVER\_ND.CPP, the following statement in *main()*:

```
v_ptr = new Vector ;
```

calls Vector::**operator new()**, but the call to **new** in the *Allocate()* function:

---

```
void Vector::Allocate (int n)
{
//...
array = new double[n] ;
//...
}
```

calls the global **new** operator!

For those readers interested in programming for Windows, the following overloaded **new** and **delete** operators allocate and deallocate, respectively, virtual memory from a disk file:

```
void FAR* Vector::operator new (size_t size)
{
// amount of global memory available
DWORD dwAvailable = GlobalCompact (0) ;

// request greater than available
if (size > dwAvailable)
{
    MessageBox (GetActiveWindow (),
        (LPCSTR) "insufficient memory",
        (LPCSTR) "Vector Warning",
        MB_ICONEXCLAMATION | MB_OK) ;
    return NULL ;
}

// handle to global memory
HGLOBAL hGlobalMemory =
    GlobalAlloc (GMEM_MOVEABLE | GMEM_ZEROINIT, size) ;

// if unsuccessful allocation
if (!hGlobalMemory)
{
    MessageBox (GetActiveWindow (),
        (LPCSTR) "insufficient memory",
        (LPCSTR) "Vector Warning",
        MB_ICONEXCLAMATION | MB_OK) ;
    return NULL ;
}

// void far pointer to global memory
void FAR* lpvGlobalMemory = GlobalLock (hGlobalMemory) ;

// return pointer to allocated memory
return lpvGlobalMemory ;
}

void Vector::operator delete (void* obj)
{
    GlobalUnlock (GlobalHandle ((HGLOBAL)obj)) ; // unlock
}
```

---

```
GlobalFree (GlobalHandle ((HGLOBAL)obj)) ; // free
}
```

The Windows Application Programming Interface (API) function *GlobalCompact()* returns the size of the largest block of memory available:

```
DWORD GlobalCompact (DWORD dwMinFree) ;
```

*dwMinFree* is the size, in bytes, of the requested memory block. If *dwMinFree* is zero the return value of *GlobalCompact()* is the largest block of free memory available. *GlobalCompact()* determines the size of the largest block of memory by moving moveable blocks and memory that could be obtained if discardable blocks were discarded. The double-word identifier *DWORD* is defined as (*WINDEF.H*):

```
typedef unsigned long DWORD ; // WINDEF.H
```

The *MessageBox()* function declaration is:

```
int MessageBox (HWND hwndParent, LPCSTR lpszText,
                 LPCSTR lpszTitle, UINT fuStyle) ;
```

where *hwndParent* is a handle to the parent window, *lpszText* is the memory address of the text to be displayed in the message box, *lpszTitle* is the memory address of the text to be displayed in the title of the message box and *fuStyle* is the style of the message box.

The *GlobalAlloc()* function allocates the specified number of bytes, *dwAlloc*, from the global heap:

```
HGLOBAL GlobalAlloc (UINT fuAlloc, DWORD dwAlloc) ;

typedef const void NEAR* HANDLE ; // WINDOWS.H
//...
typedef HANDLE HGLOBAL ;
```

*fuAlloc* is an identifier that specifies how the memory is to be allocated. *GMEM\_MOVEABLE* allocates moveable memory and *GMEM\_ZEROINT* initialises the contents of the memory allocated to zero. *GlobalAlloc()* returns a handle to the allocated memory block if the request is successful; if not, it returns the NULL pointer. After receiving a handle to a global block of memory it is then necessary to lock the block of memory:

```
void FAR* GlobalLock (HGLOBAL hglb) ;
```

Locking a block of memory is essential so that it will not be moved or discarded unless the memory is explicitly unlocked or reallocated by the Windows API *GlobalReAlloc()* function. The return value of *GlobalLock()* is **void** *FAR\**. A far pointer to **void** is necessary because memory is being allocated in segments other than the default segment. A far pointer occupies four bytes of memory. The low-order two bytes are the *offset*, while the high-order two bytes are the *selector*.

The overloaded **delete** operator first unlocks the block of memory allocated by **operator new()** and then frees the block of memory, assuming that the block of memory is not locked. The function signatures of *GlobalUnlock()* and *GlobalLock()* are:

```
BOOL GlobalUnlock (HGLOBAL hglb) ;
```

```
HGLOBAL GlobalFree (HGLOBAL hglb) ;
```

*GlobalFree()* invalidates the handle *hglb*.

An alternative to overloading the **new** and **delete** operators specifically for each user-defined **class** is to implement a memory **class**. Staying with global memory in a Windows environment, the following is a **class** which encapsulates global memory allocation:

```
// win_mem.cpp
// illustrates a global memory class
// for programming in a Windows environment
#include <iostream.h> // C++ I/O
#include <windows.h> // Windows header file

class GlobalMemory
{
private:
    HGLOBAL hGlobalMemory ; // handle to memory
    void FAR* mem_address ; // address of memory
public:
    // constructor
    GlobalMemory ()
        : hGlobalMemory (NULL), mem_address (NULL) {} ;
    // destructor
    ~GlobalMemory () ;
    // member functions
    DWORD SizeOfBlock () ;
    void FAR* MemoryAddress () const { return mem_address ; }
    void FAR* Allocate (UINT size,
                        UINT mem_flags=GMEM_MOVEABLE|GMEM_ZEROINIT) ;
    void FAR* ReAllocate (UINT size,
                         UINT mem_flags=GMEM_MOVEABLE|GMEM_ZEROINIT) ;
};

// destructor
GlobalMemory::~GlobalMemory ()
{
    GlobalUnlock (hGlobalMemory) ; // unlock
    if (hGlobalMemory) // free
        GlobalFree (hGlobalMemory) ;
}

// return size of memory block
DWORD GlobalMemory::SizeOfBlock ()
{
    if (!hGlobalMemory)
        return 0 ;
    else
        return GlobalSize (hGlobalMemory) ;
}
```

```
// allocate memory
void FAR* GlobalMemory::Allocate (UINT size, UINT mem_flags)
{
    hGlobalMemory = GlobalAlloc (mem_flags, size) ;

    // if unsuccessful allocation
    if (!hGlobalMemory)
    {
        MessageBox (GetActiveWindow (),
                    (LPCSTR) "insufficient memory",
                    (LPCSTR) "GlobalMemory Warning",
                    MB_ICONEXCLAMATION | MB_OK) ;
        hGlobalMemory = NULL ;
        return NULL ;
    }
    mem_address = GlobalLock (hGlobalMemory) ;
    return mem_address ;
}

// re-allocate memory
void FAR* GlobalMemory::ReAllocate (UINT size,
                                    UINT mem_flags)
{
    HGLOBAL h = GlobalReAlloc (hGlobalMemory, size,
                               mem_flags) ;

    // if unsuccessful allocation
    if (!h)
    {
        MessageBox (GetActiveWindow (),
                    (LPCSTR) "insufficient memory",
                    (LPCSTR) "GlobalMemory Warning",
                    MB_ICONEXCLAMATION | MB_OK) ;
        return NULL ;
    }
    else
    {
        hGlobalMemory = h ;
        mem_address = GlobalLock (hGlobalMemory) ;
    }
    return mem_address ;
}

void main ()
{
    const int SIZE = 10 ;

    GlobalMemory g_mem ; // global memory object

    // allocate memory
    int* px = (int*) g_mem.Allocate (SIZE*sizeof(int)) ;
```

```

// set
for (int i=0; i<SIZE; i++)
    *(px+i)=i ;

// get
for (int j=0; j<SIZE; j++)
    cout << *(px+j) << " " ;
cout << endl ;
cout << "size: " << g_mem.SizeOfBlock () << endl ;

// re-allocate memory
px = (int*) g_mem.ReAllocate (2*SIZE*sizeof(int)) ;

// get
for (int k=0; k<2*SIZE; k++)
    cout << *(px+k) << " " ;
cout << endl ;
cout << "size: " << g_mem.SizeOfBlock () << endl ;
}

```

with output:

```

0 1 2 3 4 5 6 7 8 9
size: 32
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
size: 64

```

The GlobalMemory **class** declares two **private** data members:

```

class GlobalMemory
{
private:
    HGLOBAL hGlobalMemory ; // handle to memory
    void FAR* mem_address ; // address of memory
    //...
};

```

which define the handle and memory address of a block of global memory. GlobalMemory's constructor initialises its two data members to NULL, while its destructor unlocks and frees an allocated global block of memory. GlobalMemory contains four member functions. The access member function *MemoryAddress()* simply returns the memory address of the block of memory. *SizeOfBlock()* returns the size, in bytes, of a global memory object and makes use of the API function *GlobalSize()*:

```
DWORD GlobalSize (HGLOBAL hglb) ;
```

The first call to *SizeOfBlock()* in *main()* displays a size of 32 bytes for the global memory object *g\_mem*. The size of the memory object returned by *GlobalSize()* is greater than the size of 20 bytes requested by *GlobalAlloc()*. If the amount of memory allocated is greater than the amount requested, Windows guarantees that the object has access to the allocated memory. The *Allocate()* member function requests a block of memory from the operating

system and returns a far pointer to the memory. The *ReAllocate()* member function makes use of the API function *GlobalReAlloc()* to alter the size and attributes of the memory object:

```
HGLOBAL GlobalReAlloc (HGLOBAL hglb, DWORD cbNewSize,
                      UINT fuAlloc) ;
```

If *GlobalReAlloc()* is successful at reallocating the memory associated with the handle to the memory object, *hglb*, the function returns a handle to the object, else NULL is returned. The data held in the block of memory pointed to by the memory object are preserved in the reallocation. This is illustrated in the program output of WIN\_MEM.CPP above. Although the memory object *g\_mem* is increased from a requested 20 to 40 bytes, the original 10 integers (0:9) are preserved. Note that the default argument *mem\_flags* in *ReAllocate()* initialises the data of the newly allocated memory object to zero.

You have probably gathered from the above brief discussion that memory management in Windows is a big topic and beyond the scope of this text. However, the above discussion does illustrate that the ***new*** and ***delete*** operators can be overloaded to suit the exact needs of your own classes and operating system, or alternatively that a memory ***class*** can be implemented which is tailored exactly to different types of memory. We shall revisit the subject of memory management when we discuss templates and inheritance.

For the reader interested in C programming for Windows I recommend the two introductory books of Lafore (1993) and Petzold (1992).

## 12.26.6 Placement

C++ allows an object to be placed at a memory address specified by the programmer using the *placement syntax* of the ***new*** operator:

```
// placemnt.cpp
// illustrates placement
#include <iostream.h> // C++ I/O

class X
{
private:
    int data ;
public:
    X ()
        : data (0) {}
    X (int i)
        : data (i) {}
    void* operator new (size_t size, void* ptr)
    {
        return ptr; // place object at specified location
    }
};

void main ()
{
    int i (0), * i_ptr = &i ;
```

```

void* location = (void*)i_ptr ; // set to valid memory
                                // address
X* x_ptr = new (location) X (1) ; // place

x_ptr->~X () ; // remove

// int* j_ptr = new (location) int (1) ; // error
}

```

The overloaded **operator new()** function of **class X** places an object of **class X** at a memory position pointed to by the pointer argument **ptr**. Placing an object of **class X** at a specified location memory address now requires the **new (location) X** syntax with the **size\_t** argument **size** passed implicitly.

To remove an object of **class X** pointed to by **x\_ptr** from a specified memory address (which was allocated via the **new** placement syntax) the following syntax is used:

```
x_ptr->~X () ;
```

## 12.27 Returning Local Objects from Member Functions

When an object is returned by a function a temporary object is automatically created by the compiler to hold the object. It is this temporary object which is returned by a function. When the object has been returned by the function the temporary object is destroyed. This can cause some very serious run-time errors:

```

// ret_prob.cpp
// illustrates a problem with returning
// an object from a function
#include <iostream.h> // C++ I/O

class Object
{
private:
    int* ptr ;
public:
    Object () ;
    ~Object () ;
    int Get () const { return *ptr ; }
}; // class Object

Object::Object ()
{
    ptr = new int (0);
    cout << "constructor called" << endl ;
}

Object::~Object ()
{
    delete ptr ;
}

```

```

cout << "destructor called" << endl ;
}

// return a copy
Object ReturnCopy (const Object& obj)
{
    Object return_obj ;
    return_obj = obj ;
    return return_obj ;
}

void main ()
{
    Object object1, object2 ;

    object2 = ReturnCopy (object1) ;
}

```

with output:

```

constructor called
constructor called
constructor called
destructor called
destructor called
destructor called
destructor called

```

The output illustrates that there are three constructor calls but four destructor calls! Examine the function *ReturnCopy()*:

```

Object ReturnCopy (const Object& obj)
{
    Object return_obj ;
    return_obj = obj ;
    return return_obj ;
}

```

*Object*'s constructor is called for each of the two objects, *object1* and *object2*, in *main()* and for the 'local' object, *return\_obj*, in *ReturnCopy()*. *Object*'s destructor is first called when *return\_obj* goes out of scope (i.e. when *ReturnCopy()* returns) and called a second time when the temporary object returned by *ReturnCopy()* is destroyed. Finally, *Object*'s destructor is called twice when objects *object1* and *object2* are destroyed.

Note that *Object*'s constructor is not called when a temporary object is created in function *ReturnCopy()* since this would reinitialise the object being copied. One solution to this problem is to return an object by reference or pointer. However, this is not always possible. For instance, objects 'local' to a function cannot be returned by reference:

```

Object& ReturnCopy (const Object& obj)
{

```

---

```
//...
return return_obj ;
}
```

An alternative solution involves the use of a *copy constructor*. It was noted in Chapter 9 that a copy constructor is called for creating a temporary that is to be returned from a function. Try adding the following copy constructor to program RET\_PROB.CPP:

```
Object::Object (const Object& obj)
{
    ptr = new int ;
    *ptr = obj.Get () ;
    cout << "copy constructor called" << endl ;
}
```

The program output will now be:

```
constructor called
constructor called
constructor called
copy constructor called
destructor called
destructor called
destructor called
destructor called
```

which illustrates that the temporary object returned by *ReturnCopy()* invokes Object's copy constructor, which allocates memory accordingly and makes an exact copy of the Object::ptr data member.

Let's now revisit the VECT1.CPP program, which illustrates a practical application of the copy constructor:

```
// vect1.cpp
//...
class Vector
{
private:
    // data members
    double* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector () { Allocate (3) ; }
    Vector (int n) { Allocate (n) ; }
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ;
                 array = NULL ; n_elements = 0 ; }
```

```

// member functions
int NumberElements () const
{ return n_elements ; }

double& Value (int index) ;
const double& Value (int index) const ;
void New (int new_n) ;
// overloaded operators
Vector& operator = (const Vector& v) ;
double& operator [] (int index) ;
const double& operator [] (int index) const ;
Vector operator + (const Vector& v) ;
// friends
friend ostream& operator << (ostream& s,
                                const Vector& v) ;

}; // class Vector
//...
// copy constructor
Vector::Vector (const Vector& v)
{
    Allocate (v.NumberElements()) ;
    for (int i=0; i<v.NumberElements(); i++)
        array[i] = v.Value (i) ;
}

// public member functions:

// returns element value (non-const object)
double& Vector::Value (int index)
{
    assert (index >= 0 || index < n_elements) ;
    return array[index] ;
}

// returns element value (const object)
const double& Vector::Value (int index) const
{
    assert (index >= 0 || index < n_elements) ;
    return array[index] ;
}
//...
// overloaded operators:

// assignment operator =
Vector& Vector::operator = (const Vector& v)
{
    // verify that the two vectors are of equivalent
    // dimensions
    assert(n_elements == v.NumberElements()) ;

    for (int i=0; i<v.NumberElements(); i++)
        Value (i) = v.Value (i) ;
}

```

```

    return *this ;
}
//...
// addition operator +
Vector Vector::operator + (const Vector& v)
{
// verify that the two vectors are of equivalent dimensions
assert(n_elements == v.NumberElements()) ;

Vector add (v.NumberElements()) ;

for (int i=0; i<v.NumberElements(); i++)
    add.Value (i) = Value (i) + v.Value (i) ;
return add ;
}
//...
void main ()
{
//...
Vector v1, v2, v3 ;
v3 = v1 + v2 ; // run-time error if copy-constructor removed
}

```

The copy constructor definition is:

```

Vector::Vector (const Vector& v)
{
Allocate (v.NumberElements()) ;
for (int i=0; i<v.NumberElements(); i++)
    array[i] = v.Value (i) ;
}

```

which calls `Vector::Allocate()` to allocate sufficient memory for the `Vector` object, `v`, being copied and then proceeds to make an exact copy of the array of `Vector` elements. If the copy constructor is removed from `VECT1.CPP`, the overloaded `Vector::operator + ()` function:

```

Vector Vector::operator + (const Vector& v)
{
//...
Vector add (v.NumberElements()) ;
//...
return add ;
}

```

will generate a run-time error.

## 12.28 Passing Objects to Functions

The `VECT1.CPP` program, described in the previous section, illustrated problems that can arise when returning a 'local' object from a member function. Similarly, when constructors and a

destructor are defined for a given **class**, passing objects by value to a function can cause unexpected results. To illustrate problems that occur when objects are passed by value to a function, consider the program:

```
// pas_prob.cpp
// illustrates a problem with passing
// an object by value to a function
#include <iostream.h> // C++ I/O

class Object
{
private:
    int* ptr ;
public:
    Object (int i) ;
    ~Object () ;
    int Get () const { return *ptr ; }
}; // class Object

Object::Object (int i)
{
    ptr = new int (i);
    cout << "constructor called" << endl ;
}

Object::~Object ()
{
    delete ptr ;
    cout << "destructor called" << endl ;
}

// pass by value
void Display (Object obj)
{
    cout << obj.Get () << endl ;
}

void main ()
{
    Object object (1) ;

    Display (object) ;
}
```

with output:

```
constructor called
1
destructor called
destructor called
```

The program output illustrates that `Object`'s constructor is called once but `Object`'s destructor is called twice. When `object` is defined in `main()`, `Object`'s constructor is automatically called and allocates memory for the data member `object.ptr`. When `object` is passed by value to `Display()` a copy is made and assigned to `obj`. Note that although a copy of an object is made when passed by value to a function, the object's constructor is not called. Calling the appropriate constructor when an object is passed to a function would effectively reinitialise the object – which is not what we want. Thus, when an object is passed by value to a function, the object's constructor is not called. However (and here's the problem), the function argument `object`, `obj`, `destructor` is called when the function returns, and `obj` goes out of scope. The destructor will therefore deallocate the memory associated with `obj.ptr`, which in turn is the same memory associated with `object.ptr`. Thus, when `object` is destroyed in `main()` an unknown block of memory is deleted, which will generally cause unpredictable run-time errors.

One solution to this problem is simply to pass an object to a function by reference or pointer instead of by value. When an object is passed either by reference or by pointer the original object from the calling function is accessed and no copy is made, thus eliminating a destructor call. Try modifying the `Display()` function in program PAS\_PROB.CPP so that `obj` is passed by reference:

```
void Display (const Object& obj)
{
    cout << obj.Get () << endl ;
}
```

As illustrated in the previous section, a copy constructor allows a copy of an object to allocate its own memory:

```
Object::Object (const Object& obj)
{
    ptr = new int ;
    *ptr = obj.Get () ;
    cout << "copy constructor called" << endl ;
}
```

## 12.29 The `this` Pointer

When an object calls a member function, a pointer to the object is implicitly returned to the member function. The pointer that is returned to the member function is called the `this` pointer. The `this` pointer enables a member function to know the memory address of the object that invoked the member function. The following program illustrates a use of the `this` pointer with the `Point` `class`:

```
// this.cpp
// illustrates the this pointer
//...
class Point
{
private:
    // private data members
```

```

    double x, y, z ;
public:
//...
Point* GetObjectAddress () ;
void ChangeObject (double xa, double ya, double za) ;
//...
}; // class Point

// member functions:

// return address of object invoking member function
Point* Point::GetObjectAddress ()
{
return this ;
}

// change object's data members via the this pointer
void Point::ChangeObject (double xa, double ya, double za)
{
this->x = xa ; this->y = ya ; this->z = za ;
}
//...
void main ()
{
Point p (1.0, 2.0, 3.0) ; // object

cout << "p: " << p << endl ;
cout << "address of p (&p) : "
<< &p << endl ;
cout << "address of p (this): "
<< p.GetObjectAddress () << endl ;

p.ChangeObject (4.0, 5.0, 6.0) ; // change object

cout << "p: " << p << endl ;
}

```

with output:

```

p: (1, 2, 3)
address of p (&p) : 0x454f23da
address of p (this): 0x454f23da
p: (4, 5, 6)

```

With the help of the two member functions *GetObjectAddress ()* and *ChangeObject ()*, the above program illustrates the **this** pointer. Let's examine *GetObjectAddress ()* first:

```

Point* Point::GetObjectAddress ()
{
return this ;
}

```

```
}
```

*GetObjectAddress()* simply returns the memory address of the object that invokes the function by returning the **this** pointer. The program illustrates that the memory address pointed to by **this** is indeed identical to the memory address of the object that invokes an object's member function. The **class** or type of **this** in the above *GetObjectAddress()* function is **Point\* const**. In general, the type of the **this** pointer in a non-**static** member function of a **class X** is **X\* const**. If a member function is declared as **const** or **volatile**, the type of **this** is **const X\* const** or **volatile X\* const** respectively.

The function *ChangeObject()* illustrates, although somewhat hypothetically, that the **this** pointer can be used to alter an object's data members:

```
void Point::ChangeObject (double xa, double ya, double za)
{
    this->x = xa ; this->y = ya ; this->z = za ;
}
```

Note the use of the indirect member access operator (**->**), since **this** is a pointer.

To date, the overloaded assignment operator for **class Point** has been defined as:

```
inline Point Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return Point (x, y, z) ;
}
```

which copies the data members of a **Point** object on the right-hand side of the **=** operator exactly to the respective data members of a **Point** object on the left-hand side of the **=** operator. The **operator=()** member function returns a temporary **Point** object by value by invoking **Point**'s three-argument constructor. Returning a **Point** object from **operator=()** enables the **=** operator to be chained in expressions.

The **this** pointer allows us not only to eliminate invoking **Point**'s three-argument constructor but also to return an object by reference:

```
inline Point& Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}
```

Using the dereferencing operator (\*) as a prefix operator to the **this** pointer accesses the actual object at the memory address pointed to by **this**. The following program demonstrates the use of the **this** pointer for returning **Point** objects for the overloaded **=** and the prefix and postfix **++** and **--** operators:

```
// this_pt.cpp
// illustrates the this pointer applied to class Point
//...
class Point
{
private:
```

```
// private data members
double x, y, z ;
public:
//...
// overloaded operators
Point& operator = (const Point& p) ;
Point& operator ++ () ; // prefix
Point operator ++ (int) ; // postfix
Point& operator -- () ; // prefix
Point operator -- (int) ; // postfix
//...
}; // class Point

// overloaded operators:

// assignment operator
inline Point& Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}

// prefix increment operator
inline Point& Point::operator ++ ()
{
    x++ ; y++ ; z++ ;
    return *this ;
}

// postfix increment operator
inline Point Point::operator ++ (int)
{
    Point temp = *this ;
    x++ ; y++ ; z++ ;
    return temp ;
}

// prefix decrement operator
inline Point& Point::operator -- ()
{
    x-- ; y-- ; z-- ;
    return *this ;
}

// postfix increment operator
inline Point Point::operator -- (int)
{
    Point temp = *this ;
    x-- ; y-- ; z-- ;
    return temp ;
}
```

```
//...
void main ()
{
    Point p, q (1.0, 2.0, 3.0);

    cout << "p      : " << p << endl
        << "q      : " << q << endl;

    p = q;

    cout << "(p=q) : " << p << endl;

    cout << "p++   : " << p++ << endl
        << "++q   : " << ++q << endl;
}
```

with output:

```
p      : (0, 0, 0)
q      : (1, 2, 3)
(p=q) : (1, 2, 3)
p++   : (1, 2, 3)
++q   : (2, 3, 4)
```

In line with the C++ definition of prefix and postfix increment and decrement operators, the overloaded postfix operators for **class** Point first copy the object that the operator acts on to a temporary object, increment the object's data members and then return the temporary object. The overloaded postfix operators return a Point object rather than a reference because a function cannot return a reference to a local object.

One of the key applications of the **this** pointer is in returning an object from an overloaded **operator=()** function. This was illustrated above for the Point **class** in THIS\_PT.CPP, where the data members x,y and z of **class** Point are of type **double**. Let's now examine the use of the **this** pointer and the **operator=()** function for the String **class** presented in ND\_STR.CPP, which encapsulates a pointer to a **char** data member:

```
// this_str.cpp
// illustrates the use of the this pointer with the
// operator=() function for class String
#include <iostream.h> // C++ I/O
#include <string.h> // strlen(), strcpy()
#include <assert.h> // assert()

class String
{
private:
    char* string; // pointer to char
    int length; // length of string
public:
    // constructor
    String (const char str[] = 0);
    // destructor
```

```
~String () ;
// member function
int Length () const { return length ; }
// overloaded operator
String& operator = (const String& str) ;
// friend
friend ostream& operator << (ostream& s,
                                const String& str) ;
} ; // String class

// 1 arg. constructor
String::String (const char str[])
{
if (str == NULL) // NULL string
{
    length = 0 ;
    string = NULL ;
}
else // get length, allocate, test, copy
{
    length = strlen (str) ;
    string = new char[length+1] ;
    assert (string != NULL) ;
    strcpy (string, str) ;
}
}

// destructor
String::~String ()
{
delete [] string ; // deallocate memory

string = NULL ; // set string pointer to NULL
length = 0 ; // set string length to zero
}

// overloaded operator

// assignment operator =
String& String::operator = (const String& str)
{
if (string != str.string)
{
delete [] string ;
length = str.length ;

if (str.string == NULL) // NULL string
    string = NULL ;
else
{
    string = new char[str.length+1] ;
```

```

        assert (string != NULL) ;
        strcpy (string, str.string) ;
    }
}

return *this ;
}

// friend:

// overloaded insertion operator (object)
inline ostream& operator << (ostream& s, const String& str)
{
    return s << str.string ;
}

void main ()
{
    String s1 = "a string" ;
    String s2 = "a longer string" ;

    cout << "before =: " << endl ;
    cout << "s1.string: " << s1 << ", s1.length: "
        << s1.Length () << endl ;

    s1 = s2 ;

    cout << "after =: " << endl ;
    cout << "s1.string: " << s1 << ", s1.length: "
        << s1.Length () << endl ;
}

```

with output:

```

before =:
s1.string: a string, s1.length: 8
after =:
s1.string: a longer string, s1.length: 15

```

The overloaded `String::operator=()` function first checks to see whether a `String` object is being assigned to itself. The left-hand side operand of the assignment operator pointed to by the `string` data member is then deleted, a new length assigned to the `length` data member and sufficient memory requested for the new string if it is not a NULL string. Finally, the `String` object operated on by the `operator=()` function is returned with the help of the `this` pointer.

### 12.29.1 Further Illustration of `this`

As a further illustration of the `this` pointer, and to help illustrate the explicit use of the indirection and address-of operators, consider the following program:

```
// this_exp.cpp
```

```

// experiments with the this pointer
#include <iostream.h> // C++ I/O

class X
{
private:
    char* charray ;
public:
    X ()
    { charray = "string" ; }
    void This () ;
    friend ostream& operator << (ostream& s, const X& x)
    { s << x.charray ; return s ; }
};

void X::This ()
{
    cout << "this : " << this << endl
        << "*this: " << *this << endl
        << endl ;

    cout << "this->charray      : " << this->charray      << endl
        << "(*this).charray   : " << (*this).charray   << endl
        << "this->charray[0] : " << this->charray[0] << endl
        << "(*this).charray[0]: " << (*this).charray[0] << endl
        << endl ;

    cout << "&this->charray     : " << &this->charray     << endl
        << "&(*this).charray   : " << &(*this).charray   << endl
        << "&this->charray[0] : " << &this->charray[0] << endl
        << "&(*this).charray[0]: " << &(*this).charray[0]<< endl
        << endl ;
}

void main ()
{
    X ().This () ;
}

```

with output:

```

this : 0x2b0f2256
*this: string

this->charray      : string
(*this).charray   : string
this->charray[0]  : s
(*this).charray[0]: s

&this->charray     : 0x2b0f2256
&(*this).charray   : 0x2b0f2256

```

---

```
&this->charray[0] : string
&(*this).charray[0]: string
```

### 12.29.2 Restrictions on this

The **this** pointer cannot be used in either **static** member functions or **friend** functions. **static** member functions do not have an implicit **this** pointer because they are not related to a particular object. **friend** functions do not have a **this** pointer because **friend** functions are not members of a **class**.

## 12.30 A Matrix class

Let's conclude this chapter by developing a Matrix **class**. A first attempt is given in the program MTRX.CPP:

```
// mtrx.cpp
// a Matrix class
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()

class Matrix
{
private:
    class Attributes // nested class
    {
    public:
        double** m; // pointer to matrix
        int      rows, cols;
    };
    Attributes* pm;
public:
    // constructor
    Matrix (int r=1, int c=1, double value=0.0) ;
    // destructor
    ~Matrix () ;
    // member functions
    int Rows () const { return pm->rows ; }
    int Cols () const { return pm->cols ; }
    // overloaded operators
    Matrix& operator = (const Matrix& m) ;
    double& operator ()(int r, int c) ;
    const double& operator ()(int r, int c) const ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Matrix& m) ;
}; // class Matrix
```

```

// 3 arg. constructor
Matrix::Matrix (int r, int c, double value)
{
    pm = new Attributes ;
    assert (pm != NULL) ;
    pm->m = new double*[r] ; // array of pointers to double
    assert (pm->m != NULL) ;

    for (int i=0; i<r; i++)
    {
        // array of double's for each row
        pm->m[i] = new double[c] ;
        assert (pm->m[i] != NULL) ;
    }

    pm->rows = r ;
    pm->cols = c ;

    // initialise matrix
    for (int k=0; k<r; k++)
        for (int l=0; l<c; l++)
            pm->m[k][l] = value ;
}

// destructor
Matrix::~Matrix ()
{
    for (int i=0; i<pm->rows; i++)
        delete pm->m[i] ;
    delete pm->m ;
    delete pm ;
}

// overloaded operators:

// assignment operator =
Matrix& Matrix::operator = (const Matrix& m)
{
    // verify that two matrices are equivalent
    assert(m.pm->rows == pm->rows && m.pm->cols == pm->cols) ;

    for(int i=0; i<pm->rows; i++)
        for(int j = 0; j<pm->cols; j++)
            pm->m[i][j] = m.pm->m[i][j] ;
    return *this ;
}

// subscript operator () (non-const object)
double& Matrix::operator () (int r, int c)
{
    // verify that index values are in range
}

```

```

assert (r >= 0 && r < pm->rows && c >= 0 && c < pm->cols) ;
return pm->m[r][c] ;
}

// subscript operator () (const object)
const double& Matrix::operator () (int r, int c) const
{
// verify that index values are in range
assert (r >= 0 && r < pm->rows && c >= 0 && c < pm->cols) ;
return pm->m[r][c] ;
}

// friend:

// insertion operator <<
ostream& operator << (ostream& s, const Matrix& m)
{
for (int i=0; i<m.pm->rows; i++)
{
for (int j=0; j<m.pm->cols; j++)
    s << setprecision (2)
        << setiosflags (ios::showpoint | ios::fixed)
        << setw (10)
        << m.pm->m[i][j] ;
    s << endl ;
}
return s ;
}

void main ()
{
Matrix m (3, 3, 4.5), n (3, 3) ;

cout << "Matrix m: " << endl << m ;
cout << "Matrix n: " << endl << n ;

// Matrix assignment
n = m ;

cout << "Matrix n (n=m): " << endl << n ;

// Matrix subscript operator
n(2, 2) = 222.22 ;

cout << "Matrix n (n(2,2)): " << endl << n ;
cout << "n.rows: " << n.Rows ()
    << ", n.cols: " << n.Cols () << endl ;
}

```

with output:

```

Matrix m:
    4.50      4.50      4.50
    4.50      4.50      4.50
    4.50      4.50      4.50
Matrix n:
    0.00      0.00      0.00
    0.00      0.00      0.00
    0.00      0.00      0.00
Matrix n (n=m):
    4.50      4.50      4.50
    4.50      4.50      4.50
    4.50      4.50      4.50
Matrix n (n(2,2)):
    4.50      4.50      4.50
    4.50      4.50      4.50
    4.50      4.50      222.22
n.rows: 3, n.cols: 3

```

A nested **class**, **Attributes**, is declared within **class Matrix**:

```

class Attributes // nested class
{
    public:
        double** m; // pointer to matrix
        int      rows, cols;
};

```

**class Attributes** encapsulates a pointer to a matrix as a pointer to a **double** pointer, **m**, and the number of rows and columns of a two-dimensional matrix. Note that the data members of **Attributes** are declared **public** so that the pointer, **pm**, has direct access to **Attributes**' data members. Declaring **Attributes**' data members as **public** is feasible in this instance since **Attributes** is declared a **private** nested **class** of **Matrix**. Two **const** access member functions, **Rows()** and **Cols()**, are defined, which return the number of rows and columns of a **Matrix** object.

The three-argument **Matrix** constructor first requests sufficient memory for the pointer **pm**:

```
pm = new Attributes ;
```

Then, sufficient memory is requested for an array of pointers to **double** for each row of the matrix:

```
pm->m = new double*[r] ;
```

For each row of the matrix, sufficient memory is then requested to hold an array of elements of type **double** of size equal to the number of columns of the matrix:

```

for (int i=0; i<r; i++)
{
    pm->m[i] = new double[c] ;
    //...
}

```

Each pointer returned by the **new** operator is tested, via the *assert()* macro, against the NULL pointer to verify that memory allocation is successful. The constructor concludes by setting the number of rows and columns of the matrix and initialising the value of all of the elements of the matrix. A **for**-loop is used for the initialisation of the matrix elements since the C++ library function *memset()* cannot be used to set a block of memory to a **double** value.

The Matrix destructor first uses a **for**-loop to deallocate the memory associated with each row of the matrix, then deallocates the memory that holds the array of pointers to **double** and finally deallocates the memory pointed to by the *pm* pointer.

The definitions of the overloaded = and << operators are fairly straightforward. The () operator is overloaded as a matrix subscript operator for indexing the elements of a Matrix object:

```
n(2, 2) = 222.22 ;
```

**Matrix**::**operator()** is overloaded for both non-constant and constant objects of **class Matrix**:

```
double&      Matrix::operator () (int r, int c) ;
const double& Matrix::operator () const (int r, int c) ;
```

The return type of the two overloaded member functions is a reference to **double** rather than a **double**. Returning a reference causes the () subscript operator to act on an object's data members directly without returning a copy. Returning by reference allows the **Matrix**::**operator()** function to appear on either the left-hand or right-hand side of the assignment operator.

As it stands, the Matrix **class** appears to work well. However, if the **class** is to be developed further a copy constructor is essential, for the reasons outlined above of passing objects to and returning objects from functions. The assigned subscript operator for **class Matrix** is the nasty () operator (e.g. *m*(1, 2)), which is a cross between something out of Fortran and a C++ function call. But most importantly, the Matrix **class** has been developed quite independently of the previously developed Vector **class**. Since a matrix is a system of equal-sized vectors, it makes sense to build our Matrix **class** using the Vector **class**. Another very good reason for using Vector in our design of Matrix is that it solves the problem of subscripting the elements of a Matrix object in a C++ array style, e.g. *m*[1][2].

To increase the modularity of the Vector and Matrix classes the **class** declarations are placed in .H header files and the member function definitions are placed in .CPP implementation files. First, the Vector **class**:

```
// vector.h
// header file for Vector class

#ifndef _VECTOR_H // prevent multiple includes
#define _VECTOR_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...
#include <assert.h> // assert()

class Vector
```

```

{
private:
    // data members
    double* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector ()
        : array (NULL), n_elements (0) {}
    Vector (int n) { Allocate (n) ; }
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ;
                  array = NULL ; n_elements = 0 ; }
    // member functions
    int NumberElements () const
        .... { return n_elements ; }
    double& Value (int index) ;
    const double& Value (int index) const ;
    void New (int new_n) ;
    // overloaded operators
    Vector& operator = (const Vector& v) ;
    double& operator [] (int index) ;
    const double& operator [] (int index) const ;
    Vector operator + (const Vector& v) ;
    Vector operator - (const Vector& v) ;
    // friend
    friend ostream& operator << (ostream& s,
                                         const Vector& v) ;
}; // class Vector

#endif // _VECTOR_H

```

with corresponding implementation file VECTOR.CPP. And now, the Matrix **class**:

```

// matrix.h
// header file for Matrix class

#include "vector.h"

#ifndef _MATRIX_H // prevent multiple includes
#define _MATRIX_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()
#include <math.h> // fabs()

```

```
enum Boolean { FALSE, TRUE } ;

class Matrix
{
private:
    Vector** array ;
    int      rows, cols ;
public:
    // constructor
    Matrix (int r=1, int c=1, double value=0.0) ;
    // copy constructor
    Matrix (const Matrix& m) ;
    // destructor
    ~Matrix () ;
    // member functions
    double&      Value (int row, int col) ;
    const double& Value (int row, int col) const ;
    int           Rows () const { return rows ; }
    int           Cols () const { return cols ; }
    Vector        GaussElimination (Vector& b,
                                    Boolean& solved) ;
    // overloaded operators
    Matrix&       operator = (const Matrix& m) ;
    Vector&       operator [] (int index) ;
    const Vector& operator [] (int index) const ;
    Matrix         operator + (const Matrix& m) ;
    Matrix         operator - (const Matrix& m) ;
    Matrix         operator * (const Matrix& m) ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Matrix& m) ;
}; // class Matrix

#endif // _MATRIX_H

// matrix.cpp
// implementation file for Matrix class

#include "matrix.h"

const double TOLERANCE = 1e-06 ;

// swap two doubles
void Swap (double& a, double& b)
{
    double temp = a ;
    a = b ;
    b = temp ;
}

// 3 arg. constructor
```

```
Matrix::Matrix (int r, int c, double value)
{
    // array of pointers to Vector
    array = (Vector**)new Vector[r] ;
    assert (array != NULL) ;

    for (int i=0; i<r; i++)
    {
        array[i] = new Vector (c) ; // array of Vectors
        assert (array[i] != NULL) ;
    }

    rows = r ; // set row & column dimensions
    cols = c ;

    // initialise matrix
    for (int k=0; k<r; k++)
        for (int l=0; l<c; l++)
            Value (k, l) = value ;
}

// copy constructor
Matrix::Matrix (const Matrix& m)
{
    // array of pointers to Vector
    array = (Vector**)new Vector[m.Rows()] ;
    assert (array != NULL) ;

    for (int i=0; i<m.Rows(); i++)
    {
        array[i] = new Vector (m.Cols()) ; // array of Vectors
        assert (array[i] != NULL) ;
    }

    rows = m.Rows () ; // set row & column dimensions
    cols = m.Cols () ;

    // initialise matrix
    for (int k=0; k<m.Rows(); k++)
        for (int l=0; l<m.Cols(); l++)
            Value (k, l) = m.Value (k, l) ;
}

// destructor
Matrix::~Matrix ()
{
    for (int i=0; i<rows; i++)
        delete array[i] ;
    delete [] array ;

    rows = cols = 0 ;
}
```

```
}

// public member functions:

// returns element value (non-const object)
double& Matrix::Value (int row, int col)
{
    assert (row >= 0 && row < rows && col >=0 && col < cols) ;
    return array[row]->Value(col) ;
}

// returns element value (const object)
const double& Matrix::Value (int row, int col) const
{
    assert (row >= 0 && row < rows && col >=0 && col < cols) ;
    return array[row]->Value(col) ;
}

// Gaussian elimination
Vector Matrix::GaussElimination (Vector& b, Boolean& solved)
{
    solved = FALSE ;

    int n = rows ; // assign number of rows to n

    Vector x (n) ; // solution Vector

    // Gaussian elimination:

    for (int i=0; i<n; i++)
    {
        double max_pivot = fabs (Value (i, i)) ;
        int pivot_row = i ;

        // find maximum pivot & row in which it occurs
        for (int k=i+1; k<n; k++)
        {
            if (fabs(Value(k, i)) > max_pivot)
            {
                max_pivot = fabs (Value (k, i)) ;
                pivot_row = k ;
            }
        }

        if (max_pivot > TOLERANCE) // if pivot o.k.
        {
            solved = TRUE ;

            // if pivot found, swap rows
            if (i != pivot_row)
            {
```

```

for (int j=0; j<n; j++)
    Swap (Value (i, j),
          Value (pivot_row, j)) ;
    Swap (b[i], b[pivot_row]) ;
}
for (int j=i+1; j<n; j++)
{
    double multiplier = Value (j, i) / Value (i, i) ;
    for (int k=i; k<n; k++)
        Value (j, k) -= multiplier * Value (i, k) ;
    b[j] -= multiplier * b[i] ;
}
}
else // if pivot not o.k.
{
    solved = FALSE ;
    return x ; // return zero solution Vector
}
}

// back substitution:

// (n-1) element
x[n-1] = b[n-1] / Value (n-1, n-1) ;

// for i=n-2,n-3,...0
for (int s=n-2; s>=0; s--)
{
    double sum = 0.0 ;
    for (int t=s+1; t<=n-1; t++)
        sum += Value (s, t) * x[t] ;
    x[s] = (b[s] - sum) / Value (s, s);
}
return x ;
} // GaussElimination()

// overloaded operators:

// assignment operator =
Matrix& Matrix::operator = (const Matrix& m)
{
    // verify that the two matrices are of equivalent
    // dimensions
    assert(rows == m.Rows() && cols == m.Cols()) ;

    for (int i=0; i<rows; i++)
        for (int j = 0; j<cols; j++)
            Value (i, j) = m.Value (i, j) ;
    return *this ;
}

```

```
// subscript operator [] (non-const object)
Vector& Matrix::operator [] (int index)
{
    // verify that index value is in range
    assert (index >= 0 && index < rows) ;
    return *array[index] ;
}

// subscript operator [] (const object)
const Vector& Matrix::operator [] (int index) const
{
    // verify that index value is in range
    assert (index >= 0 && index < rows) ;
    return *array[index] ;
}

// addition operator +
Matrix Matrix::operator + (const Matrix& m)
{
    assert (rows == m.Rows() && cols == m.Cols()) ;

    Matrix add (rows, cols) ;

    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            add.Value (i, j) = Value (i, j) + m.Value (i, j) ;
    return add ;
}

// subtraction operator -
Matrix Matrix::operator - (const Matrix& m)
{
    assert (rows == m.Rows() && cols == m.Cols()) ;

    Matrix sub (rows, cols) ;

    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            sub.Value (i, j) = Value (i, j) - m.Value (i, j) ;
    return sub ;
}

// multiplication operator *
Matrix Matrix ::operator * (const Matrix& m)
{
    assert (cols == m.Rows()) ;

    Matrix result (rows, m.Cols()) ;

    for (int i=0; i<rows; i++)
    {
```

```

    for (int j=0; j<m.Cols(); j++)
    {
        double sum = 0.0 ;
        for (int k=0; k<cols; k++)
            sum += Value (i, k) * m.Value(k, j) ;
        result.Value(i, j) = sum ;
    }
}
return result ;
}

// friend:

// insertion operator <<
ostream& operator << (ostream& s, const Matrix& m)
{
    for (int i=0; i<m.rows; i++)
    {
        for (int j=0; j<m.cols; j++)
            s << setprecision (2)
                << setiosflags (ios::showpoint | ios::fixed)
                << setw (10)
                << m.Value (i, j) ;
        s << endl ;
    }
}
return s ;
}

```

Program MTRX\_TST.CPP tests the Vector and Matrix classes:

```

// mtrx_tst.cpp
// tests the modified Matrix class which constructs an array
// of Vector objects and allows C++-style subscripting of
// Matrix elements
#include <iostream.h> // C++ I/O

#include "vector.h" // class Vector
#include "matrix.h" // class Matrix

void main ()
{
    Vector v (3) ;
    Matrix m (3, 3, 2.5), n (3, 3) ;

    cout << "Vector v: " << endl << v << endl ;
    cout << "Matrix m: " << endl << m ;

    // Matrix assignment
    n = m ;

    // Matrix subscript operator

```

```

n[1][2] = 222.22 ;

cout << "Matrix n: " << endl << n ;

// Matrix multiplication
Matrix m1 (2, 3) ;
Matrix m2 (3, 3) ;
m1[0][0] = 3 ; m1[0][1] = 2 ; m1[0][2] = -1 ;
m1[1][0] = 0 ; m1[1][1] = 4 ; m1[1][2] = 6 ;

m2[0][0] = 1 ; m2[0][1] = 0 ; m2[0][2] = 2 ;
m2[1][0] = 5 ; m2[1][1] = 3 ; m2[1][2] = 1 ;
m2[2][0] = 6 ; m2[2][1] = 4 ; m2[2][2] = 2 ;

Matrix m3 (2, 3) ;
m3 = m1 * m2 ;
cout << "Matrix m3: " << endl << m3 ;

// Gaussian elimination
Matrix a (3, 3) ;
Vector b (3), x (3) ;

a[0][0] = 1.0 ; a[0][1] = 2.0 ; a[0][2] = 3.0 ;
a[1][0] = 2.0 ; a[1][1] = 3.0 ; a[1][2] = 4.0 ;
a[2][0] = 3.0 ; a[2][1] = 4.0 ; a[2][2] = 1.0 ;
b[0] = 14.0 ; b[1] = 20.0 ; b[2] = 14.0 ;

cout << "Matrix a: " << endl << a ;
cout << "Vector b: " << endl << b << endl ;

Boolean bool = FALSE ;

x = a.GaussElimination (b, bool) ;

if (bool)
    cout << "solution Vector x of system [a]{x}={b}: "
          << endl << x << endl ;
else
    cout << "solution by Gauss elimination failed" << endl ;
}

```

and generates the following output:

```

Vector v:
[0.00, 0.00, 0.00]
Matrix m:
 2.50      2.50      2.50
 2.50      2.50      2.50
 2.50      2.50      2.50
Matrix n:
 2.50      2.50      2.50

```

```

    2.50      2.50      222.22
    2.50      2.50      2.50
Matrix m3:
    7.00      2.00      6.00
    56.00     36.00     16.00
Matrix a:
    1.00      2.00      3.00
    2.00      3.00      4.00
    3.00      4.00      1.00
Vector b:
[14.00, 20.00, 14.00]
solution Vector x of system [a]{x}={b}:
[1.00, 2.00, 3.00]

```

The above declaration of **class** Vector is similar to that of program VECT1.CPP.

The **private** data members of **class** Matrix are declared as:

```

class Matrix
{
private:
    Vector** array ;
    int      rows, cols ;
//...
};

```

The **private** data member array is a pointer to a Vector pointer. Within the body of the three-argument constructor of Matrix sufficient memory is requested for a matrix by first defining an array of pointers to Vector for each row of the matrix:

```
array = (Vector**) new Vector[r] ;
```

and then requesting sufficient memory for each Vector row of the matrix:

```

for (int i=0; i<r; i++)
{
    array[i] = new Vector (c) ;
//...
}

```

The Matrix destructor first deletes the memory allocated for each Vector row and then the memory allocated to hold the array of pointers to Vector, finally setting the number of rows and columns to zero:

```

Matrix::~Matrix ()
{
    for (int i=0; i<rows; i++)
        delete array[i] ;
    delete [] array ;

    rows = cols = 0 ;
}

```

**class Matrix** defines a copy constructor and two overloaded member functions, *Value()*, that access the value of a particular element for both non-constant and constant **Matrix** objects. The insertion operator (*<<*) is overloaded to enable C++-style output of **Matrix** objects.

The arithmetic binary operators *+*, *-* and *\** are overloaded for **Matrix** objects. If two (*m*×*n*) matrices **A**=[*a<sub>jk</sub>*] and **B**=[*b<sub>jk</sub>*] are added, the sum matrix **C**[*c<sub>jk</sub>*]=**A**+**B** is size (*m*×*n*) with elements:

$$c_{jk} = a_{jk} + b_{jk}, \quad j = 1, 2, \dots, m; \quad k = 1, 2, \dots, n$$

and similarly for the subtraction of two matrices.

Suppose now that **A**=[*a<sub>jk</sub>*] is an (*m*×*n*) matrix and **B**=[*b<sub>jk</sub>*] is an (*r*×*p*) matrix; then the product **C**[*c<sub>jk</sub>*]=**AB** is size (*m*×*p*) with elements:

$$c_{jk} = a_{j1}b_{1k} + a_{j2}b_{2k} + \dots + a_{jn}b_{nk} = \sum_{i=1}^n a_{ji}b_{ik}$$

and is defined only when *r*=*n* (i.e. the number of rows in the second matrix is equal to the number of columns of the first matrix). Note that matrices are associative and distributive with respect to addition and subtraction, but are non-commutative with respect to multiplication; that is **AB**≠**BA**. An example of the multiplication of two **Matrix** objects, **m1** and **m2**, is given in the *main()* function of MTRX\_TST.CPP.

Let's now examine the indexing of elements of a **Matrix** object.

### 12.30.1 A Subscript Operator for **class Matrix**

It was noted in Chapter 10 and the first implementation of **class Matrix** in program MTRX.CPP that it is not a straightforward operation to overload the subscript operator for a two-dimensional **Matrix** **class**. Consider the use of the subscript operator with an object of **class Matrix**:

```
Matrix m (10, 10) ;
//...
m[row][col] = x ;
```

To understand exactly how indexing the elements of a **Matrix** object occurs we must remember that the subscript operator, **[ ]**, is a binary left-associative operator. Therefore, the above expression is equivalent to:

```
(m[row])[col] = x ;
```

where **m[row]** indicates the **Vector** **row** of the matrix and **[col]** indicates the element at position **col** of the respective row. Thus, to index an element in a two-dimensional matrix will require the overloading of two individual **[ ]** operators for both **Matrix** and **Vector** objects. The left-hand overloaded **[ ]** operator implicitly acts on a **Matrix** object and is passed the row number of the matrix:

```
Vector& Matrix::operator [] (int index) // non-const objects
{
```

---

```
//...
return *array[index] ;
}
```

whereas the right-hand overloaded [] operator implicitly acts on a `Vector` object and is passed the element number of a given row:

```
double& Vector::operator [] (int index) // non-const objects
{
//...
return array[index] ;
}
```

Notice that subscripting the elements of a `Matrix` object in a C++ style is made possible by constructing a `Matrix` object as an array of `Vector` objects.

### 12.30.2 Gaussian Elimination of a System of Linear Equations

This section examines the development of a member function `GaussElimination()` that solves a system of  $n$  linear equations in  $n$  unknowns  $x_1, x_2, \dots, x_n$ :

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

where the coefficient matrix  $A=[a_{ij}]$  is an  $(n \times n)$  matrix and  $\mathbf{x}=[x_i]$  and  $\mathbf{b}=[b_i]$  are column vectors.

Let's begin by considering a diagonal matrix (i.e.  $a_{ij}=0$  for  $i \neq j$ ;  $a_{ii}=0$  for  $i=1, 2, \dots, n$ ). In this case, the solution vector is known immediately:

$$x_i = \frac{b_i}{a_{ii}}, \quad i = 1, 2, \dots, n$$

If matrix  $A$  is upper-triangular with no diagonal element,  $a_{ii}$ , equal to zero, then  $\mathbf{x}$  is easily found:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

The unknown  $x_n$  is found immediately:

$$x_n = \frac{b_n}{a_{nn}}$$

Given  $x_n, x_{n-1}$  is found from the  $(n-1)$ th equation:

$$x_{n-1} = \frac{b_{n-1} - a_{n-1}x_n}{a_{n-1, n-1}}$$

and in general:

$$x_{n-1} = \frac{b_i - a_{i,i+1}x_{i+1} - \dots - a_{in}x_n}{a_{ii}}, \quad i = n-1, n-2, \dots, 1$$

This process of determining  $x_i$  is referred to as *back-substitution*, an algorithm of which is outlined below:

*algorithm: back-substitution*

given: upper-triangular form of  $A, b, n$

$$x_n = b_n/a_{nn}$$

for  $i=n-1, n-2, \dots, 1$

$$x_i = \left( b_i - \sum_{j=i+1}^n a_{ij}x_j \right) / a_{ii}$$

For example, if we apply the back-substitution algorithm to the system:

$$\begin{bmatrix} 3 & 4 & 1 \\ & 1 & 2 \\ & & 2 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 14 \\ 8 \\ 6 \end{Bmatrix}$$

we find:

$$x_3 = b_3 / a_{33} = 6 / 2 = 3$$

$$x_2 = (b_2 - a_{23}x_3) / a_{22} = (8 - 2 \cdot 3) / 1 = 2$$

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3) / a_{11} = (14 - 4 \cdot 2 - 1 \cdot 3) / 3 = 1$$

Once a set of linear equations has been reduced to an upper-triangular matrix, the solution vector  $\mathbf{x}$  is easily found by performing back-substitution. The key to solving a set of linear equations lies in the process of reducing a set of simultaneous equations to triangular form. One of the more well-known methods is Gaussian elimination, which is best illustrated by using an example. Consider the system of equations for which a solution is required:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 14 \\ 20 \\ 14 \end{Bmatrix}$$

The first part of the elimination process eliminates  $x_1$  from the second and third equations. This is achieved by subtracting multiples of the first equation from the second and third

equations. To eliminate  $x_1$  from the second and third equations *multipliers* of  $2/1=2$  and  $3/1=3$ , respectively, are required:

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & \\ -2 & -8 & \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 14 \\ -8 \\ -28 \end{Bmatrix}$$

During the elimination of  $x_1$  from equations two and three, the first equation is called the *pivotal equation* and its diagonal coefficient the *pivot*. Eliminating  $x_2$  from the third equation now requires that the second equation becomes the pivotal equation, with element  $a_{12}$  the pivot and a multiplier of  $3/1=2$ :

$$\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & \\ 4 & & \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 14 \\ -8 \\ 12 \end{Bmatrix}$$

Back-substitution can now be performed to determine vector  $\mathbf{x}$ :

$$\mathbf{x} = \begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix}$$

An algorithm for Gaussian elimination is:

*algorithm: Gaussian elimination*

```
given: A, b, n
for i = 1, 2, ..., n-1
    for k = i+1, ..., n
        multiplierki = aki/aii
        for j=i+1, ..., n
            akj = akj - multiplierki * aij
            bk = bk - multiplierki * bi
```

What if a small or zero pivot element is encountered? One solution would be simply to exit the routine, but a more popular approach is to perform *partial pivoting*. To illustrate the technique of partial pivoting, consider once more the system of equations:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} 14 \\ 20 \\ 14 \end{Bmatrix}$$

At present, the element coefficients of  $x_1$  are 1, 2 and 3 for equations 1, 2 and 3, respectively, with multipliers 2 and 3. To make the value of the pivot as large as possible and hence reduce the magnitude of the multipliers, interchange rows 1 and 3:

$$\begin{bmatrix} 3 & 4 & 1 \\ 2 & 3 & 4 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ 20 \\ 14 \end{bmatrix}$$

This system now has multipliers of 2/3 and 1/3 for rows 2 and 3, respectively. This process is repeated for each part of the triangularisation.

The procedure described above is implemented in the member function *Matrix::GaussElimination()*, remembering that the indexing of an array of size *n* in C++ is in the range (0:*n*-1) rather than (1:*n*). A typical application of *GaussElimination()* is demonstrated in MTRX\_TST.CPP:

```
Matrix a (3, 3) ;
Vector b (3), x (3) ;
Boolean bool ;
//...
x = a.GaussElimination (b, bool) ;
```

It is worth highlighting that swapping all elements of two rows for a large system of equations can be computationally very expensive. Thus, in practice an integer array of ‘pointers’ is defined which records the order of the rows without actually performing element swapping. To define such an array in *GaussElimination()* would require the use of a dynamic one-dimensional array or vector of integers – which at present we have not discussed.

Note that *complete* pivoting is possible where the element of largest magnitude in the submatrix ( $A'=[a_{jk}]$ ;  $k, j \geq i$ ) is located and moved to the pivot position by row and column interchanges if necessary. In practice, such a process is generally considered to be computationally too expensive, and partial pivoting is the generally adopted technique.

A measure of the efficiency (or inefficiency) of the Gaussian elimination algorithm is the number of arithmetic operations required to obtain a solution vector. Excluding back-substitution there are  $n(n^2-1)/3$  additions,  $n(n^2-1)/3$  multiplications and  $n(n-1)/2$  divisions. The back-substitution part of the Gaussian elimination algorithm consists of  $n(n-1)/2$  additions,  $n(n-1)$  multiplications and  $n$  divisions.

For a more detailed examination of matrix computations the interested reader is referred to Jennings and McKeown (1992) and Golub and Van Loan (1989).

### 12.30.3 Return Value of *GaussElimination()*

The return value of *GaussElimination()* is an object of **class** Vector. This is feasible for small Vector objects, but very inefficient for larger objects because of the time and memory required for the compiler to generate a temporary object to hold the return object of *GaussElimination()*. A better approach is to return a pointer to the solution Vector object returned by *GaussElimination()*. Adopting this approach requires a few small alterations to the above MATRIX.H/.CPP and MTX\_TST.CPP programs, which are summarised below:

```
// matrix.h
//...
class Matrix
{
public:
    Vector* GaussElimination (Vector& b, Boolean& solved) ;
```

```

//...
//...
};

// matrix.cpp
//...
Vector* Matrix::GaussElimination (Vector& b, Boolean& solved)
{
//...
Vector* x = new Vector (n) ; // solution Vector
//...
// back substitution:
//...
x->Value(n-1) = b[n-1] / Value (n-1, n-1) ;
//...
for (int t=s+1; t<=n-1; t++)
    sum += Value (s, t) * x->Value(t) ;
x->Value(s) = (b[s] - sum) / Value (s, s);
}
return x ;
} // GaussElimination()

// mtrx_tst.cpp
//...
void main ()
{
Vector* x = new Vector (3) ;
//...
x = a.GaussElimination (b, bool) ;

if (bool)
    cout << "solution Vector x of system [a]{x}={b}: "
        << endl << *x << endl ;
//...
delete x ;
}

```

## 12.31 Summary

Hopefully the size of the present chapter has helped to emphasise the level of importance attached to pointers when programming in C++. If there are any topics in this chapter that you feel uncomfortable with, then I seriously recommend referring back to them now before proceeding further.

A pointer is simply a variable that holds the memory address of another variable or object. Pointers are defined using an asterisk (\*) after the data type or **class** that is to be pointed to. The lvalue of a pointer is in fact the lvalue or memory address of another variable or object. The lvalue of the pointer is the memory address of the pointer itself. The memory address of an object can be assigned to a pointer by means of the address-of operator (&), and the object itself can be accessed indirectly by means of the indirection operator (\*) preceding the pointer.

The data members of an object or member function of the object's **class** are accessed by the direct member access operator (.), whereas the members of an object can be accessed indirectly via a pointer to the object by the indirect member access operator (->).

As with objects, pointers can be **const** modified so that a programmer has complete flexibility of defining either a pointer to a constant object or a constant pointer to an object or both. C++ supports a pointer to **void** which allows any type of pointer to be assigned to itself. Pointers to **void** can greatly increase the generality of a function.

Pointers enable the memory address of an object to be passed to or returned by a function. Passing or returning an object by pointer eliminates any copying of the object, since the actual memory address of the object is being referenced and not the object itself.

Pointers are variables and as such can be used in arithmetical expressions. Pointer arithmetic is, however, restricted to the addition, subtraction, increment and decrement operators.

There is a close relationship between arrays and pointers in C++, since the name of an array is a synonym for the memory address of the array's zeroth element. This property of arrays enables the memory address of an array to be assigned to a pointer and then the elements of the array accessed by pointer arithmetic rather than by the array subscript operator. C++ also adopts the convention that a function name is a synonym for its memory address, which enables the memory address of a function to be assigned to a pointer.

C++ offers a programmer two operators for performing dynamic memory allocation, namely **new** and **delete**. The **new** operator allocates a block of memory from the operating system and returns a pointer to that block. The **delete** operator deallocates a block of memory allocated by the **new** operator. The **new** and **delete** operators can be used either to allocate or deallocate memory for objects or arrays of objects. Since **new** and **delete** are operators they can be overloaded specifically for a given user-defined **class**.

When an object of a **class** calls a member function of that **class**, a pointer to that object is implicitly returned to the member function and is referred to as the **this** pointer.

## Exercises

12.1 By dereferencing the pointer to **int** **p\_int**, assign the value 11 to **i**:

```
void main ()
{
    int i (0) ;

    int* p_int (NULL) ;
    //...
}
```

12.2 The trapezoidal rule of numerical integration approximates a continuously differentiable function  $f(x)$  within the interval  $a \leq x \leq b$  by a piecewise linear function which is the polygon of  $n$  chords of the curve  $f$ :

$$I = \int_a^b f(x) dx = h \left( \frac{1}{2} f(a) + f(x_1) + \dots + f(x_{n-1}) + \frac{1}{2} f(b) \right)$$

where  $h = (b-a)/n$  and  $x_j (= a+jh; j=1:n-1)$  are the equal interval points. Develop a function which performs trapezoidal integration and is passed an arbitrary function with the following signature:

---

```
double Function (double x) ;
```

- 12.3 Rather than implementing a Matrix **class** as an array of pointers to Vector objects, as shown in the present chapter, declare a Matrix **class** which encapsulates a single Vector object and consequently stores a two-dimensional matrix as a one-dimensional vector.
- 12.4 Develop three triangle classes which encapsulate object, reference and pointer Point data members.
- 12.5 The C++ library function *bsearch()* has the following signature:

```
void* bsearch (const void* key, const void* base,
                size_t n_elem, size_t width,
                int (*fcmp)(const void*,  

                const void*)) ; // STDLIB.H
```

*bsearch()* performs a binary search of an array (pointed to by *base*) of *n\_elem* elements (each of size *width* in bytes) for element. The comparison function, *fcmp*, compares the elements of an array and returns an integer value of <0, ==0 or >0 if the first element is less than, equal to or greater than the second element respectively. If a match for *element* is found in the array then *bsearch()* returns the first array index for which a match is found, else *bsearch()* returns NULL. The **typedef** *size\_t* (**unsigned**) is defined in STDDEF.H. For *bsearch()* to work the array elements must be in order.

Write a program which searches, using *bsearch()*, a sorted array of integers for a given integer.

- 12.6 For the String **class** of ND\_STR.CPP, overload the **new** and **delete** operators.

- 12.7 Develop a simple linked list **class**, *LinkedList*, to hold a list of integers. A linked list consists of an arbitrary number of nodes which encapsulate both a data value and a pointer to the next node in the list. Implement a node as a Node **class** nested within *LinkedList*. Provide a *LinkedList*::*Add()* member function which allows integers to be added to a *LinkedList* object.

# Templates

*Your quote here.*  
– B. Stroustrup

*Templates allow a programmer to create generic functions and parametrised types or generics. The type of data or objects that a template function or class operates on remains unspecified in the function definition and class declaration until a function is called or an object is defined. Such type-independent functions and classes frequently arise when the operation of a function or data of an object is independent of type. The real power of templates is that a function can be defined or a class declared just once, but can be operated on by all C++ integral or user-defined types.*

*To see templates in action, a linked list class, `LinkedList`, is implemented along with an associated iterator class, `LinkedListIterator`. Iterators are functions or classes whose sole purpose is to iterate through the encapsulated data elements of an object. Iterators provide a safe way of ensuring that out-of-bounds object element indexing does not occur. Vector and Matrix classes are also developed and help to illustrate composed types and the common pitfall when designing template classes of inserting type dependencies.*

*A `Polygon` class is developed which makes good use of `LinkedList` by encapsulating a linked list of `Point` vertices. Although a relatively simple implementation, class `Polygon` allows an object to be defined with an arbitrary number of vertices for both convex and concave polygons, and supports operations for determining whether a point is inside a polygon and returning the convex hull of a polygon. class `Polygon` illustrates that by encapsulating a polygon's vertices as a `LinkedList` object, rather than a pointer to `Point`, all the memory management issues behind defining a `Polygon` object are performed by `LinkedList`. The use of powerful dynamic data structures such as `Vector` and `LinkedList` allows the programmer to focus on the application at hand rather than repeatedly getting involved in the memory management of objects.*



## 13.1 Function Overloading

Let us first examine defining a function which returns the maximum of its two arguments. We could simply define functions `CharMax()`, `IntMax()`, `FloatMax()` etc., which operate on `char`, `int`, `float` data, respectively, or we could alternatively define a set of overloaded functions with the same function name:

```

// fun_over.cpp
// implements a Max() function with
// different argument types by function overloading
#include <iostream.h> // C++ I/O

// returns the maximum of two chars
char Max (char a, char b)
{
    return a > b ? a : b ;
}

// returns the maximum of two ints
int Max (int a, int b)
{
    return a > b ? a : b ;
}

// returns the maximum of two floats
float Max (float a, float b)
{
    return a > b ? a : b ;
}

// returns the maximum of two doubles
double Max (double a, double b)
{
    return a > b ? a : b ;
}

void main ()
{
    char ca = 'a', cb = 'b' ;
    int ia = 1 , ib = 2 ;
    float fa = 3.0, fb = 4.0 ;
    double da = 6.0, db = 5.0 ;

    cout << "Max(char, char)      : " << Max (ca, cb) << endl ;
    cout << "Max(int, int)       : " << Max (ia, ib) << endl ;
    cout << "Max(float, float)   : " << Max (fa, fb) << endl ;
    cout << "Max(double, double): " << Max (da, db) << endl ;
}

```

with output:

```

Max(char, char)      : b
Max(int, int)       : 2
Max(float, float)   : 4
Max(double, double): 6

```

This approach works fine, but we observe from the above definitions of the four overloaded `Max()` functions that each function has exactly the same setup. Each function has two

arguments of the same type, returns a type the same as the type of its arguments and has an identical single **return** statement:

```
return a > b ? a : b ;
```

If we now require that the *Max()* function operates on other types or classes, such as *Complex* or *Point*, we would have to overload *Max()* specifically for these classes.

## 13.2 Template Functions

Alternatively, C++ allows a programmer to define *Max()* as a *template* or *generic* function that is defined independent of a particular type or **class**. The type or **class** of an object that a template function operates on is implicitly passed as an argument. Let's modify the above program, *FUN\_OVER.CPP*, defining *Max()* as a **template** function:

```
// tmp_fun.cpp
// implements a Max() function as a template function
#include <iostream.h> // C++ I/O

// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}

void main ()
{
    // note: identical to main() in fun_over.cpp
    char ca = 'a', cb = 'b' ;
    int ia = 1 , ib = 2 ;
    float fa = 3.0, fb = 4.0 ;
    double da = 6.0, db = 5.0 ;

    cout << "Max(char, char) : " << Max (ca, cb) << endl ;
    cout << "Max(int, int) : " << Max (ia, ib) << endl ;
    cout << "Max(float, float) : " << Max (fa, fb) << endl ;
    cout << "Max(double, double): " << Max (da, db) << endl ;
}
```

Immediately we note the reduced program size of *TMP\_FUN.CPP* compared to *FUN\_OVER.CPP*. Also, it is worth noting that the *main()* functions in both of the above two programs are identical (except for the comment), which illustrates that there is no syntactical difference between a **template** function call and a *normal* function call. In contrast with **template** classes, you do not specify the type of the **template** arguments when calling a **template** function. For example, in the call:

```
int i, j ;
//...
```

---

```
int k = Max (i, j) ;
```

the compiler knows that the call is of the form `Max(int, int)`, since the type of `i` and `j` is `int`.

The `Max()` **template** function is defined as:

```
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}
```

and is of the general single type argument or parameter **template** function definition syntax:

```
template <class type>
return_type FunctionName (parameter_list)
{
    // ...
}
```

**template** is a C++ keyword. `type`<sup>1</sup> is a type argument which indicates the type on which the function operates and is placed between the set of angle brackets (< and >). The scope of `type` extends from the function declarator to the end of the **template** function definition. Although `type` is preceded by the keyword **class** the type argument can in fact be any valid type, and not necessarily a **class** (e.g. a **double**). Templates are automatically instantiated by the compiler replacing `type` with the actual data type specified in the function call. Note that `type T` is used throughout the function declarator of `Max()`, including the return type.

In program TMP\_FUN.CPP the `Max()` function is defined before it is called in `main()`. If the definition of `Max()` appears after the definition of `main()`, we require a **template** declaration of `Max()`:

```
// tmp_dec.cpp
// illustrates a template function declaration
#include <iostream.h> // C++ I/O

// global template function declaration
template <class T> T Max (T a, T b) ;

void main ()
{
    // local-to main() declaration of Abs()
    int Abs (int n) ;
    // local-to-main() declaration of Swap()
    template <class T> void Swap (T& a, T& b) ; // error!

    int i = 1, j = 2;
```

---

<sup>1</sup> The generally adopted letter and name for a single type argument are `T` and `Type`, respectively. Perhaps the letter `C` should be adopted for **class** or `TC` for `type` or **class**, since subtle differences do exist between a `type` and a **class** in C++.

```

int max = Max (i, j) ;
cout << "max: " << max ;

int abs = Abs (-1) ;
cout << "abs: " << abs ;
}

// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
return a > b ? a : b ;
}

// returns the absolute value of an int
int Abs (int n)
{
return n > 0 ? n : -n ;
}

// swaps two objects
template <class T>
void Swap (T& a, T& b)
{
T temp ;

temp = a ;
a = b ;
b = temp ;
}

```

The **template** declaration of *Max()* is a global declaration. Template declarations must be global declarations, whereas non-**template** functions can be declared local to a function. This is illustrated by the **template** declaration of *Swap()* within *main()*, which will generate a compilation error and the non-**template** declaration of *Abs()* within *main()*, which is legal.

### 13.2.1 A Note on Style

The **template** function declarator of *Max()* was written on two lines:

```

template <class T>
T Max (T a, T b)

```

rather than on a single line:

```
template <class T> T Max (T a, T b)
```

As with all style issues, the compiler couldn't care less. However, from a programmer's point of view the first version greatly increases the clarity of the function definition. This issue of style is particularly relevant to defining out-of-line **class** member functions, e.g.:

```
template <class T>
inline Point<T>& Point<T>::operator = (const Point<T>& p)
```

### 13.2.2 Templates or Macros?

You may be thinking that a **template** function is a kind of macro:

```
#define Max(a, b) ((a > b) ? a : b)
```

However, using the `#define` pre-processor directive circumvents the static type-checking mechanism of C++ and allows `Max()` to perform some strange comparisons. Errors generated by macros can be very difficult to detect, whereas C++ compilers are generally very good at detecting **template** errors. Macros are generally difficult to implement and require extra syntax to avoid precedence problems and to keep the pre-processor happy.

### 13.2.3 Template Function Arguments

To ensure that the compiler can generate the correct function code from a **template** function, at least one **template** argument type must influence the type of the function. For instance, all of the following are legal **template** function declarations:

```
template <class T> void Function1 (T) ;
template <class T> void Function2 (T, int) ;
template <class T, class C> int Function3 (C) ;
```

whereas the following are illegal:

```
template <class T> void Function3 (int) ;
template <class T> T Function4 (int) ;
template <class T> void* operator new (size_t size) ;
```

### 13.2.4 Multiple Type Arguments

Template functions can also be defined with multiple type arguments, but non-type arguments in a function **template** are not allowed. Template classes, as we shall see later, can be declared with both multiple type and non-type arguments. To illustrate multiple type arguments, consider the following program:

```
// mult_arg.cpp
// illustrates multiple type arguments for template functions
#include <iostream.h> // C++ I/O

// returns the maximum of two objects
template <class T, class S>
void Display (T a, S b)
{
    cout << a << " " << b << endl ;
}

void main ()
```

---

```
{
Display (0, 10.0) ;
Display ("hello", 0) ;
}
```

*Display()* is defined for two generic types, T and S, separated by the comma operator in the **template** parameter type list.

### 13.2.5 Template Functions do not Perform Implicit Casting

It was noted in Chapter 4 that C++ performs implicit casting from one C++ integral type to another:

```
int    i ;
double d ;
//...
i = d ; // implicit cast of double to int
```

This can cause problems when passing arguments to a function. The following program illustrates calling a function *IntMax(int,int)* with two **double** arguments. However, implicit casting is not performed on **template** function arguments:

```
// dif_type.cpp
// illustrates that template functions
// do not perform implicit casting
#include <iostream.h> // C++ I/O

// returns the maximum of two int's
int IntMax (int a, int b)
{
    return a > b ? a : b ;
}

// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}

void main ()
{
    int    ia = 1 , ib = 2    ;
    double da = 3.3, db = 4.4 ;

    // IntMax(double,double)
    cout << IntMax (da, db) << endl ; // o.k.

    // Max()
    cout << Max (ia, ib) << endl ; // o.k. : Max(int,int)
    cout << Max (ia, da) << endl ; // error: Max(int,double)
```

---

```

cout << Max (ia, int(da)) << endl ; // o.k.: Max(int,int)
}

```

Within `main()` an attempt is made to call the **template** function `Max()` with two different types. This will result in a compilation error of the form ‘no match found for `Max(int, double)`’, since `Max()` is defined with both arguments to be of the same type.

### 13.2.6 Overloading Template Functions

Template functions can be overloaded:

```

// tmp_over.cpp
// illustrates that template functions can be overloaded
#include <iostream.h> // C++ I/O

// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}

// returns the maximum of three objects
template <class T>
T Max (T a, T b, T c)
{
    return c > (a > b ? a : b) ? c : a > b ? a : b ;
}

void main ()
{
    int i = 1, j = 2, k = 3;
    cout << Max (i, j) << endl ;
    cout << Max (i, j, k) << endl ;
}

```

The above program overloads the `Max()` **template** function for two and three arguments.

In addition, **template** functions can be overloaded for specific versions of the **template**. The following program explicitly overloads the `Max()` function for arguments of type **int**:

```

// tmp_o1.cpp
// illustrates overloading a template function
#include <iostream.h> // C++ I/O

// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}

```

```
// returns the maximum of two int's
int Max (int a, int b)
{
    cout << "Max(int,int) called" << endl ;
    return a > b ? a : b ;
}

void main ()
{
    int ia      = 1 , ib = 2      ;
    double da = 3.0, db = 4.0 ;

    double max = Max (da, db) ; // calls Max()
    Max (ia, ib) ;           // calls overloaded Max()
}
```

If a **template** function is explicitly overloaded it is the overloaded function that takes priority in a function call. The ability to overload a **template** function explicitly gives a programmer a greater degree of flexibility in designing **template** functions.

### 13.3 Template Classes

You've probably already guessed that C++ supports **template** classes. The philosophy is similar to that of **template** functions in that a **template class** declaration is a mould for a set of classes. To illustrate **template** classes, consider the Point **template class**:

```
// pt_tmpl.cpp
// illustrates a Point template class
#include <iostream.h> // C++ I/O

// template class Point
template <class T>
class Point
{
private:
    // private data members
    T x, y, z ;
public:
    // constructors
    Point () ;
    Point (T x_arg, T y_arg, T z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (T t)
        : x (t), y(t), z (t) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    T& X () { return x ; } // non-const objects
```

```
T& Y () { return y ; }
T& Z () { return z ; }
const T& X () const { return x ; } // const objects
const T& Y () const { return y ; }
const T& Z () const { return z ; }
// overloaded operators
Point& operator = (const Point& p) ;
Point operator + (const Point& p) ;
Point operator - (const Point& p) ;
Point operator * (const Point& p) ;
// friends
friend ostream& operator << (ostream& s,
                                const Point<T>& p) ;
friend istream& operator >> (istream& s, Point<T>& p) ;
}; // template class Point

// constructor:

// no arg. constructor
template <class T>
inline Point<T>::Point ()
: x (0.0), y (0.0), z (0.0) {}

// overloaded operators:

// assignment operator =
template <class T>
inline Point<T>& Point<T>::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}

// addition operator +
template <class T>
inline Point<T> Point<T>::operator + (const Point& p)
{
    return Point<T> (x+p.x, y+p.y, z+p.z) ;
}

// subtraction operator -
template <class T>
inline Point<T> Point<T>::operator - (const Point& p)
{
    return Point<T> (x-p.x, y-p.y, z-p.z) ;
}

// multiplication operator *
template <class T>
inline Point<T> Point<T>::operator * (const Point& p)
{
```

```
return Point<T> (x*p.x, y*p.y, z*p.z) ;  
}  
  
// overloaded operators/friends:  
  
// overloaded insertion operator <<  
template <class T>  
inline ostream& operator << (ostream& s, const Point<T>& p)  
{  
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;  
}  
  
// overloaded extraction operator >>  
template <class T>  
inline istream& operator >> (istream& s, Point<T>& p)  
{  
    return s >> p.x >> p.y >> p.z ;  
}  
  
void main ()  
{  
    Point<double> p (1, 2, 3), q (p), r ; // double data members  
  
    cout << "p: " << p << endl ;  
    cout << "q: " << q << endl ;  
  
    r = q ;  
  
    cout << "r: " << r << endl ;  
  
    Point<int> s ; // int data members  
    Point<float> t ; // float data members  
  
    cout << "s: " << s << endl ;  
    cout << "t: " << t << endl ;  
  
    cout << "sizeof(Point<int>) : "  
        << sizeof (Point<int>) << endl ;  
    cout << "sizeof(Point<float>) : "  
        << sizeof (Point<float>) << endl ;  
    cout << "sizeof(Point<double>): "  
        << sizeof (Point<double>) << endl ;  
    cout << "sizeof s: " << sizeof s << endl ;  
    cout << "sizeof t: " << sizeof t << endl ;  
    cout << "sizeof r: " << sizeof r << endl ;  
  
    int i (0) ;  
    float f (0.0) ;  
  
    f = i ; // o.k.: implicit conversion
```

```
Point<int>    u ;
Point<int>    v ;
Point<float>   w ;

u = v ; // o.k.: u & v are of same type
//w = v ; // error: no implicit conversion
}
```

with output:

```
p: (1, 2, 3)
q: (1, 2, 3)
r: (1, 2, 3)
s: (0, 0, 0)
t: (0, 0, 0)
sizeof(Point<int>) : 6
sizeof(Point<float>) : 12
sizeof(Point<double>): 24
sizeof s: 6
sizeof t: 12
sizeof r: 24
```

The declaration of **template class** Point takes the form:

```
template <class T>
class Point
{
private:
    // private data members
    T x, y, z ;
//...
};
```

and is of the general single type argument syntax:

```
template <class type>
class ClassName
{
//...
};
```

The prefix **template<class T>** indicates to the compiler that a **template class** is being declared. As with **template** functions, **type** is a type argument that indicates the type or **class** on which the **class** operates, and is to be specified when the **template class** is instantiated. **type** is placed between the set of angle brackets (< and >). Next, the keyword **class** precedes the name of the **template class**, **ClassName**.

Following a valid **template class** declaration, an instance or object of the **class** is similar to non-**template** classes except for the inclusion of the type or **class** that the **template class** operates on:

```
ClassName<type> object ;
```

where type is the type or **class** name of the object, object, that the **template class** operates on. For example:

```
Point<int> p ; // int data members
Point<double> q ; // double data members
```

Point<double> operates just as if it had been declared as the non-**template class**:

```
class Point
{
private:
    // private data members
    double x, y, z ;
//...
};
```

The Point **template class** of program PT\_TMP.CPP defines both inline and out-of-line constructors and member functions. Let's examine the inline constructors and member functions first:

```
template <class T>
class Point
{
private:
    //...
public:
    Point (T x_arg, T y_arg, T z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (T t)
        : x (t), y(t), z (t) {}
//...
// public member functions
T& X () { return x ; }
//...
};
```

The layout of the Point **template class** is similar to the non-**template** Point **class** declarations presented in Chapter 9, except that the **double** type has been replaced by the argument T. Do ensure that argument T is correctly used. One of the best techniques for developing a **template class** is to copy an existing (working) non-**template class**, since it can be difficult to develop a **template class** from scratch. The above Point **template class** was originally copied from the program THIS\_PT.CPP of Chapter 12.

The copy constructor of **template class** Point is defined inline as:

```
// copy constructor
Point (const Point& p)
    : x (p.x), y (p.y), z (p.z) {}
```

and not:

```
Point (const Point<T>& p)
```

```
: x (p.x), y (p.y), z (p.z) {}
```

The scope of type argument T extends from the **class** declaration prefix **template <class T>** to the end of the declaration and within this scope. Qualification with the argument type **<T>** can be included, but is not necessary for defining inline member functions. This is not the case for **friend** functions (as we shall see later) because **friend** functions are not implicitly **template** functions, whereas member functions of a **template class** are implicitly **template** functions.

Program PT\_TMP.CPP defines a constructor, four overloaded **operator()** functions and two **friend** functions outside the **template class** declaration. The no-argument constructor is defined as:

```
// no arg. constructor
template <class T>
inline Point<T>::Point ()
: x (0.0), y (0.0), z (0.0) {}
```

When defining constructors and member functions outside a **template class** declaration the keyword **template** and the type arguments must precede each function definition. Thus, such functions are defined by function templates. When defining a **template class** constructor the **template** arguments are specified only once: **Point<T>::Point()** as opposed to **Point<T>::Point<T>()**. Note that while the one- and three-argument constructors of **template class** **Point** can be defined **inline** within the **template class** declaration, the no-argument constructor cannot be defined **inline**. This is because data members **x**, **y** and **z** cannot be initialised without reference to an object.

The **x**, **y** and **z** data members of **Point** are initialised to a floating-point zero, which is OK for **float** and **double** data types but far from ideal for the **int** data type and user-defined classes. An alternative **Point** data member initialisation is to rely on argument T's default constructor:

```
template <class T>
inline Point<T>::Point ()
: x (T()), y (T()), z (T()) {}
```

which would in fact now allow this default constructor to be defined inline. Unfortunately, C++ does not guarantee the value of uninitialised identifiers or objects for integral types or classes which call their default constructor. For instance, replacing the default constructor of program PT\_TMP.CPP with the above constructor and then proceeding to define an uninitialised object with **int** data members:

```
Point<int> p ;
```

produced the following output when I ran the modified program:

```
p: (16669, 9251, 8)
```

which is not how we want the default constructor of **template class** **Point** to behave.

In contrast with defining constructors outside a **template class** declaration, member functions and overloaded operator functions must include the **template** argument(s) (but not the keywords **template** or **class**) prior to the scope resolution operator (**::**):

---

```
template <class T>
inline Point<T>& Point<T>::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}
```

Note that each of the overloaded +, - and \* **operator()** functions returns a **Point<T>** object; **Point<T>::operator+()** is defined as:

```
template <class T>
inline Point<T> Point<T>::operator + (const Point& p)
{
    return Point<T> (x+p.x, y+p.y, z+p.z) ;
}
```

Within the function body of **main()** the three objects p, q and r are defined with **double** data members, while s and t are defined with **int** and **float**, respectively, data members:

```
Point<double> p (1, 2, 3), q (p), r ;
//...
Point<int> s ;
Point<float> t ;
```

Each of the three data types **int**, **float** and **double** is passed as an argument inside the angle brackets (< and >) when an object is defined.

Also, the **main()** function of PT\_TMP.CPP illustrates the use of the **sizeof** operator with both **template class** **Point** and objects of **template class** **Point** for **int**, **float** and **double** data members:

```
cout << "sizeof(Point<int>) : "
      << sizeof (Point<int>)      << endl ;
//...
cout << "sizeof s: " << sizeof s << endl ;
```

The program output illustrates that the size of **template class** **Point** depends on the type or **class** of argument T.

Finally, the following statements in **main()** illustrate an important point concerning **template** classes:

```
//...
int i (0) ;
float f (0.0) ;

f = i ; // o.k.: implicit conversion

Point<int> u ;
Point<int> v ;
Point<float> w ;

u = v ; // o.k.: u & v are of same type
```

```
//w = v ; // error: no implicit conversion
```

We know that C++ implicitly performs conversions from one data type to another (e.g. **int** to **float**), but this does not mean that there is an implicit conversion from **Point<int>** to **Point<float>**. This makes sense when you consider that template **class Point<T>** could have alternatively declared a pointer to the T data member, **T\***, for which there is no implicit conversion from **int\*** to **float\***.

### 13.3.1 Declaring Template Classes in a Header File

At this point it would be a natural step to transfer the declaration and definition of template **class Point** to a .H header file and a .CPP implementation file, respectively, to enable programs other than PT\_TMP.CPP to make use of **template class Point**. Unfortunately, unlike other **non-template** functions, a **template** function definition must accompany its declaration within the same file scope. This is because **template** functions do not have external linkage as do **non-template** functions. This is a far from ideal situation, since the generally adopted convention is not to place C++ code in header files but reserve header files for function and **class** declarations. Also, many commercial compilers will not perform compilation on a header file with a .H,.HXX filename extension, restricting compilation to .C,.CPP,.CXX etc. filename extensions.

The following header file, PT\_TMP.H, thus contains both **Point template class** declaration and function definitions:

```
// pt_tmp.h
// template class Point

#ifndef _PT_TMP_H // prevent multiple includes
#define _PT_TMP_H

// template class Point
template <class T>
class Point
{
private:
    // private data members
    T x, y, z ;
public:
    //...
    //...
};

//...
#endif // _PT_TMP_H
```

The following program #includes PT\_TMP.H:

```
// tmp_tst.cpp
// tests the template class Point which is
// declared and defined in the header file pt_tmp.h
#include <iostream.h> // C++ I/O
#include "pt_tmp.h"    // template class Point
```

---

```
void main ()
{
    Point<int> p (1, 2) ;
    cout << "p: " << p << endl ;
}
```

### 13.3.2 friend Functions

A **friend** function of a **template class** is not implicitly a **template** function, and as a consequence the type argument must be specified in both the declaration and definition of the **friend** function. The following program is a modification of program FRI\_FUNC.CPP presented in Chapter 11:

```
// fri_func.cpp
// illustrates a friend function of a template class
#include <iostream.h> // C++ I/O
#include <math.h> // atan()

template <class T>
class Circle
{
private:
    T radius ;
public:
    // constructors
    Circle (T r)
        : radius (r) {}
    // friend
    friend double Area (const Circle<T>& c) ;
}; // class Circle

// friend function Area() of template class Circle
template <class T>
double Area (const Circle<T>& c)
{
    // area=pi*r^2
    double pi = 4.0 * atan (1.0) ;
    return pi * c.radius * c.radius ;
}

void main ()
{
    Circle<int> circle (5.0) ;

    double area = Area (circle) ;

    cout << "area of circle: " << area << endl ;
}
```

Area() is declared a **friend** of **template class** Circle:

```
friend double Area (const Circle<T>& c) ;
```

and incorporates type T in the declaration.

### 13.3.3 Multiple Type and Non-Type Arguments

When we discussed **template** functions it was noted that all **template** arguments for a function **template** must be type or **class** arguments. Template classes, however, can be declared with both multiple type and non-type arguments. Non-type arguments are rarely used because the same functionality can generally be provided by a constructor. Nevertheless, they do offer a programmer more flexibility in designing **template** classes. Consider the following **template class** Vector, which is a modification of VECTOR.H/.CPP from Chapter 12:

```
// mtca.cpp
// multiple template class arguments
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()

template <class T, int size>
class Vector
{
private:
    T* array; // note: no n_elements data member!
public:
    // constructor
    Vector () ;
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array; array = NULL; }
    // member functions
    int NumberElements () const { return size; }
    T& Value (int index) ;
    const T& Value (int index) const ;
    void Display () ;
}; // template class Vector

// constructor
template <class T, int size>
Vector<T,size>::Vector ()
{
    assert (size) ;

    array = new T[size] ;
    assert (array != NULL) ;
    // zero array
    memset ((void*)array, 0, (size_t)size*sizeof(T)) ;
}

// copy constructor
```

```

template <class T, int size>
Vector<T,size>::Vector (const Vector& v)
{
    assert (size) ;

    array = new T[size] ;
    assert (array != NULL) ;
    // zero array
    memset ((void*)array, 0, (size_t)size*sizeof(T)) ;
    // initialise
    for (int i=0; i<v.NumberElements(); i++)
        array[i] = v.Value (i) ;
}

// returns element value (non-const object)
template <class T, int size>
T& Vector<T,size>::Value (int index)
{
    assert (index >= 0 || index < size) ;
    return array[index] ;
}

// returns element value (const object)
template <class T, int size>
const T& Vector<T,size>::Value (int index) const
{
    assert (index >= 0 || index < size) ;
    return array[index] ;
}

// display a Vector
template <class T, int size>
void Vector<T,size>::Display ()
{
    cout << "[" ;
    for (int i=0; i<size; i++)
    {
        cout << setprecision (2)
            << setiosflags (ios::showpoint | ios::fixed)
            << array[i] ;
        i!=size-1 ? cout << ", " : cout << "]" ;
    }
}

void main ()
{
    Vector<int, 5> v ;

    v.Value (0) = 1 ;
    v.Value (4) = 4 ;
}

```

```

v.Display () ;

Vector<int*, 5> vp ;
int i (0) ;
int* i_ptr = &i ;

vp.Value (0) = i_ptr ;

cout << endl ;
cout << "lvalue i      :" << &i
    << ", rvalue i     :" << i       << endl ;
cout << "lvalue i_ptr:" << &i_ptr
    << ", rvalue i_ptr: " << i_ptr << endl ;
vp.Display () ;
}

```

with output:

```

[1, 0, 0, 0, 4]
lvalue i      : 0x23cf2564, rvalue i      : 0
lvalue i_ptr: 0x23cf2560, rvalue i_ptr: 0x23cf2564
[0x23cf2564, 0x00000000, 0x00000000, 0x00000000, 0x00000000]

```

Template **class** Vector is now declared as:

```

template <class T, int size>
class Vector
{
private:
    T* array ; // note: no n_elements data member!
    //...
};

```

Since the size of a Vector, **size**, is explicitly passed as an argument, the **n\_elements** data member is no longer required. The **private** member function **Allocate()** of the **Vector** **class** of program VECTOR.CPP (Chapter 12) is no longer required, since a single no-argument constructor suffices in the above design of the **template class** Vector.

Note that the above version of **template class** Vector does not overload the insertion and extraction operators (<< and >>). By keeping Vector's array data member **private**, the overloaded functions **operator<<()** and **operator>>()** must be declared **friends** of **template class** Vector so that they have access to Vector's array data member. However, a **friend** function is not within the scope of the **class** in which it is declared, and will thus be defined as a **template** function. It was noted in the previous section that **template** functions cannot be defined with non-type arguments, so the following is illegal:

```

template <class T, int size> // error
ostream& operator << (ostream& s, const Vector<T,size>& v)
{
    //...
}

```

Thus, a *Display()* member function is defined to display the elements of a `Vector` object. A `Vector` object, `v`, of five elements each of type `int` is defined in `main()` as:

```
Vector<int, 5> v ;
```

Note that the non-type arguments in a **template class** declaration must be constant expressions.

The `main()` function of MTCA.CPP also illustrates defining a `Vector<int*>` object whose elements are pointers to type `int`:

```
Vector<int*, 5> vp ;
```

### 13.3.4 Overloading Template Classes?

Unlike **template** functions, **template** classes cannot be overloaded. As a result, the following program code is illegal:

```
template <class T>
class X
{
//...
};

//...
// error: multiple declaration of template class X
template <class T, class S>
class X
{
//...
};
```

However, a **template class** can have an explicit **class** declaration. For instance:

```
template <class T>
class Vector
{
//...
};

//...
class Vector<char*>
{
//...
};
```

The explicit **class** declaration will be used for strings and the **template class** declaration will be used for all other types and classes. Note that the **class** name is accompanied by the specific type of **class** in angle brackets for the explicit **template class** declaration.

The following program illustrates in more detail the above explicit **class** declaration for **template class** `Vector`:

```
// exp_tmp.cpp
// illustrates that a template class can have
```

```
// an explicit template class declaration
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()
#include <string.h> // strlen(), strcpy()

enum Boolean
{ FALSE, TRUE };

class String
{
private:
    char* string ; // pointer to char
    int length ; // length of string
public:
    // constructor
    String (const char str[] = 0) ;
    // copy constructor
    String::String (const String& str) ;
    // destructor
    ~String () ;
    // member function
    int Length () { return length ; }
    // overloaded operators
    String& operator = (const String& str) ;
    char& operator [] (int index) ;
    const char& operator [] (int index) const ;
    // friend
    friend ostream& operator << (ostream& s,
                                         const String& str) ;
}; // class String

// template class Vector
template <class T>
class Vector
{
//...
}; // template class Vector

class Vector<char*>
{
private:
    // data members
    String* array ;
    int n_elements ;
public:
    // constructor
    Vector (int n) ;
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                 n_elements ; }
```

```
// member functions
int NumberElements () const
{ return n_elements ; }
String& Value (int index) ;
const String& Value (int index) const ;
// overloaded operators:
String& operator [] (int index) ;
const String& operator [] (int index) const ;
// friend
friend ostream& operator << (ostream& s,
                                     const Vector<char*>& v) ;
} ; // class Vector<char*>
//...

// Vector<char*>:

// 2 arg. constructor
Vector<char*>::Vector (int n)
{
    assert (n) ;
    array = new String[n] ;
    n_elements = n ;
}

String& Vector<char*>::Value (int index)
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

const String& Vector<char*>::Value (int index) const
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

String& Vector<char*>::operator [] (int index)
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

const String& Vector<char*>::operator [] (int index) const
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

// friend:
// overloaded insertion operator <<
```

```

ostream& operator << (ostream& s, const Vector<char*>& v)
{
    s << "[" ;
    for (int i=0; i<v.n_elements; i++)
    {
        s << v.Value (i) ;
        i!=v.n_elements-1 ? s << ", " : s << "]" ;
    }
    return s ;
}

void main ()
{
    // Vector<T>
    Vector<int> v (5, 0) ;

    v.Value (0) = 1 ;
    v.Value (4) = 4 ;

    cout << v << endl ;

    // Vector<char*>
    Vector<char*> vs (3) ;

    vs[0] = "a string" ;
    vs[1] = "another string" ;
    vs[2] = "and another string" ;

    cout << vs << endl ;

    cout << vs[0][0] << endl ;
}

```

with output:

```

[1, 0, 0, 0, 4]
[a string, another string, and another string]
a

```

The above String **class** is based on the String **class** declared and defined in ND\_STR.CPP of Chapter 12, whereas the above Vector **template class** is based on the Vector **class** declaration and definitions in files VECTOR.H/.CPP of Chapter 12. The **template** Vector **class** listed above is discussed in more detail later in the chapter, when both **template** Vector and Matrix classes are discussed. Although the type of the explicit **class** declaration for **template class** Vector is **char\***, the underlying implementation of **class** Vector<**char\***> is an array of String objects. An array of String objects is used in preference to a double pointer to **char**, **char\*\***, simply because the String **class** is available and because it greatly simplifies the implementation of **class** Vector<**char\***>. However, since **class** String supports dynamic memory allocation, the elements of an object of **class** Vector<**char\***> can be of varying lengths, as illustrated in the program output of EXP\_TMP.CPP.

### 13.3.5 Combined Use of Template Functions and Classes

Clearly, if both **template** functions and classes are correctly implemented the two can be combined. The following program revisits the *Max()* **template** function and the Point **template class**:

```
// tmp_c_f.cpp
// illustrates the use of a template class
// as an argument to a template function
#include <iostream.h> // C++ I/O

enum Boolean { FALSE, TRUE };

// template class Point
template <class T>
class Point
{
private:
    // private data members
    T x, y, z ;
public:
    //...
    // overloaded operators
    //...
    Boolean operator > (const Point& p) ;
    //...
}; // template class Point
//...
// greater than operator
template <class T>
inline Boolean Point<T>::operator > (const Point& p)
{
    return x>p.x && y>p.y && z>p.z ? TRUE : FALSE ;
}
//...
// template function Max()
// returns the maximum of two objects
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}

void main ()
{
    Point<double> p (1, 2), q (2, 3) ;

    Point<double> max = Max (p, q) ;

    cout << "Max (p, q): " << max << endl ;
}
```

with output:

```
Max (p, q): (2, 3, 0)
```

Provided the **template class**, in this case Point, overloads the `>` operator, the `Max()` **template** function works fine. An overloaded `Point::operator>()` function has been added to the **template class** Point in the above program, which performs a comparison on each respective x, y and z data member of two Point objects. Try commenting out the `Point::operator>()` function declaration and definition in the above program. This will generate a compilation error of the form ‘illegal operation’ occurring at the `return` statement of the `Max()` function. Neglecting to provide the necessary overloaded operators for **template** classes used in conjunction with **template** functions can result in having to find some very confusing compilation errors.

### 13.3.6 Nested Classes

A **template class** can contain the declaration of another **class** within its own declaration. For instance, a linked list **class** (`LinkedList`), which is examined in more detail later, contains a nested **Node class**:

```
// nested.cpp
// illustrates nested template classes
#include <iostream.h> // C++ I/O

template <class T>
class LinkedList
{
private:
    // nested Node class
    class Node
    {
public:
        T      data ;
        Node* next ;
        //...
        } * first, * last ;
public:
    LinkedList () {/*...*/}
    //...
};

void main ()
{
    LinkedList<int> ll_obj ;
}
```

The nested **class** `Node` is declared **private** to `LinkedList<>` because it is an implementation detail of `LinkedList<>`, and an object of `LinkedList<>` should not have direct access to `Node`'s data members or member functions. Also, it makes no sense to define an object of **class** `Node` except in the context of a `LinkedList<>` object. In the present case it is

quite legitimate to make `Node` a nested **class**, since it is a relatively simple **class**. More complex classes are generally, however, best declared separately rather than nested.

Note that the nested **class** `Node` encapsulates a pointer called `next`, also of **class** `Node`! This is an example of *recursive class declaration*, and is a frequently used technique in the design of linked lists.

### 13.3.7 The **typename** Keyword

In the previous subsection we saw that a **class** declaration can be nested within another **template class** declaration. If, however, a name is used within a **template class** declaration that is not explicitly declared to be a type or **class**, it is assumed that the name does not name a type unless the name is qualified by the **typename** keyword:

```
// typename.cpp
// illustrates typename

#include <iostream.h> // C++ I/O

template <class T>
class A
{
private:
    T data ;
public:
    A ()
        : data () {}
    class X // local class X
    {
    };
    X x ; // o.k.: T::X is a type name
    Y y ; // error: Y is not a type name
    T::Z z ; // error: T::Z is not a type name
    typename T::I i ; // o.k.: T::I is a type name
    typename T::I* ip ; // o.k. pointer to T::I
};

class B
{
public: // B::I publicly accessible
    class I
    {
    };
};

template <typename T> // typename replacing class
T Max (T a, T b)
{
    return a > b ? a : b ;
}

void main ()
```

```

{
A<B> a ;

int x (1), y (2) ;
cout << "max (x, y): " << Max (x, y) << endl ;
}

```

Use of the type name X within **template class** A is legal because X is explicitly declared within A, whereas Y and T : : Z are assumed not to be type names. The use of **typename** within A indicates that T : : I is a type name. The above program also illustrates that the **typename** keyword can be used to replace the **class** keyword in a **template** declaration.

## 13.4 Smart Pointers

Let us develop a **class** called XPtr which operates like a C++ pointer to another **class** called X:

```

// smrt_ptr.cpp
// illustrates smart pointers
#include <iostream.h> // C++ I/O

// X class
class X
{
private:
    int x_data ;
public:
    X ()
        : x_data (0) {}
    int Get ()
        { return x_data ; }
};

// X pointer class
class XPtr
{
private:
    X* ptr ;
public:
    // constructor
    XPtr (X* x)
        : ptr (x) {}
    // overloaded operator
    X* operator -> () const
        { return ptr ; }
    // conversion function
    operator X* () const
        { return ptr ; }
};

```

```

// smart pointer template class
template <class T>
class Ptr
{
private:
    T* ptr ;
public:
    // constructor
    Ptr (T* p)
        : ptr (p) {}
    // overloaded operator
    T* operator -> () const
        { return ptr ; }
    // conversion function
    operator T* () const
        { return ptr ; }
};

void main ()
{
    X x ;

    XPtr xp = &x ;
    cout << xp->Get () << endl ;
    X* x_ptr0 = xp ;

    Ptr<X> xtp = &x ;
    cout << xtp->Get () << endl ;
    X* x_ptr1 = xtp ;
}

```

The `main()` function illustrates that an object of `class XPtr`, `xp`, acts as if it is a C++ pointer to `class X` – often referred to as a *smart pointer*. This is made possible by overloading the `operator->()` function for `class XPtr`, which encapsulates a `private` data member pointer to `class X`. `XPtr` also supports a conversion function `operator X*()`, which enables a pointer to `class X` to be defined and initialised to a `XPtr` object.

The smart pointer `class XPtr` is a perfect candidate to make a `template class`, since in general smart pointers are independent of the `class` of an object that they point to. Such a smart pointer `template class`, `Ptr`, is declared in the above program and demonstrated in `main()`.

## 13.5 A GlobalMemory Template class for Windows

Before discussing `Vector` and `Matrix` `template` classes let's briefly revisit the Global-Memory `class` discussed in Chapter 12:

```

// win_tmp.cpp
// illustrates a global memory template class
// for programming in a Windows environment

```

```

//...
template <class T>
class GlobalMemory
{
private:
    HGLOBAL hGlobalMemory ; // handle to memory
    void FAR* mem_address ; // address of memory
public:
    // constructor
    GlobalMemory ()
        : hGlobalMemory (NULL), mem_address (NULL) {} ;
    //...
};

//...
// allocate memory
template <class T>
void FAR* GlobalMemory<T>::Allocate (UINT size, UINT
                                         mem_flags)
{
    hGlobalMemory = GlobalAlloc (mem_flags,
                                size*sizeof(T)) ;
    //...
    return mem_address ;
}

// re-allocate memory
template <class T>
void FAR* GlobalMemory<T>::ReAllocate (UINT size, UINT
                                         mem_flags)
{
    HGLOBAL h = GlobalReAlloc (hGlobalMemory,
                               size*sizeof(T), mem_flags) ;
    //...
    return mem_address ;
}

void main ()
{
    const int SIZE = 10 ;

    GlobalMemory<int> g_mem ; // global memory object

    // allocate memory
    int* px = (int*) g_mem.Allocate (SIZE) ;
    // re-allocate memory
    px = (int*) g_mem.ReAllocate (2*SIZE) ;
    //...
}

```

The GlobalMemory **template class** relieves a user of the **class** of having to incorporate the size of the data type or **class** to which the memory object refers:

---

```
int* px = (int*) g_mem.Allocate (SIZE) ;
```

instead of having to indicate the size of the data type or **class** explicitly in the *Allocate()* or *ReAllocate()* member function calls, as was the case for the non-**template** Global-Memory **class** of Chapter 12:

```
int* px = (int*) g_mem.Allocate (SIZE*sizeof(int)) ;
```

In addition, we could have performed the casting of the far pointer, **void FAR\***, returned by the Windows API function *GlobalLock()* to **T\*** within the *Allocate()* and *ReAllocate()* member functions:

```
template <class T>
T* GlobalMemory<T>::Allocate (UINT size, UINT mem_flags)
{
    hGlobalMemory = GlobalAlloc (mem_flags, size*sizeof(T)) ;
    //...
    mem_address = GlobalLock (hGlobalMemory) ;
    return (T*)mem_address ;
}
```

thus relieving a user of explicitly having to perform the casting:

```
int* px = g_mem.Allocate (SIZE) ;
```

This procedure was not adopted in WIN\_TMP.CPP because one of three pointers (near, far or huge) can be used to point to a block of memory allocated by Windows, depending on the size of the block. When a single object is stored in global memory and is less than the 65 536 bytes of a segment, far pointers can be used. If a single object is greater than 65 536 bytes, huge pointers are required. Letting the *Allocate()* and *ReAllocate()* member functions return a far pointer to **void** greatly increases the generality of the functions by allowing a user of the **class** to perform the correct casting.

## 13.6 Vector and Matrix Template Classes

This section discusses Vector and Matrix **template** classes. These classes are modifications of the non-**template** Vector and Matrix classes discussed towards the end of Chapter 12 (see VECTOR.H/.CPP, MATRIX.H/.CPP and MTX\_TST.CPP). The **template** Vector and Matrix classes are tested below in program MTMP\_TST.CPP, which also makes use of **template class** Point, which is declared and defined in PT\_TMP.H and is identical to the PT\_TMP.H header file discussed earlier:

```
// pt_tmp.h
// template class Point

#ifndef _PT_TMP_H // prevent multiple includes
#define _PT_TMP_H

#include <iostream.h> // C++ I/O
```

---

```
// template class Point
template <class T>
class Point
{
private:
    // private data members
    T x, y, z;
public:
    //...
}; // template class Point
//...
#endif // _PT_TMP_H
```

Template classes Vector and Matrix are declared and defined in VEC\_TMP.H and MTX\_TMP.H, respectively, and are both listed in full below because several modifications have been made to the non-**template** classes presented in Chapter 12:

```
// vec_tmp.h
// template class Vector

#ifndef _VEC_TMP_H // prevent multiple includes
#define _VEC_TMP_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()

// template class Vector
template <class T>
class Vector
{
private:
    // data members
    T* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector () { Allocate (3) ; }
    Vector (int n, T obj) ;
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                  n_elements = 0 ; }
    // member functions
    int NumberElements () const { return n_elements ; }
    T& Value (int index) ;
    const T& Value (int index) const ;
    void New (int new_n) ;
```

```
// overloaded operators
Vector& operator = (const Vector& v) ;
T& operator [] (int index) ;
const T& operator [] (int index) const ;
Vector operator + (const Vector& v) ;
Vector operator - (const Vector& v) ;
Vector operator * (const Vector& v) ;
// friend
friend ostream& operator << (ostream& s,
                                const Vector<T>& v) ;
}; // class Vector

// private member function:

// allocates memory for elements of Vector<T>
template <class T>
void Vector<T>::Allocate (int n)
{
    assert (n) ;

    array = new T[n] ;
    assert (array != NULL) ;
    n_elements = n ;
    for (int i=0; i<n; i++)
        array[i] = T () ;
}

// 2 arg. constructor
template <class T>
Vector<T>::Vector (int n, T obj)
{
    Allocate (n) ;
    for (int i=0; i<n; i++)
        array[i] = obj ;
}

// copy constructor
template <class T>
Vector<T>::Vector (const Vector& v)
{
    Allocate (v.NumberElements()) ;
    for (int i=0; i<v.NumberElements(); i++)
        *(array+i) = v.Value (i) ;
}

// public member functions:

// returns element value (non-const object)
template <class T>
T& Vector<T>::Value (int index)
{
```

```
assert (index >= 0 || index < n_elements) ;
return *(array+index) ;
}

// returns element value (const object)
template <class T>
const T& Vector<T>::Value (int index) const
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

// change number of elements
template <class T>
void Vector<T>::New (int new_n)
{
    delete [] array ;
    array      = NULL ;
    n_elements = 0 ;

    Allocate (new_n) ;
}

// overloaded operators:

// assignment operator =
template <class T>
Vector<T>& Vector<T>::operator = (const Vector& v)
{
    // verify that the two vectors are of equivalent dimensions
    assert(n_elements == v.NumberElements()) ;

    for (int i=0; i<v.NumberElements(); i++)
        *(array+i) = v.Value (i) ;
    return *this ;
}

// subscript operator [] (non-const object)
template <class T>
T& Vector<T>::operator [] (int index)
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
}

// subscript operator [] (const object)
template <class T>
const T& Vector<T>::operator [] (int index) const
{
    assert (index >= 0 || index < n_elements) ;
    return *(array+index) ;
```

```
}
```

```
// addition operator +
template <class T>
Vector<T> Vector<T>::operator + (const Vector& v)
{
    // verify that the two vectors are of equivalent dimensions
    assert(n_elements == v.NumberElements()) ;

    Vector<T> add (v.NumberElements(),T()) ;

    for (int i=0; i<v.NumberElements(); i++)
        add.Value (i) = Value (i) + v.Value (i) ;
    return add ;
}
```

```
// subtraction operator -
template <class T>
Vector<T> Vector<T>::operator - (const Vector& v)
{
    // verify that the two vectors are of equivalent dimensions
    assert(n_elements == v.NumberElements()) ;

    Vector<T> sub (v.NumberElements(),T()) ;

    for (int i=0; i<v.NumberElements(); i++)
        sub.Value (i) = Value (i) - v.Value (i) ;
    return sub ;
}
```

```
// multiplication operator *
template <class T>
Vector<T> Vector<T>::operator * (const Vector& v)
{
    // verify that the two vectors are of equivalent dimensions
    assert(n_elements == v.NumberElements()) ;

    Vector<T> mult (v.NumberElements(),T()) ;

    for (int i=0; i<v.NumberElements(); i++)
        mult.Value (i) = Value (i) * v.Value (i) ;
    return mult ;
}
```

```
// friend:
```

```
// overloaded insertion operator <<
template <class T>
ostream& operator << (ostream& s, const Vector<T>& v)
{
    s << "[" ;

```

```

for (int i=0; i<v.n_elements; i++)
{
    s << setprecision (2)
    << setiosflags (ios::showpoint | ios::fixed)
    << v.Value (i) ;
    i!=v.n_elements-1 ? s << ", " : s << "]";
}
return s ;
}

#ifndef // _VEC_TMP_H

// mtx_tmp.h
// template class Matrix

#ifndef _MTX_TMP_H // prevent multiple includes
#define _MTX_TMP_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()
#include <math.h> // fabs()
#include "vec_tmp.h" // template class Vector

const double TOLERANCE = 1e-06 ;

// swap two objects
template <class T>
void Swap (T& a, T& b)
{
    T temp = a ;
    a = b ;
    b = temp ;
}

enum Boolean { FALSE, TRUE } ;

// template class Matrix
template <class T>
class Matrix
{
private:
    // data members
    Vector<T>** array ;
    int rows, cols ;
    // member function
    void Allocate (int r, int c, T obj) ;
public:
    // constructors
    Matrix () ;
    Matrix (int r, int c, T init_obj) ; // no def. args
}

```

```

// copy constructor
Matrix (const Matrix& m) ;
// destructor
~Matrix () ;
// member functions
T& Value (int row, int col) ;
const T& Value (int row, int col) const ;
int Rows () const { return rows ; }
int Cols () const { return cols ; }
Vector<T> GaussElimination (Vector<T>& b,
                             Boolean& solved) ;

// overloaded operators
Matrix& operator = (const Matrix& m) ;
Vector<T>& operator [] (int index) ;
const Vector<T>& operator [] (int index) const ;
Matrix operator + (const Matrix& m) ;
Matrix operator - (const Matrix& m) ;
Matrix operator * (const Matrix& m) ;
// friend
friend ostream& operator << (ostream& s,
                                 const Matrix<T>& m) ;
}; // class Matrix

// private member function:

// allocate sufficient memory for Matrix
template <class T>
void Matrix<T>::Allocate (int r, int c, T obj)
{
    // array of pointers to Vector<T>
    array = (Vector<T>**)new Vector<T>[r] ;
    assert (array != NULL) ;

    for (int i=0; i<r; i++)
    {
        // array of Vector<T>'s
        array[i] = new Vector<T> (c, obj) ;
        assert (array[i] != NULL) ;
    }

    rows = r ; // set row & column dimensions
    cols = c ;
}

// no arg. constructor
template <class T>
Matrix<T>::Matrix ()
{
    Allocate (1, 1, T()) ; // (1x1) default matrix

    // initialise matrix to T's default constructor
}

```

```
Value (1, 1) = T() ;
}

// 3 arg. constructor
template <class T>
Matrix<T>::Matrix (int r, int c, T obj)
{
    Allocate (r, c, init_obj) ;

    // initialise matrix
    for (int k=0; k<r; k++)
        for (int l=0; l<c; l++)
            Value (k, l) = obj ;
}

// copy constructor
template <class T>
Matrix<T>::Matrix (const Matrix& m)
{
    // array of pointers to Vector<T>
    array = (Vector<T>**) new Vector<T>[m.Rows()] ;
    assert (array != NULL) ;

    for (int i=0; i<m.Rows(); i++)
    {
        // array of Vector<T>'s
        array[i] = new Vector<T> (m.Cols(), T()) ;
        assert (array[i] != NULL) ;
    }

    rows = m.Rows() ; // set row & column dimensions
    cols = m.Cols() ;

    // initialise matrix
    for (int k=0; k<m.Rows(); k++)
        for (int l=0; l<m.Cols(); l++)
            Value (k, l) = m.Value (k, l) ;
}

// destructor
template <class T>
Matrix<T>::~Matrix ()
{
    for (int i=0; i<rows; i++)
        delete array[i] ;
    delete [] array ;

    rows = cols = 0 ;
}

// public member functions:
```

```

// returns element value (non-const object)
template <class T>
T& Matrix<T>::Value (int row, int col)
{
    assert (row >= 0 && row < rows && col >=0
            && col < cols) ;
    return array[row]->Value(col) ;
}

// returns element value (const object)
template <class T>
const T& Matrix<T>::Value (int row, int col) const
{
    assert (row >= 0 && row < rows && col >=0
            && col < cols) ;
    return array[row]->Value(col) ;
}

// Gauss elimination
template <class T>
Vector<T> Matrix<T>::GaussElimination (Vector<T>& b,
Boolean& solved)
{
    solved = FALSE ;

    int n = rows ; // assign number of rows to n

    Vector<T> x (n, T()) ; // solution Vector

    // Gaussian elimination:

    for (int i=0; i<n; i++)
    {
        T max_pivot = fabs (Value (i, i)) ;
        int pivot_row = i ;

        // find maximum pivot & row in which it occurs
        for (int k=i+1; k<n; k++)
        {
            if (fabs(Value(k, i)) > max_pivot)
            {
                max_pivot = fabs (Value (k, i)) ;
                pivot_row = k ;
            }
        }

        if (max_pivot > TOLERANCE) // if pivot o.k.
        {
            solved = TRUE ;
        }
    }

    // if pivot found, swap rows
}

```

```

if (i != pivot_row)
{
    for (int j=0; j<n; j++)
        Swap (Value (i, j),
              Value (pivot_row, j)) ;
    Swap (b[i], b[pivot_row]) ;
}
for (int j=i+1; j<n; j++)
{
    T multiplier = Value (j, i) / Value (i, i) ;
    for (int k=i; k<n; k++)
        Value (j, k) -= multiplier *
                        Value (i, k) ;
    b[j] -= multiplier * b[i] ;
}
}
else // if pivot not o.k.
{
    solved = FALSE ;
    return x ; // return zero solution Vector
}
}

// back substitution:

// (n-1) element
x[n-1] = b[n-1] / Value (n-1, n-1) ;

// for i=n-2,n-3,...0
for (int s=n-2; s>=0; s--)
{
    T sum (0.0) ; // assuming floating-point type T!
    for (int t=s+1; t<=n-1; t++)
        sum += Value (s, t) * x[t] ;
    x[s] = (b[s] - sum) / Value (s, s) ;
}
return x ;
} // GaussElimination()

// overloaded operators:

// assignment operator =
template <class T>
Matrix<T>& Matrix<T>::operator = (const Matrix& m)
{
    // verify that the two matrices are of equivalent
    // dimensions
    assert(rows == m.Rows() && cols == m.Cols()) ;

    for (int i=0; i<rows; i++)
        for (int j = 0; j<cols; j++)

```

```
        Value (i, j) = m.Value (i, j) ;
    return *this ;
}

// subscript operator [] (non-const object)
template <class T>
Vector<T>& Matrix<T>::operator [] (int index)
{
    // verify that index value is in range
    assert (index >= 0 && index < rows) ;
    return *array[index] ;
}

// subscript operator [] (const object)
template <class T>
const Vector<T>& Matrix<T>::operator [] (int index) const
{
    // verify that index value is in range
    assert (index >= 0 && index < rows) ;
    return *array[index] ;
}

// addition operator +
template <class T>
Matrix<T> Matrix<T>::operator + (const Matrix& m)
{
    assert (rows == m.Rows() && cols == m.Cols()) ;

    Matrix<T> add (rows, cols, T()) ;

    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            add.Value (i, j) = Value (i, j) + m.Value (i, j) ;
    return add ;
}

// subtraction operator -
template <class T>
Matrix<T> Matrix<T>::operator - (const Matrix& m)
{
    assert (rows == m.Rows() && cols == m.Cols()) ;

    Matrix<T> sub (rows, cols, T()) ;

    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            sub.Value (i, j) = Value (i, j) - m.Value (i, j) ;
    return sub ;
}

// multiplication operator *
```

```

template <class T>
Matrix<T> Matrix<T>:: operator * (const Matrix& m)
{
    assert (cols == m.Rows()) ;

    Matrix<T> result (rows, m.Cols(), T()) ;

    for (int i=0; i<rows; i++)
    {
        for (int j=0; j<m.Cols(); j++)
        {
            T sum (0.0) ; // assuming floating-point type T!
            for (int k=0; k<cols; k++)
                sum += Value (i, k) * m.Value(k, j) ;
            result.Value(i, j) = sum ;
        }
    }
    return result ;
}

// friend:

// insertion operator <<
template <class T>
ostream& operator << (ostream& s, const Matrix<T>& m)
{
    for (int i=0; i<m.rows; i++)
    {
        for (int j=0; j<m.cols; j++)
            s << setprecision (2)
                << setiosflags (ios::showpoint | ios::fixed)
                << setw (10)
                << m.Value (i, j) ;
        s << endl ;
    }
    return s ;
}

#endif // _MTX_TMP_H

```

The majority of the **template** features of **template class** Vector and Matrix are similar to those discussed with reference to the Point **template class**. The **Vector<T>::Allocate()** **private** member function now uses the default constructor of argument type T to initialise the elements of a Vector object:

```

template <class T>
void Vector<T>::Allocate (int n)
{
    //...
    for (int i=0; i<n; i++)
        array[i] = T () ;
}

```

---

 }

whereas the two-argument constructor initialises the newly allocated elements of a `Vector` object to the second constructor argument object of type `T`:

```
template <class T>
Vector<T>::Vector (int n, T obj)
{
//...
for (int i=0; i<n; i++)
    array[i] = obj ;
}
```

Similarly, the `Allocate()` **private** function of the `Matrix` **template class** is passed an object of type `T` to initialise the elements of a `Matrix` object:

```
template <class T>
void Matrix<T>::Allocate (int r, int c, T obj)
{
// array of pointers to Vector<T>
array = (Vector<T>**) new Vector<T>[r] ;
assert (array != NULL) ;

for (int i=0; i<r; i++)
{
// array of Vector<T>'s
array[i] = new Vector<T> (c, obj) ;
assert (array[i] != NULL) ;
}

rows = r ; // set row & column dimensions
cols = c ;
}
```

Remember to include the argument type `T` when requesting memory with the `new` operator (`new Vector<T>`). Template **class** `Matrix` has a three-argument constructor. The first two arguments define the row and column dimensions of a `Matrix` object and the third argument is an object of type `T` which is used to initialise the elements of the newly allocated `Matrix` object.

So, let's see the above **template** classes in action:

```
// mtmp_tst.cpp
// tests the Point, Vector and Matrix template classes
#include <iostream.h> // C++ I/O

#include "pt_tmp.h" // template class Point
#include "vec_tmp.h" // template class Vector
#include "mtx_tmp.h" // template class Matrix

void main ()
{
```

```

//Matrix<float> a (3, 3, 0.0) ;
//Vector<float> b (3, 0.0), x (3, 0.0) ;
Matrix<double> a (3, 3, 0.0) ;
Vector<double> b (3, 0.0), x (3, 0.0) ;

a[0][0] = 1.0 ; a[0][1] = 2.0 ; a[0][2] = 3.0 ;
a[1][0] = 2.0 ; a[1][1] = 3.0 ; a[1][2] = 4.0 ;
a[2][0] = 3.0 ; a[2][1] = 4.0 ; a[2][2] = 1.0 ;
b[0] = 14.0 ; b[1] = 20.0 ; b[2] = 14.0 ;

cout << "Matrix<double> a: " << endl << a ;
cout << "Vector<double> b: " << endl << b << endl ;

Boolean bool = FALSE ;

x = a.GaussElimination (b, bool) ;

if (bool)
    cout << "soln. Vector<double> x of system [a]{x}={b}: "
        << endl << x << endl ;
else
    cout << "soln. by Gauss elimination failed" << endl ;

cout << endl ;

Point< Point<int> > ppi ; // composed type

cout << "ppi: " << ppi << endl ;

Vector<double> v_init ;
Vector< Vector<double> > vvd (2, v_init); // composed type

cout << "vvd: " << vvd << endl ;

// not supported yet!
//Point<double> p_init ;
//Matrix < Point<double> > mpd (2, 2, p_init) ;
}

```

with program output:

```

Matrix<double> a:
    1.00      2.00      3.00
    2.00      3.00      4.00
    3.00      4.00      1.00
Vector<double> b:
[14.00, 20.00, 14.00]
soln. Vector<double> x of system [a]{x}={b}:
[1.00, 2.00, 3.00]

ppi: ((0, 0, 0), (0, 0, 0), (0, 0, 0))

```

---

```
vvd: [[0.00, 0.00, 0.00], [0.00, 0.00, 0.00]]
```

Vector and Matrix objects, with **double** elements, are defined as follows:

```
Matrix<double> a (3, 3, 0.0) ;
Vector<double> b (3, 0.0), x (3, 0.0) ;
```

Program MTMP\_TST.CPP also illustrates the *composed type* or *nested template* feature of templates, where a **template class** is itself a **template argument**:

```
Point< Point<int> > ppi ;
```

which defines an object ppi with elements of type `Point<int>`. The program output illustrates the three `Point<int>` elements of object ppi – are templates amazing or what?

Pay particular attention to the space between the two `>` operators, which is required so that the token `> >` is not confused with the right-shift or extraction operators. The Borland C++ (version 5.0) compiler issues a compilation warning (not an error) of the form ‘use `> >` instead of `>>` for nested templates’ if the token `>>` is used.

MTMP\_TST.CPP also defines a Vector object vvd of a composed type consisting of two sets of three `Vector<double>` elements. Program MTMP\_TST.CPP finally contains the following commented-out statement:

```
Matrix < Point<double> > mpd (2, 2, p_init) ;
```

This statement results in a series of compilation errors because the **template class** Matrix is not completely type-independent. The `GaussElimination()` member function and the overloaded `operator*` () function both contain several elements which are dependent on floating-point numbers:

```
template <class T>
Vector<T> Matrix<T>::GaussElimination (Vector<T>& b,
Boolean& solved)
{
//...
for (int i=0; i<n; i++)
{
// floating-point abs.
T max_pivot = fabs (Value (i, i));
//...
// find maximum pivot & row in which it occurs
for (int k=i+1; k<n; k++)
{
// floating-point abs.
if (fabs (Value (k, i)) > max_pivot)
{
// floating-point abs.
max_pivot = fabs (Value (k, i)) ;
//...
}
}
//...
```

```

// back substitution:
//...
for (int s=n-2; s>=0; s--)
{
    T sum (0.0) ; // assuming floating-point type T!
    //...
}
return x ;
} // GaussElimination()

template <class T>
Matrix<T> Matrix<T>:: operator * (const Matrix& m)
{
//...
for (int i=0; i<rows; i++)
{
    for (int j=0; j<m.Cols(); j++)
    {
        T sum (0.0) ; // assuming floating-point type T!
        //...
    }
}
return result ;
}

```

The above dependencies are a result of the Gaussian elimination and the matrix multiplication algorithms being hard-wired to floating-point and integer arithmetic. This is further demonstrated by calling the *GaussElimination()* function for a **Matrix** object with **int** data type elements. When we discuss the combined use of templates and inheritance in Chapter 15 a possible solution to this problem will be presented.

When designing **template** classes and functions, ensure that there is no inherent type or **class** dependency in the implementation of the **template**, because a user will undoubtedly subject the **template** to a whole range of different types, classes and contexts.

## 13.7 Iterators

Frequently, when dealing with objects that encapsulate a collection of other objects or elements, we need a mechanism of traversing some or all of the object elements. Such an iteration process is generally implemented as an iterative block within a program. Alternatively, we can implement the iterative process as a member function of a **class** or specifically define an iterator **class** whose sole purpose is to iterate through the elements of an object. Let us first examine the design of iterator functions for **template class Vector**.

### 13.7.1 Iterator Functions

In practice, we are frequently required to perform a given operation on all or some of the elements of an array. For instance, consider truncating the **int** elements of a **Vector** object which are greater than a certain value:

---

```

const int MAX = BIG ;
//...
void main ()
{
    Vector v ;
    //...
    for (int i=0; i<v.NumberElements(); i++)
        if (v[i] >= MAX)
            v[i] = MAX ;
    //...
}

```

This works fine, but it would be nice to hide away the **for**-loop that iterates over the object elements and make the **Vector** object responsible for performing the iteration. One approach is first to define an iterator function which performs the object element operation:

```

void IteratorFunction (int& i)
{
    if (i >= MAX)
        i = MAX ;
}

```

and then simply to pass this function to a member function, *ForEachElement()* of **class Vector** which will iterate through the elements (however many or few there may be) of a **Vector** object:

```

const int MAX = BIG ;
//...
void main ()
{
    Vector v ;
    //...
    v.ForEachElement (IteratorFunction) ;
    //...
}

```

Also, we frequently need to test all or some of the elements of an object against a given condition to find the first or last element which satisfies the condition. Similarly, we can pass a user-supplied test condition function to *FirstElement()* or *LastElement()* member functions which perform the necessary iteration:

```

//...
int TestConditionFunction (const int& i)
{
    //...
}
//...
void main ()
{
    Vector v ;
    //...
}

```

```

int v_first = v.FirstElement (TestConditionFunction) ;
//...
int v_last = v.LastElement (TestConditionFunction) ;
//...
}

```

The above form of iterator is illustrated below for **template class Vector**:

```

// v_it_f.h
// template class Vector with iterators

#ifndef _V_IT_F_H // prevent multiple includes
#define _V_IT_F_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()
//...
// template class Vector
template <class T>
class Vector
{
//...
public:
//...
    void      New (int new_n) ;
    void      ForEachElement (void (*fp)(T&), int first,
                           int last) ;
    int       FirstElement (Boolean (*fp)(const T&),
                           int first, int last) const ;
    int       LastElement (Boolean (*fp)(const T&),
                           int first, int last) const ;
//...
}; // template class Vector
//...
// change number of elements, maintaining old data
template <class T>
void Vector<T>::New (int new_n)
{
    assert (new_n > n_elements) ;

    T* temp_array = new T[new_n] ;
    assert (temp_array != NULL) ;

    for (int i=0; i<new_n; i++) // initialise temp. array
        *(temp_array+i) = T () ;

    for (int j=0; j<n_elements; j++) // copy over old data
        *(temp_array+j) = *(array+j) ;
    delete [] array ;
    array      = temp_array ;
}

```

```

n_elements = new_n ;
}

// iterate through elements of a Vector object
template <class T>
void Vector<T>::ForEachElement (void (*fp) (T&), int first,
                                int last)
{
    assert (first < last) ;
    for (int i=first; i<=last; i++)
        fp (*array+i)) ;
}

// returns the index to the first element of a Vector object
// that satisfies a condition, else -1
template <class T>
int Vector<T>::FirstElement (Boolean (*fp) (const T&),
                               int first, int last) const
{
    assert (first < last) ;
    assert (first >= 0 && first < n_elements-1) ;
    for (int i=first; i<=last; i++)
        if (fp (*(array+i)) != FALSE)
            return i ;
    return -1 ;
}

// returns the index of the first element of a Vector object
// that satisfies a condition, else -1
template <class T>
int Vector<T>::LastElement (Boolean (*fp) (const T&),
                               int first, int last) const
{
    assert (first < last) ;
    assert (last > 0 && last <= n_elements-1) ;
    for (int i=last; i>=first; i--)
        if (fp (*(array+i)) != FALSE)
            return i ;
    return -1 ;
}
//...
#endif // _V_IT_F_H

```

The definitions of the member functions *ForEachElement()*, *FirstElement()* and *LastElement()* have been extended so that a user can specify a range of object elements in which the iterator and test condition functions will operate<sup>2</sup>. Let us first examine the *ForEachElement()* iterator member function definition:

---

<sup>2</sup> The operation of functions *ForEachElement()*, *FirstElement()* and *LastElement()* are similar in principle to the functions *ForEach()*, *FirstThat()* and *LastThat()*, respectively, supported by the Borland C++ (version 5.0) Array and associated container classes.

---

```
template <class T>
void Vector<T>::ForEachElement (void (*fp)(T&), int first,
int last)
{
    assert (first < last) ;
    for (int i=first; i<=last; i++)
        fp (*(array+i)) ;
}
```

*ForEachElement()* is passed a pointer to a user-supplied iterator function with the following signature for type T:

```
void IteratorFunction (T&) ;
```

*ForEachElement()* simply loops through the specified range of elements, passing each object element to the iterator function.

The *FirstElement()* function loops through the specified range of elements, passing each object element to the test condition function. If the test condition function returns TRUE, then *FirstElement()* returns the index of the first element in the specified range to pass the test. If no element within the specified range passes the test then a value of -1 is returned to indicate that the test was unsuccessful:

```
template <class T>
int Vector<T>::FirstElement (Boolean (*fp)(const T&),
                                int first, int last) const
{
    assert (first < last) ;
    assert (first >= 0 && first < n_elements-1) ;
    for (int i=first; i<=last; i++)
        if (fp (*(array+i)) != FALSE)
            return i ;
    return -1 ;
}
```

and similarly for *LastElement()*.

Aside from iterators, the *Vector::New()* member function has been modified in VEC\_ITER.H to allow the size of a *Vector* object to be increased while maintaining the object's original elements. A program which tests the above iterators and the *Vector::New()* member function is given in ITER\_TST.CPP:

```
// it_f_tst.cpp
// tests the template class Vector with iterators

#include "vec_iter.h" // template class Vector

// iterator function
void ITFunction (int& i)
{
    i = 5 ;
}
```

```

// test condition function
Boolean TCFUNCTION (const int& i)
{
    return i == 5 ? TRUE : FALSE ;
}

void main ()
{
    Vector<int> v (5, 0) ;
    cout << "v: " << v << endl ;

    v.ForEachElement (ITFunction, 0, 4) ;
    cout << "v: " << v << endl ;

    Vector<int> u (5, 0) ;
    u[2] = 5 ;

    cout << "first: " << u.FirstElement (TCFunction, 0,
                                           u.NumberElements()-1)
        << endl ;
    cout << "last : " << u.LastElement (TCFunction, 0, 1)
        << endl ;

    u.New (10) ;
    cout << "u: " << u << endl ;
}

```

with output:

```

v: [0, 0, 0, 0, 0]
v: [5, 5, 5, 5, 5]
first: 2
last : -1
u: [0, 0, 5, 0, 0, 4908, 4908, 4908, 4908, 4908]

```

### 13.7.2 Iterator Classes

An alternative approach to iterating through the elements of a `Vector` object is to develop a `VectorIterator` **class** which provides access to a `Vector` object's elements. An object of an iterator **class** hides the details of a `Vector` object and how its elements are accessed from a user, thus eliminating the common user error of stepping out of bounds when indexing the elements of a `Vector` object in a loop construct. The following header file declares a `VectorIterator` **class**:

```

// v_it_c.h
// template class VectorIterator

#ifndef _V_IT_C_H // prevent multiple includes
#define _V_IT_C_H

```

```

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()

enum Boolean { FALSE, TRUE };

// template class Vector
template <class T>
class Vector
{
private:
    // data members
    T* array ;
    int n_elements ;
    //...
public:
    //...
    friend class VectorIterator<T> ;
}; // template class Vector

// template class VectorIterator
template <class T>
class VectorIterator
{
private:
    int c_elem ; // current element
    const Vector<T>* vec ; // current Vector
public:
    // constructor
    VectorIterator (const Vector<T>& v)
        : c_elem (0), vec (&v) {}
    // member functions
    Boolean More () const
    { return c_elem < vec->n_elements ? TRUE : FALSE ; }
    T& Current () const { return vec->array[c_elem]; }
    void Next () { c_elem++ ; }
    void ReStart () { c_elem = 0 ; }
}; // template class VectorIterator
//...
#endif // _V_IT_C_H

```

VectorIterator is made a **friend** of **class** Vector so that the declaration of VectorIterator has direct access to Vector's **private** data members, array and n\_elements.

VectorIterator has two **private** data members, c\_elem and vec. The **int** data member c\_elem acts as an index to an element of a Vector object which is currently being pointed to. The vec data member is a pointer to a Vector object which a VectorIterator iterator object operates on. Both c\_elem and vec are initialised when an iterator object is defined via the one-argument constructor, which associates a Vector object with a VectorIterator iterator object.

Class VectorIterator supports four member functions. The function *More()* returns logical-true if more elements remain, from the current position, in a Vector object. Cur-

*rent()* returns the element object at the current position, *Next()* advances the current position to the next element, and *ReStart()* returns the current position to the start of the element list.

In the implementation of *VectorIterator*, above, the *vec* data member was declared a pointer to a *Vector* object. Alternatively, *vec* can be declared a reference to *Vector*:

```
template <class T>
class VectorIterator
{
private:
    int             c_elem ; // current element
    const Vector<T>& vec ; // current Vector
public:
    // constructor
    VectorIterator (const Vector<T>& v)
        : c_elem (0), vec (v) {}
    //...
};
```

A program which tests *VectorIterator* is:

```
// it_c_tst.cpp
// tests the template class VectorIterator

#include "v_it_c.h" // template classes Vector and
VectorIterator

void main ()
{
    Vector<int> v (5, 0);

    // iterate thro' Vector object elements using integer i
    for (int i=0; i<v.NumberElements(); i++)
        cout << v.Value (i) << " " ;

    cout << endl ;

    // iterate thro' Vector object elements using iterator
    // object
    for (VectorIterator<int> vi (v); vi.More(); vi.Next())
        cout << vi.Current () << " " ;
}
```

with output:

```
0 0 0 0 0
0 0 0 0 0
```

The *main()* function illustrates two different methods of iterating through the elements of a *Vector* object *v*. The similarity between the integer variable *i* and the iterator object *vi* is self-evident. We shall revisit iterator classes when we discuss inheritance in Chapter 15.

## 13.8 A `LinkedList` class

To date we have seen the use of arrays, vectors and matrices, whose size can be either fixed at compile-time or determined at run-time. The dynamic array is clearly more versatile than a compile-time array, but it still suffers from a major flaw. If the size of an array is continually changing as a result of elements being inserted or deleted from the array, then the array mechanism of storing a list of elements contiguously in memory becomes an inefficient structure because existing elements must be repositioned. An alternative structure for storing a list of elements is to use a *linked list*, which is a more responsive structure to the insertion of new elements or deletion of existing elements. A linked list contains a set of *nodes* or *links* in which each node not only stores an object but also a link (implemented as a pointer) to the next node, allowing the nodes not to be stored contiguously in memory. If the linked list maintains pointers to the first and last nodes of the list, then the entire list of nodes can be accessed and new nodes can easily be added.

The **template class** `LinkedList` examined below is based on the non-**template** `LinkedList` **class** presented in Adams *et al.* (1995, pp. 926–60) and a linked list **class** discussed in Stroustrup (1991, pp. 265–70).

First, let us examine the **class** declaration of `LinkedList` and then examine each of `LinkedList`'s member functions more closely:

```
// ll.h
// template class LinkedList

#ifndef _LL_H // prevent multiple includes
#define _LL_H

#include <iostream.h> // C++ I/O
#include <assert.h> // assert()

#include "lli.h" // template class LinkedListIterator
#include "bool.h" // type Boolean

// swaps two objects
template <class T>
void Swap (T& a, T& b)
{
    T temp ;

    temp = a ;
    a = b ;
    b = temp ;
}

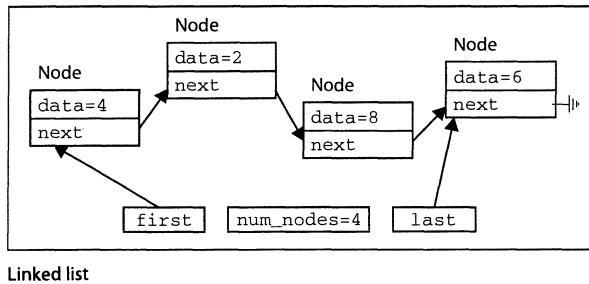
template <class T>
class LinkedList
{
private:
    int num_nodes ;
    // nested Node class
    class Node
```

```

{
public:
    T      data ;
    Node* next ;
    // constructors
    Node ()
        : data (), next (NULL) {}
    Node (const T& d, Node* ptr=NULL)
        : data (d), next (ptr) {}
    // destructor
    ~Node () { delete next ; }
} * first, * last ; // class Node
public:
    enum { START, END } ;
    // constructors
    LinkedList ()
        : num_nodes (0), first (NULL), last (NULL) {}
    LinkedList (const LinkedList& l) ;
    // destructor
    ~LinkedList ()
        { delete first ; num_nodes = 0 ;
          first = last = NULL ; }
    // member functions
    int NumberNodes () const
        { return num_nodes ; }
    Boolean Empty ()
        { return num_nodes==0 ? TRUE : FALSE ; }
    int Search (const T& obj) ;
    void Insert (const T& obj, int loc) ;
    void Head (const T& obj) ;
    void Append (const T& obj) ;
    void Delete (int loc) ;
    void Sort () ;
    // overloaded operators
    LinkedList& operator = (const LinkedList& l) ;
    T&         operator [] (int index) ;
    Boolean     operator == (const LinkedList& l) ;
    Boolean     operator != (const LinkedList& l) ;
    // friends
    friend ostream& operator << (ostream& s,
                                    const LinkedList<T>& l) ;
    friend class LinkedListIterator<T> ;
}; // template class LinkedList
//...
// member function definitions
//...
#endif // _LL_H

```

The **class** declaration of `LinkedList` encapsulates two pointer data members, `first` and `last`, of the nested **class** `Node`, which respectively point to the first and last nodes in a linked list, and an **int** data member called `num_nodes`, which represents the total number

**Fig. 13.1** A linked list consisting of four Node objects.

of nodes in a linked list; an example of a linked list is shown in Fig. 13.1<sup>3</sup>. The **class** **Node** is a nested **class** because it is an implementation detail specific to **class** **LinkedList**. **Node** encapsulates a **data** data member of **class** or type **T** which represents the object held by a given node. The **class** **Node** is in fact a recursive **class** declaration, since it contains a **next** data member of **class** **Node** which points to a node's next node in a linked list. An object of **class** **Node** can be defined in one of two ways. The **Node** default constructor allows an object to be defined in which **data** is initialised using the default constructor of **T** and **next** points to the **NULL** pointer. Alternatively, the one-argument constructor of **Node** allows **data** to be initialised to a given object and the **next** pointer set to a given memory address. **Node**'s destructor deallocates the memory associated with the **next** pointer.

The copy constructor of **LinkedList** first copies the number of nodes of the list being copied. Two pointers to **Node** are then defined for traversing the list being copied and the new list. If the number of nodes being copied is zero then an empty list is simply generated. Otherwise, each node is copied by using the two-argument constructor of **class** **Node** and the **new** operator:

```

template <class T>
LinkedList<T>::LinkedList (const LinkedList& l)
{
    num_nodes = l.num_nodes ;

    Node* old_node (l.first) ; // old list
    Node* new_node (NULL) ; // new list

    if (num_nodes == 0) // if empty list
        first = NULL ;
    else // copy first node
    {
        new_node = new Node (old_node->data) ; // copy old node
        assert (new_node != NULL) ;
        old_node = old_node->next ;
        first = new_node ;
    }
    // copy remaining nodes
    for (int i=1; i<num_nodes; i++, old_node=old_node->next,
  
```

<sup>3</sup> This type of linked list is frequently referred to as a *double-ended* linked list, since both **first** and **last** pointers are maintained, as opposed to a *single-ended* linked list, which maintains a single **first** pointer.

---

```

        new_node=new_node->next)
{
// copy old node
new_node->next = new Node (old_node->data) ;
assert (new_node->next != NULL) ;
}
last = new_node ;
}
```

The *Search()* member function sequentially searches the nodes of a linked list from the start of the list for a supplied object. If the object is found in the list then its index is returned, else a value of -1 is returned, indicating failure:

```

template <class T>
int LinkedList<T>::Search (const T& obj)
{
Node* ptr (first) ;

for (int i=0; i<num_nodes; i++, ptr=ptr->next)
    if (ptr->data == obj)
        return i ;
return -1 ;
}
```

An object can be inserted at the start or head of a list, the end or tail of a list or at any intermediate position. These three cases are supported by the *Insert()* member function. It is a fairly straightforward process to insert an object either at the start or the end of a list, whereas inserting an object at an intermediate position first requires the previous node to be located from the point of insertion. Then it is a simple matter of exchanging next pointers between the previous node and the insertion node:

```

template <class T>
void LinkedList<T>::Insert (const T& obj, int loc)
{
assert (loc >= 0 || loc < num_nodes) ;

Node* ptr = new Node (obj) ; // pointer to inserted node
assert (ptr != NULL) ;

if (loc == LinkedList<T>::START) // insert at start of list
{
ptr->next = first ;
first = ptr ;
if (num_nodes == 0) // empty list
    last = ptr ;
}
else if (loc == LinkedList<T>::END) // insert at end of list
{
if (num_nodes == 0)
    first = ptr ;
else
```

```

        last->next = ptr ;
        last = ptr ;
    }
else // insert at specified location
{
    Node* prev_ptr (first) ;

    // locate previous node in list
    for (int j=0; j<loc-1; j++)
        prev_ptr = prev_ptr->next ;

    ptr->next = prev_ptr->next ;
    prev_ptr->next = ptr ;
}
num_nodes++ ; // adjust no. of nodes
}

```

With the general *Insert()* function defined, two functions, *Head()* and *Append()*, can be defined which place an object at the start and end of a list respectively:

```

template <class T>
void LinkedList<T>::Head (const T& obj)
{
    LinkedList::Insert (obj, LinkedList<T>::START) ;
}

template <class T>
void LinkedList<T>::Append (const T& obj)
{
    LinkedList::Insert (obj, LinkedList<T>::END) ;
}

```

The *Delete()* member function removes a node from a linked list at a specified position. If the node to be removed is at the start of a list then the *first* pointer points to the second node in the list. If the node to be removed is a node other than the first in a list then it is first necessary to locate the node's previous node. Then it is a simple matter of exchanging next pointers between the previous node and the node to be removed from the list:

```

template <class T>
void LinkedList<T>::Delete (int loc)
{
    assert (loc >= 0 || loc < num_nodes) ;

    Node* ptr (NULL) ; // pointer to deleted node

    if (loc == LinkedList<T>::START) // delete first node in
                                    // list
    {
        ptr = first ;
        first = first->next ;
        ptr->next = NULL ;
    }
    else // locate previous node in list
    {
        Node* prev_ptr (first) ;
        for (int j=0; j<loc-1; j++)
            prev_ptr = prev_ptr->next ;
        prev_ptr->next = prev_ptr->next->next ;
    }
}

```

```

    }
else // delete at specified location
{
    Node* prev_ptr (first) ;

    // locate previous node in list
    for (int j=1; j<loc; j++)
        prev_ptr = prev_ptr->next ;

    ptr = prev_ptr->next ;
    prev_ptr->next = ptr->next ;
    ptr->next = NULL ;

    if (loc == num_nodes-1) // last node in list
        last = prev_ptr ;
    }
    delete ptr ;
    num_nodes-- ; // adjust no. of nodes
}

```

The *Sort()* member function of **template class** `LinkedList` sorts the elements of a linked list by value using the bubblesort algorithm:

```

template <class T>
void LinkedList<T>::Sort ()
{
    for (int i=0; i<num_nodes-1; i++)
        for (int j=num_nodes-1; i<j; j--)
            if ((*this)[j] < (*this)[j-1])
                ::Swap ((*this)[j], (*this)[j-1]) ;
}

```

The form of the overloaded assignment operator is similar to that of the copy constructor:

```

template <class T>
LinkedList<T>& LinkedList<T>::operator = (const LinkedList& l)
{
    Node* ptr (first) ;

    // delete existing nodes
    for (int i=0; i<num_nodes; i++, first=first->next, ptr=first)
    {
        delete ptr ;
    }

    num_nodes = l.num_nodes ;

    Node* old_node (l.first) ; // old list
    Node* new_node (NULL) ; // new list

    if (num_nodes == 0) // if empty list

```

```

        first = NULL ;
else // copy first node
{
    new_node = new Node (old_node->data) ; // copy old node
    assert (new_node != NULL) ;
    old_node = old_node->next ;
    first = new_node ;
}
// copy remaining nodes
for (int j=1; j<num_nodes; j++, old_node=old_node->next,
      new_node = new_node->next)
{
    // copy old node
    new_node->next = new Node (old_node->data) ;
    assert (new_node->next != NULL) ;
}
last = new_node ;
return *this ;
}

```

The overloaded random access subscript operator is defined as:

```

template <class T>
T& LinkedList<T>::operator [] (int index)
{
    assert (index >= 0 || index < num_nodes) ;

    Node* loc (first) ;

    for (int i=0; i<index; i++)
        loc = loc->next ;
    return loc->data ;
}

```

The overloaded equality relational operator performs a node-wise comparison of two linked lists:

```

template <class T>
Boolean LinkedList<T>::operator == (const LinkedList& l)
{
    if (num_nodes != l.num_nodes)
        return FALSE ;

    Node* ptr (first) ; // initialise pointers to 2 lists
    Node* l_ptr (l.first) ;

    for (int i=0; i<num_nodes; i++, ptr=ptr->next,
          l_ptr=l_ptr->next)
        if (ptr->data != l_ptr->data)
            return FALSE ;
    return TRUE ;
}

```

The overloaded inequality relational operator is implemented as a logical negation of the equality operator:

```
template <class T>
Boolean LinkedList<T>::operator != (const LinkedList& l)
{
    if (!(*this == l)) // logical negation of ==
        return TRUE ;
    return FALSE ;
}
```

The overloaded insertion operator displays the object held by each node in a linked list:

```
template <class T>
ostream& operator << (ostream& s, const LinkedList<T>& l)
{
    // point to first node in list
    LinkedList<T>::Node* ptr (l.first) ;
    for (int i=0; i<l.num_nodes; i++, ptr=ptr->next)
        s << ptr->data << " " ;
    return s ;
}
```

A program which tests `LinkedList` is:

```
// ll_tst.cpp
// tests the LinkedList template class

#include <iostream.h> // C++ I/O

#include "ll.h"           // template class LinkedList
#include "lli.h"          // template class LinkedListIterator

void main ()
{
    LinkedList<int> li0 ;
    cout << "li0: " << li0 << endl ;

    // append, head, insert:
    li0.Append (2) ;
    cout << "li0: " << li0 << ", num_nodes: "
        << li0.NumberNodes () << endl ;
    li0.Head (4) ;
    cout << "li0: " << li0 << ", num_nodes: "
        << li0.NumberNodes () << endl ;
    li0.Insert (8, 2) ;
    cout << "li0: " << li0 << ", num_nodes: "
        << li0.NumberNodes () << endl ;

    // assignment
    LinkedList<int> li1 ;
```

```

li1 = li0 ;
// copy constructor
LinkedList<int> li2 (li1) ;

cout << "li2: " << li2 << endl ;
cout << "li1: " << li1 << endl ;
cout << "li0: " << li0 << endl ;

// delete:
li2.Delete (3) ;
cout << "li2.Delete(3): " << li2 << endl ;
// search:
cout << "li1.Search(8): " << li1.Search (8) << endl ;

// composed type:
LinkedList< LinkedList<int> > lli ;
lli.Append (li0) ;
lli.Append (li1) ;
lli.Append (li2) ;

cout << "lli: " << lli << endl ;

LinkedList<int> ll_s ;
ll_s.Append (5) ; ll_s.Append (9) ; ll_s.Append (1) ;
ll_s.Append (3) ; ll_s.Append (8) ; ll_s.Append (2) ;
ll_s.Append (7) ; ll_s.Append (0) ; ll_s.Append (4) ;
ll_s.Append (6) ;

ll_s.Sort () ;
cout << "sorted ll_s: " << ll_s << endl ;

// LinkedListIterator:

cout << "LinkedListIterator i (li1): " ;
for (LinkedListIterator<int> i(li1); i.More(); i.Next())
    cout << i.Current () << " " ;
}

```

with output:

```

li0:
li0: 2, num_nodes: 1
li0: 4 2, num_nodes: 2
li0: 4 2 8, num_nodes: 3
li2: 4 2 8
li1: 4 2 8
li0: 4 2 8
li2.Delete(3): 4 2
li1.Search(8): 2
lli: 4 2 8 4 2 8 4 2
sorted ll_s: 0 1 2 3 4 5 6 7 8 9

```

---

```
LinkedListIterator i (lli): 4 2 8
```

`LinkedListIterator` objects are discussed in the next section.

You may have noticed from the above function definitions that a characteristic feature of linked lists is the sequential access of the nodes in a list. A linked list object only holds pointers to the first and last nodes of the list and the total number of nodes in the list. Thus, if a node other than the first or last is to be accessed we must always start at the first node in the list and then sequentially move through the remaining nodes in the list, via the next pointer of each node, until we arrive at the appropriate node. This sequential accessing of nodes is the key disadvantage of linked lists, and if a given operation requires a high degree of accessing of interior nodes an array will generally be a more efficient mechanism for storing a list of objects.

There are several variations on the original linked list theme. The nodes of a linked list can be ordered. A linked list can be circular, in which the last node points to the first node of the list. Large linked lists can be broken down into an array of several smaller linked lists. Instead of an array of linked lists, a composed type can be used in which the nodes of a linked list are other linked lists. Another popular form of linked list is the *doubly linked* list, in which each node contains a pointer to its previous node as well as its next node. A new form of linked list to emerge recently is the *skip list*, which is a linked list with additional pointers that skip over intermediate nodes (Pugh, 1989).

### 13.8.1 A `LinkedListIterator` class

The previous section examined the implementation of a `LinkedList` **class**. Let's now briefly examine a `LinkedListIterator` **class**, which provides access to the elements held by a `LinkedList` object. The general outline of the `LinkedListIterator` **class** is similar to that of the `VectorIterator` iterator **class** discussed previously in that a `LinkedListIterator` object encapsulates two pointer data members, `c_node` and `c_list`. The `c_node` data member acts as a current position pointer to a given element of a `LinkedList` object pointed to by the `c_list` data member. The `c_node` and `c_list` data members are initialised when an iterator object is defined, thus associating a given `LinkedList` object with a `LinkedListIterator` object. Like `VectorIterator`, `LinkedListIterator` also supports the four member functions `More()`, `Current()`, `Next()` and `ReStart()` for determining whether or not more nodes are in a list, returning the current node, advancing to the next node and repositioning the current position to the start of a list, respectively. In addition, `LinkedListIterator` overloads the prefix and postfix increment operators to allow a more natural manipulation of iterator objects:

```
// lli.h
// template class LinkedListIterator

#ifndef _LLI_H // prevent multiple includes
#define _LLI_H

#include "ll.h" // template class LinkedList

template <class T>
class LinkedListIterator
{
private:
    LinkedList<T>::Node* c_node ; // current node
    const LinkedList<T>* c_list ; // current list
```

```

public:
    // constructor
    LinkedListIterator (const LinkedList<T>& l)
        : c_node (l.first), c_list (&l) {}

    // member functions
    Boolean More ()      const
    { return c_node != NULL ? TRUE : FALSE ; }
    T      Current () const { return c_node->data ; }
    void   Next ()     { c_node = c_node->next ; }
    void   ReStart ()  { c_node = c_list->first ; }

    // overloaded operators
    Boolean operator ++ ()      // prefix
    { c_node=c_node->next ; return More() ; }
    Boolean operator ++ (int)    // postfix
    { c_node=c_node->next ; return More() ; }
};

// template class LinkedListIterator

#ifndef _LLI_H

```

A typical use of a `LinkedListIterator` **class** object was demonstrated in the `main()` function of `LL_TMP.CPP`, listed in the previous subsection.

## 13.9 A Polygon **class**

As a further illustration of the `LinkedList` **template class**, let us examine a `Polygon` **class** and associated member functions which make good use of linked lists<sup>4</sup>.

### 13.9.1 A **class** Representation of a Polygon

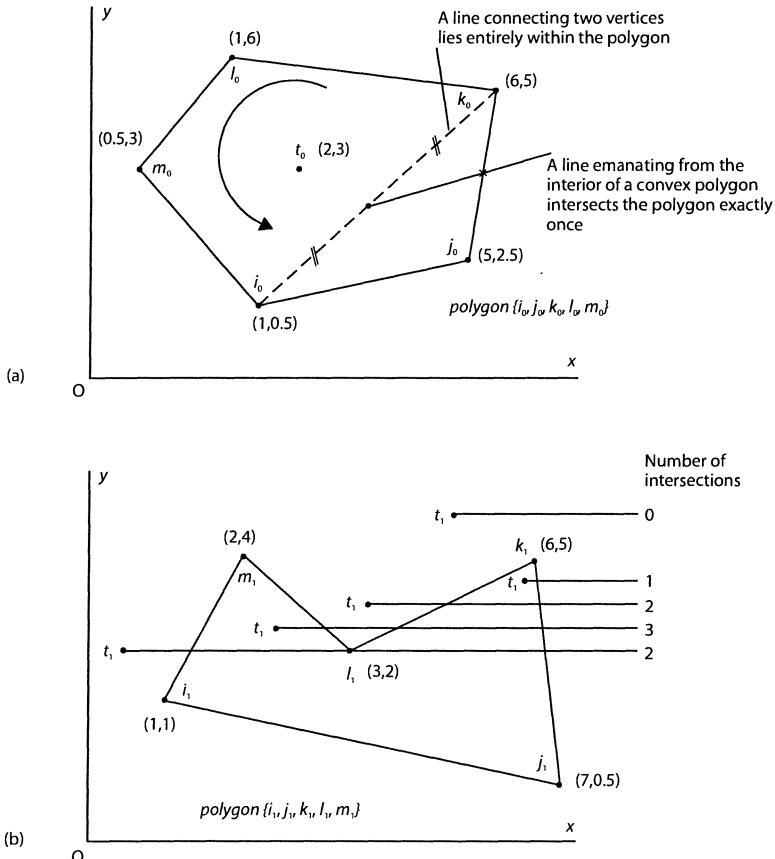
A polygon is a region consisting of a number (minimum three, i.e. a triangle) of vertices and edges which lie on a planar or curved surface. The edges of a polygon can also be either straight or curved. It is possible to model polygons as a collection of vertices, edges or surfaces, although in the present instance we shall restrict ourselves to defining a *closed* polygon purely in terms of a set of vertices existing entirely in a two-dimensional *x*-*y* plane (*z*=0). Further, it will be assumed that the edges joining polygon vertices are straight line segments.

In general, polygons are categorised as either *convex* or *concave*; see Fig. 13.2. A convex polygon has the property that a straight line segment joining any two vertices lies entirely within the polygon. Alternatively, a convex polygon can be viewed as a polygon in which all vertices are on the left-hand side of any polygon edge when traversing the polygon in an anticlockwise manner. Apart from simply stating that a concave polygon is one that is not convex, a concave polygon has the property that one or more polygon vertices lie on the right-hand side of the current polygon edge when traversing the polygon edges in an anticlockwise manner.

The **class** declaration of `Polygon` can be found in `POLY.H` and is of the form:

```
// poly.h
```

<sup>4</sup> Thanks to Tony Broome for valuable discussions while implementing the `Polygon` **class**.



**Fig. 13.2** (a) Convex and (b) concave polygons  $\{i_0, j_0, k_0, l_0, m_0\}$  and  $\{i_1, j_1, k_1, l_1, m_1\}$

```
// class Polygon

#ifndef _POLY_H // prevent multiple includes
#define _POLY_H

#include <limits.h> // INT_MAX

#include "pt&line.h" // classes Point and Line
#include "ll.h" // template class LinkedList

const double PI = 4.0 * atan (1.0) ;

// class Polygon
class Polygon
{
private:
    LinkedList<Point> verts ; // vertices
    // nested class SortPoint
    class SortPoint
```

```

{
//...
}; // nested class SortPoint
public:
enum Order { SORTED, UNSORTED_CONVEX, UNSORTED } ;
// constructor
Polygon (const LinkedList<Point>& p_list,
          Order order=SORTED) ;
// member functions
int NumberVertices () const
{ return verts.NumberNodes () ; }
Boolean PointInConvexPolygon (const Point& p) ;
Boolean PointInPolygon (const Point& p) ;
Polygon ThetaRSorted () ;
Polygon ConvexHull () ;
// friend
friend ostream& operator << (ostream& s,
                                 const Polygon& p) ;
}; // class Polygon

#endif // _POLY_H

```

with the implementation file given in POLY.CPP. Polygon encapsulates a **private** data member called *verts*, which is an object of **class** *LinkedList<Point>* and thus represents the vertices of either a convex or concave polygon as a linked list of *Point* objects. The **class** declarations of both *Point* and *Line* can be found in PT&LINE.H:

```

// pt&line.h
// classes Point and Line

#ifndef _PT&LINE_H // prevent multiple includes
#define _PT&LINE_H

#include <iostream.h> // C++ I/O
#include <math.h> // fabs()

#include "bool.h" // type Boolean

const double TOLERANCE = 1e-06 ;

// class Point
class Point
{
private:
    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}

```

```

// copy constructor
Point (const Point& p)
    : x (p.x), y (p.y), z (p.z) {}
// member functions
double& X () { return x ; }
double& Y () { return y ; }
double& Z () { return z ; }
const double& X () const { return x ; }
const double& Y () const { return y ; }
const double& Z () const { return z ; }
int CAC (const Point& p1, const Point& p2) ;
// overloaded relational operators
Boolean operator == (const Point& p) ;
Boolean operator != (const Point& p) ;
Boolean operator < (const Point& p) ;
// friend
friend ostream& operator << (ostream& s,
                                    const Point& p) ;
}; // class Point

// class Line
class Line
{
private:
    Point p0, p1 ;
public:
    // constructors
    Line ()
        : p0 (), p1 () {}
    Line (const Point& p, const Point& q)
        : p0 (p), p1 (q) {}
    // copy constructor
    Line (const Line& l)
        : p0 (l.p0), p1 (l.p1) {}
    // member functions
    Point& P0 () { return p0 ; }
    Point& P1 () { return p1 ; }
    const Point& P0 () const { return p0 ; }
    const Point& P1 () const { return p1 ; }
    Boolean Intersection (const Line& l) ;
    // friend
    friend ostream& operator << (ostream& s, const Line& l) ;
}; // class Line

#endif // _PT&LINE_H

```

with the **class** member function definitions given in PT&LINE.CPP. The Point and Line classes presented above are based on those presented in LI\_CLASS.CPP of Chapter 9.

The nested **class** SortPoint, constructor of **class** Polygon and member functions will be discussed in more detail later, but it is sufficient to say at this point that the default behaviour of Polygon's constructor assumes that the list of vertices is in order as we traverse

the `Polygon` boundary. For instance, the linked list of points for the convex polygon of Fig. 13.2(a) supplied to the `Polygon` constructor could be in the order  $i_0, j_0, k_0, l_0, m_0$  or  $i_0, m_0, l_0, k_0, j_0$  if we assume that vertex  $i_0$  is the *start* or *anchor* vertex. Clearly, the ordering of the vertices is important to ensure that the polygon edge connectivity is correct.

### 13.9.2 A Point Inside or Outside a Polygon

In determining whether a point lies inside or outside a polygon it is instructive first to consider the simpler case of when the polygon is convex. When a polygon is convex, as in Fig. 13.2(a), then in developing a member function which informs us whether or not a test point is inside or outside a `Polygon` object we can take advantage of the property that for a point to be inside a polygon the point must lie on the left-hand side of all polygon edges as we traverse the polygon in an anticlockwise manner. The advantage of defining convexity in this manner is that it lends itself to the `CAC()` and `Point::CAC()` functions defined in Chapters 6 and 9. The `Point::CAC()` member function declarator is:

```
int CAC (const Point& p1, const Point& p2) ;
```

and returns either  $-1$ ,  $1$  or  $0$ , depending on whether three `Point` objects are clockwise, anticlockwise or collinear. For example, if we traverse the polygon shown in Fig. 13.2(a) in an anticlockwise manner, using vertex  $i_0$  as the start vertex, then test point  $t_0$  is inside the polygon if the sets of points  $(t_0, i_0, j_0), (t_0, j_0, k_0), \dots$  all form an anticlockwise rotation. The following `Point::PointInConvexPolygon()` member function uses this method to test whether a given point is inside or outside a `Polygon` object:

```
Boolean Polygon::PointInConvexPolygon (const Point& p)
{
    // direction of test point and first two vertices
    int dir = p.CAC (verts[0], verts[1]) ;

    // remaining vertices
    for (int i=1; i<verts.NumberNodes(); i++)
    {
        if (i == verts.NumberNodes()-1) // if last vertex
        {
            if (dir != p.CAC (verts[i], verts[0]))
                return FALSE ;
        }
        else
        {
            if (dir != p.CAC (verts[i], verts[i+1]))
                return FALSE ;
        }
    }
    return TRUE ;
}
```

and returns logical-false if a point is not inside a `Polygon` object, else logical-true. The definition of `PointInConvexPolygon()` begins by determining the direction of rotation of the test point and the first two `Polygon` vertices in the `verts` linked list. The direction of rotation is first determined because the vertices of a `Polygon` object could have been defined

in either a clockwise or an anticlockwise manner. *PointInConvexPolygon()* then proceeds to visit the remaining vertices of a *Polygon* object, being careful to ensure that the last polygon edge consists of the last vertex in the *verts* linked list and the start vertex.

Rather than characterising a convex polygon in terms of the positioning of polygon vertices when traversing the boundary of a polygon in a given direction, we observe that a straight line emanating (in any direction) from a point in the interior of a convex polygon to a point outside a convex polygon intersects the boundary of the polygon exactly once; see Fig. 13.2(a). This property of convex polygons is a consequence of the so-called *Jordan curve theorem*<sup>5</sup>, which informs us whether an arbitrary point is inside or outside either a convex or concave polygon based on the number of intersections of a straight line, emanating from the test point, with the polygon edges. If the number of intersections is odd the point is inside the polygon; else if the number of intersections is even the point is outside the polygon. Fig. 13.2(b), based on Bowyer and Woodwark (1993, Fig. 2(iii)), illustrates horizontal test lines from various points inside and outside a concave polygon. Pay particular attention to the case in which the test line from a point outside the concave polygon intersects a polygon vertex. If we ignore intersections between the test line and a polygon vertex then the number-of-intersections rule still applies. The *Polygon::PointInPolygon()* member function determines whether a point is inside or outside either a convex or concave *Polygon* object using the number-of-intersections rule and is based on the algorithm presented in Sedgewick (1988, pp. 353–5):

```
Boolean Polygon::PointInPolygon (const Point& p)
{
    int count (0) ;

    // construct horizontal line from test point
    Line test_line (p, Point (INT_MAX, p.Y()), 1 ;

    for (int i=0; i<verts.NumberNodes(); i++)
    {
        // test if polygon vertex on test line
        l.P0 () = verts[i] ; l.P1 () = verts[i] ;
        if (!test_line.Intersection (l))
        {
            if (i == verts.NumberNodes()-1) // if last vertex
            {
                l.P0 () = verts[i] ; l.P1 () = verts[0] ;
            }
            else
            {
                l.P0 () = verts[i] ; l.P1 () = verts[i+1] ;
            }
            if (test_line.Intersection (l))
                count++ ;
        }
    }
    return (count % 2) == 1 ? TRUE : FALSE ;
}
```

---

<sup>5</sup> Refer to, for example, Preparata and Shamos (1985, Section 3.3).

`PointInPolygon()` begins by defining a `count` variable to record the number of intersections between a test line, `test_line`, and each edge, `l`, of the `Polygon` object that it operates on. The test line is a line segment which extends horizontally from the test point, `p`, to a point which has a `x`-axis component of `INT_MAX` (=32 767) to ensure that the test line extends outside the `Polygon` object it is tested against. `INT_MAX` is defined in the C++ library header file `LIMITS.H`. `PointInPolygon()` then proceeds to test for a line intersection between each polygon edge formed by adjacent polygon vertices and the test line, ensuring that the last polygon edge is dealt with correctly. As was discussed above, if a polygon vertex falls on the test line then it is not counted as an intersection. If a line intersection occurs then the `count` variable is incremented and, depending on whether `count` is odd or even after the test line is tested for intersection with all polygon edges, `PointInPolygon()` returns a value of logical-true or logical-false respectively.

### 13.9.3 Convex and Concave Polygons, Sorting Vertices and Convex Hulls

Previous subsections considered the cases when all points from the original point set were sorted and formed vertices of either a convex or concave polygon. This does not have to be the case. The constructor of **class** `Polygon` allows the vertices (in the form of a linked list) of a polygon to be defined in one of three ways: (1) as a sorted list of vertices describing either a convex or concave polygon; (2) as an unsorted list of vertices describing a convex polygon; or (3) as a completely unsorted list of vertices, the *convex hull* of which describes a convex polygon:

```
Polygon::Polygon (const LinkedList<Point>& p_list, Order  
order)  
{  
    // test type of order  
    assert (order == Polygon::SORTED ||  
            order == Polygon::UNSORTED_CONVEX ||  
            order == Polygon::UNSORTED) ;  
  
    // sorted convex or concave polygon  
    if (order == Polygon::SORTED)  
        verts = p_list ;  
    // unsorted convex polygon  
    else if (order == Polygon::UNSORTED_CONVEX)  
    {  
        verts = p_list ;  
        verts = ThetaRSorted ().verts ;  
    }  
    // unsorted convex or concave polygon  
    else  
    {  
        verts = p_list ;  
        verts = ConvexHull ().verts ;  
    }  
}
```

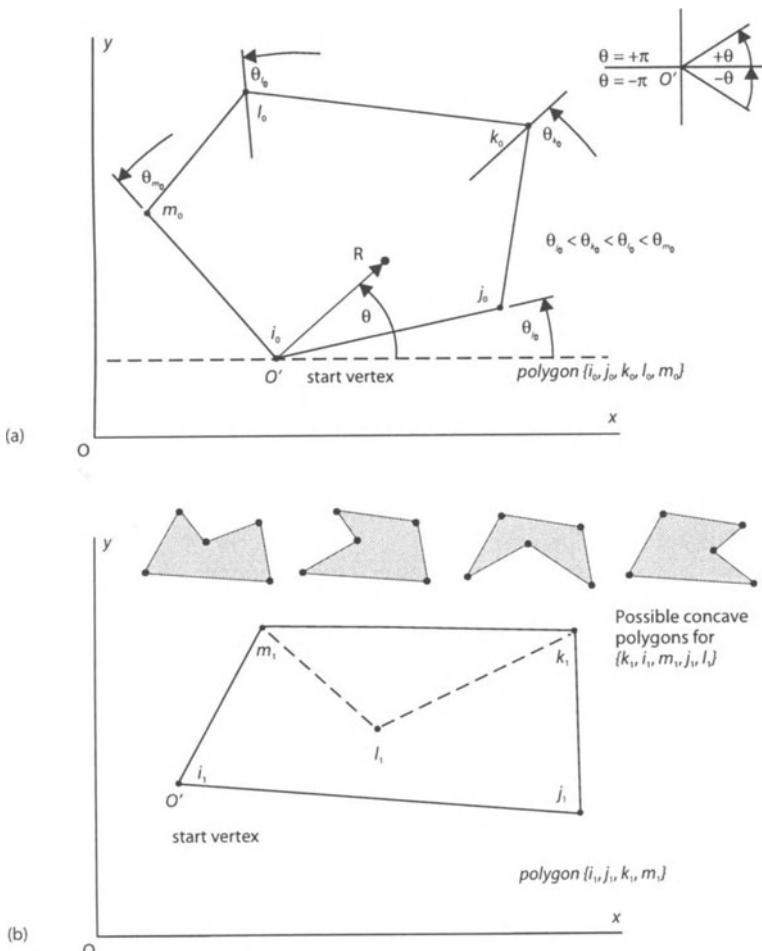
The type of ordering of the vertices is characterised by an enumerated data type called `Order`, which is **publicly** declared in **class** `Polygon`. The constructor argument `order` indicates the type of ordering, is set to `SORTED` by default and refers to a sorted list of vertices for either

a convex or concave polygon. When the list of vertices is sorted it is a simple matter of assigning the linked list of vertices, `p_list`, to the `Polygon::verts` data member. If the list of vertices is unsorted but known to form a convex polygon, then again the linked list of vertices, `p_list`, is assigned to `verts`, but the `Polygon::ThetaRSorted()` member function is subsequently called. `ThetaRSorted()` sorts the vertices of a polygon according to angle and radial distance from the polygon start vertex, but before we describe the exact details of `ThetaRSorted()`, let us examine how the polygon vertices are sorted and the need for the `Polygon::SortPoint` nested **class**.

Consider sorting the unsorted set of vertices  $\{k_0, i_0, m_0, j_0, l_0\}$  for the convex polygon of Fig. 13.3(a). One method of sorting the vertices is to sort the vertices relative to an additional point known to be internal to the set of vertices. However, in practice it suffices to sort the vertices relative to the start point, which is chosen as the leftmost ordinate (i.e. minimum  $y$ -minimum  $x$ ) vertex of the original set of vertices. With a start vertex established the remaining polygon vertices are sorted according to polar angle and radial distance from the start vertex. The radial distance comparison is only required if two vertices have the same polar angle. The polar angle,  $\theta$  ( $-\pi \leq \theta \leq +\pi$ ), is measured from the positive horizontal axis passing through the start vertex. Note the necessity of sorting the vertices relative to the start vertex (or local origin O') rather than the global origin O.

It would have been straightforward to develop a function which is passed a linked list of unsorted points and returns a sorted linked list of points in accordance with the polar angle-radial distance sort discussed above. However, since the polar angle-radial distance sorting of points is unique to a polygon, it makes sense to declare a `SortPoint` **privately** nested **class** within `Polygon` whose sole purpose is to perform a polar angle-radial distance sort of polygon vertices. The declaration of `Polygon::SortPoint` is:

```
class Polygon
{
private:
    LinkedList<Point> verts ; // vertices
    // nested class SortPoint
    class SortPoint
    {
private:
    double x, y, z ;
public:
    SortPoint ()
        : x (0.0), y (0.0), z (0.0) {}
    SortPoint (double x_arg, double y_arg,
               double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    SortPoint (const Point& p)
        : x (p.X()), y (p.Y()), z (p.Z()) {}
    double& X () { return x ; }
    double& Y () { return y ; }
    double& Z () { return z ; }
    double R () { return sqrt(x*x+y*y+z*z) ; }
    double Theta () ;
    Boolean operator < (const SortPoint& sp) ;
    Boolean operator > (const SortPoint& sp) ;
    Boolean operator == (const SortPoint& sp) ;
    Boolean operator != (const SortPoint& sp) ;
```



**Fig. 13.3** (a) Convex polygon generated from an initial unsorted set of vertices  $\{k_0, i_0, m_0, j_0, l_0\}$  to the sorted set of vertices  $\{i_0, j_0, k_0, l_0, m_0\}$ . (b) Convex polygon generated from an initial unsorted set of vertices  $\{k_1, i_1, m_1, j_1, l_1\}$  to the convex hull set of vertices  $\{i_1, j_1, k_1, m_1\}$ . Also shown are the four concave polygons which could have been formed from the set of vertices  $\{k_1, i_1, m_1, j_1, l_1\}$ .

```

    } ; // nested class SortPoint
public:
    //...
}

```

The declaration of **class** `SortPoint` is similar to that of **class** `Point` in that it encapsulates `x`, `y` and `z` data members, but in addition `SortPoint` defines `R()`, `Theta()` and alternative overloaded relational operator member functions to those of `Point`. The `R()` member function returns the radial distance of a `SortPoint` object from the origin, whereas `Theta()` returns the polar angle of a `SortPoint` object and is defined as:

```

double Polygon::SortPoint::Theta ()
{
    double theta (0.0) ;

```

---

```

if (fabs(x) < TOLERANCE && y > 0.0)           // upper vertical
    theta = 90.0 ;
else if (fabs(x) < TOLERANCE && y < 0.0) // lower vertical
    theta = -90.0 ;
else if (fabs(y) < TOLERANCE && x > 0.0) // right horizontal
    theta = 0.0 ;
else if (fabs(y) < TOLERANCE && x < 0.0) // left horizontal
    theta = 180.0 ;
else // non-vertical/non-horizontal
{
    if (x > 0.0 && y > 0.0)
        theta = atan (fabs(y)/fabs(x))*180/PI ;
    else if (x < 0.0 && y > 0.0)
        theta = 90.0 + atan (fabs(x)/fabs(y))*180/PI ;
    else if (x < 0.0 && y < 0.0)
        theta = -90.0 - atan (fabs(x)/fabs(y))*180/PI ;
    else // dx>0.0 && dy<0.0
        theta = - atan (fabs(y)/fabs(x))*180/PI ;
}
return theta ;
}

```

Member function *Theta()* first tests whether the *SortPoint* object lies in a horizontal or vertical plane which passes through the start vertex. *Theta()* then proceeds to determine which quadrant the *SortPoint* falls in relative to the start vertex; see Fig. 13.3(a).

The overloaded relational (<, >, ==, !=) operator member functions of **class SortPoint** are defined as:

```

Boolean Polygon::SortPoint::operator < (const SortPoint& sp)
{
    if (Theta() < sp.Theta())
        return TRUE ;
    else if ((fabs(Theta()-sp.Theta())<TOLERANCE) &&
              (R() < sp.R()))
        return TRUE ;
    else
        return FALSE ;
}

Boolean Polygon::SortPoint::operator > (const SortPoint& sp)
{
    if (Theta() > sp.Theta())
        return TRUE ;
    else if ((fabs(Theta()-sp.Theta())<TOLERANCE) &&
              (R() > sp.R()))
        return TRUE ;
    else
        return FALSE ;
}

Boolean Polygon::SortPoint::operator == (const SortPoint& sp)

```

```

{
    return (fabs(x-sp.x)<TOLERANCE && fabs(y-sp.y)<TOLERANCE &&
            fabs(z-sp.z)<TOLERANCE) ? TRUE : FALSE ;
}

Boolean Polygon::SortPoint::operator != (const SortPoint& sp)
{
    if (!(*this == sp)) // logical negation of ==
        return TRUE ;
    return FALSE ;
}

```

The overloaded < and > operators compare two SortPoint objects with respect to the radial distance and angle of the objects from the origin, with priority given to the angular comparison. The equality and inequality operators == and != compare two SortPoint objects simply by comparing the x, y and z data members of each object.

Note that the declaration of **class** SortPoint would have been greatly simplified if the C++ inheritance mechanism (refer to Chapter 15) had been used and SortPoint derived from Point.

Let's now return to the examination of the Polygon::ThetaRSorted() member function, which sorts the vertices of a polygon according to the angle and radial distance from the polygon start vertex:

```

Polygon Polygon::ThetaRSorted ()
{
    // sorted list of Polygon vertices
    LinkedList<Point> sorted_poly ;

    // find pt. with min y and min x (i.e. leftmost ordinate pt.)
    Point min_pt (verts[0]) ;
    int min_pt_index (0) ;
    for (int i=1; i<verts.NumberNodes(); i++)
        if ((verts[i].Y() < min_pt.Y() &&
              verts[i].X() <= min_pt.X()) ||
             (verts[i].Y() <= min_pt.Y() &&
              verts[i].X() < min_pt.X()))
    {
        min_pt = verts[i] ;
        min_pt_index = i ;
    }

    // use min_pt as 'local' origin point and
    // first point in list of sorted vertices
    sorted_poly.Append (min_pt) ;

    // translate points to 'local' point and
    // add to separate linked list excluding 'local' origin
    // point
    LinkedList<SortPoint> sp ;
    for (int j=0; j<verts.NumberNodes(); j++)
        if (j != min_pt_index)

```

```

    sp.Append (SortPoint (
        Point (verts[j].X()-min_pt.X(),
               verts[j].Y()-min_pt.Y())));
}

// sort translated points
sp.Sort () ;

// add sorted points to polygon vertices,
// remembering to translate back
for (int k=0; k<sp.NumberNodes(); k++)
    sorted_poly.Append (Point (sp[k].X()+min_pt.X(),
                               sp[k].Y()+min_pt.Y()));

return Polygon (sorted_poly) ;
}

```

The object `sorted_poly` is of **class** `LinkedList<Point>` and encapsulates a linked list of the  $\theta$ - $R$  sorted vertices. Next, the polygon start vertex (leftmost ordinate point) is found by performing a linear search through the entire list of original polygon vertices, `verts`. Also, the position of the start vertex within the `verts` linked list is recorded. The start vertex is then placed at the head of the `sorted_poly` `LinkedList<Point>` object. Next, an object `sp` of **class** `LinkedList<SortPoint>` is defined, to which all of the polygon vertices (excluding the start vertex) are appended and then sorted. Note that a `Point` vertex is first translated to the start vertex and then cast to a `SortPoint` vertex. Since object `sp` is of **class** `LinkedList<SortPoint>`, the `LinkedList<T>::Sort()` member function will call the `SortPoint::operator<()` overloaded member function. Finally, the linked list of sorted vertices is appended to `sorted_poly` and a `Polygon` object returned.

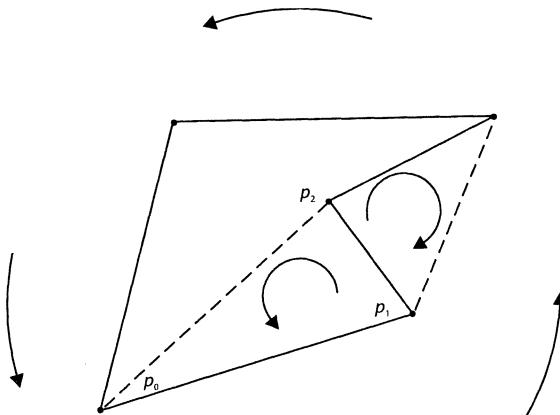
Consider now the problem of defining a `Polygon` object as the convex hull of a set of unsorted arbitrary points. Figure 13.3(b) illustrates the convex polygon  $\{i_1, j_1, k_1, m_1\}$  of the initially unsorted set of points  $\{k_1, i_1, m_1, j_1, l_1\}$ . So how do we find computationally the convex hull of a set of arbitrary points in a two-dimensional plane? At first glance the problem appears fairly straightforward, but in fact it is deceptively difficult. Several established algorithms exist, such as *gift-wrapping* or *Jarvis's march*, *Graham's scan*, and the *quickhull* or *divide and conquer* algorithm. Incidentally, an excellent treatment of convex hulls for both two and three dimensions can be found in O'Rourke (1994). The algorithm that we will examine is Graham's scan (Graham, 1972), because it is a reasonably efficient algorithm and makes good use of the previously developed `Polygon::ThetaRSorted()` and `Point::CAC()` member functions.

Suppose we are given a  $\theta$ - $R$  sorted set of points relative to a start point, as discussed above. With reference to Fig. 13.4, the Graham's scan algorithm commences at the start point by traversing the entire point set in an anticlockwise manner, examining triples of consecutive points and testing whether each triple of points forms an anticlockwise or clockwise rotation until we return to the start point. If points  $p_0 p_1 p_2$  form an anticlockwise rotation then  $p_1$  is a point on the convex hull, else if  $p_0 p_1 p_2$  form a clockwise rotation then  $p_1$  is not a hull point. `Polygon::ConvexHull()` implements the Graham's scan algorithm:

```

Polygon Polygon::ConvexHull ()
{
    // sort list and convex hull of polygon vertices
    LinkedList<Point> sorted_poly = this->ThetaRSorted ().verts ;

```



**Fig. 13.4** The Graham's scan algorithm for determining the convex hull of a set of points. Point  $p_1$  is not a hull point if points  $p_0, p_1$  and  $p_2$  form a clockwise rotation.

```

LinkedList<Point> hull_poly ;

// 'local' origin of sorted polygon is a
// hull point
hull_poly.Append (sorted_poly[0]) ;

// travel thro' sorted polygon vertices
for (int i=0; i<sorted_poly.NumberNodes()-1; i++)
{
    if (i == sorted_poly.NumberNodes()-2) // reached start
    {
        if (sorted_poly[i].CAC (sorted_poly[i+1],
                               sorted_poly[0]) >= 0)
            hull_poly.Append (sorted_poly[i+1]) ;
    }
    else
    {
        if (sorted_poly[i].CAC (sorted_poly[i+1],
                               sorted_poly[i+2]) >= 0)
            hull_poly.Append (sorted_poly[i+1]) ;
    }
}
return Polygon (hull_poly) ;
}

```

First, a `LinkedList<Point>` object, `sorted_poly`, is defined which encapsulates a  $\theta-R$  sorted linked list of the vertices of the `Polygon` object operated on by `ConvexHull()`. The object `hull_poly` is the convex hull of the polygon vertices. By definition, the start vertex must be a hull vertex and is therefore appended to `hull_poly` before the remaining vertices are examined in triples. As discussed above, if an anticlockwise rotation is described by a triple vertex set, then the intermediate vertex of the triple vertex set is a hull vertex and is therefore appended to the linked list of hull vertices, `hull_poly`.

Since `ConvexHull()` returns a `Polygon` object, `Polygon`'s constructor uses `ConvexHull()` to define a `Polygon` object which is the convex hull of the `Point` linked list `p_list` if the `Polygon` constructor argument `order` is set to `UNSORTED`:

```
Polygon::Polygon (const LinkedList<Point>& p_list, Order order)
{
    //...
    // sorted convex or concave polygon
    if (order == Polygon::SORTED)
        //...
    // unsorted convex or concave polygon
    else
    {
        verts = p_list ;
        verts = ConvexHull ().verts ;
    }
}
```

Why form a convex hull polygon from an unsorted list of vertices? Simply because a user could supply a whole variety of point sets defining a `Polygon` object, and the `Polygon` constructor has no way of determining exactly which concave or convex polygon the user intended. Consequently, the `Polygon` constructor constructs the convex hull of an unsorted list of points. Figure 13.3(b) illustrates the four possible combinations of concave polygons from the point set  $\{k_1, i_1, m_1, j_1, l_1\}$ .

The following program tests `class Polygon`:

```
// poly_tst.cpp
// tests class Polygon

#include <iostream.h> // C++ I/O

#include "pt&line.h" // classes Point and Line
#include "poly.h" // class Polygon

void main ()
{
    // define points of convex polygon and test point
    Point i0 (1.0, 0.5), j0 (5.0, 2.5), k0 (6.0, 5.0),
        l0 (1.0, 6.0), m0 (0.5, 3.0) ;
    Point test_point0 (2.0, 3.0) ;
    // define points of concave polygon and test point
    Point i1 (1.0, 1.0), j1 (7.0, 0.5), k1 (6.0, 5.0),
        l1 (3.0, 2.0), m1 (2.0, 4.0) ;
    Point test_point1 (3.0, 3.0) ;

    // sorted convex polygon:

    LinkedList<Point> ll_p0 ;
    ll_p0.Append (i0) ; ll_p0.Append (j0) ; ll_p0.Append (k0) ;
    ll_p0.Append (l0) ; ll_p0.Append (m0) ;
```

```
Polygon convex_poly (ll_p0) ;
cout << "sorted convex polygon:" << endl
    << convex_poly << endl ;

// test if test point is inside or outside convex polygon
if (convex_poly.PointInConvexPolygon (test_point0))
    cout << "point " << test_point0
        << " in convex polygon" << endl ;
else
    cout << "point " << test_point0
        << " outside convex polygon " << endl ;

// sorted concave polygon:

LinkedList<Point> ll_p1 ;
ll_p1.Append (i1) ; ll_p1.Append (j1) ; ll_p1.Append (k1) ;
ll_p1.Append (l1) ; ll_p1.Append (m1) ;

Polygon concave_poly (ll_p1) ;
cout << "sorted concave polygon:" << endl
    << concave_poly << endl ;

// test if test point is inside or outside
// convex polygon
if (concave_poly.PointInPolygon (test_point1))
    cout << "point " << test_point1
        << " in concave polygon" << endl ;
else
    cout << "point " << test_point1
        << " outside concave polygon " << endl ;

// unsorted convex polygon:

LinkedList<Point> ll_p2 ;
ll_p2.Append (k0) ; ll_p2.Append (i0) ; ll_p2.Append (m0) ;
ll_p2.Append (j0) ; ll_p2.Append (l0) ;

Polygon un_convex_poly (ll_p2, Polygon::UNSORTED_CONVEX) ;
cout << "unsorted convex polygon:" << endl
    << un_convex_poly << endl ;

// unsorted concave polygon:

LinkedList<Point> ll_p3 ;
ll_p3.Append (k1) ; ll_p3.Append (i1) ; ll_p3.Append (m1) ;
ll_p3.Append (j1) ; ll_p3.Append (l1) ;

Polygon un_concave_poly (ll_p3, Polygon::UNSORTED) ;
cout << "unsorted concave polygon:" << endl
    << un_concave_poly << endl ;
}
```

with output:

```

sorted convex polygon:
(1, 0.5, 0) (5, 2.5, 0) (6, 5, 0) (1, 6, 0) (0.5, 3, 0)
point (2, 3, 0) in convex polygon
sorted concave polygon:
(1, 1, 0) (7, 0.5, 0) (6, 5, 0) (3, 2, 0) (2, 4, 0)
unsorted convex polygon:
(1, 0.5, 0) (5, 2.5, 0) (6, 5, 0) (1, 6, 0) (0.5, 3, 0)
unsorted concave polygon:
(1, 1, 0) (7, 0.5, 0) (6, 5, 0) (2, 4, 0)

```

The above Point and Polygon objects correspond exactly to those shown in Figs. 13.2 and 13.3.

## 13.10 Summary

This chapter has illustrated **template** functions and classes. By defining a **template** function or **class** you define a mould for a family of related overloaded functions or classes by letting the data types or objects that the function or **class** operates on be arguments.

Template functions and classes are very powerful because they allow a programmer to design a generic function or **class** that can operate on a variety of different types and classes without requiring a separate implementation for each function or type of **class**. The use of templates is ideally suited to functions and classes that have the same logic and perform similar operations irrespective of the type or **class** of the data or objects that they operate on.

A **template** or generic function can be thought of as a function that has the ability to overload itself an infinite number of times according to the objects that it operates on. Template functions can have multiple type arguments but not non-type arguments. A **template class** can have both multiple type and non-type arguments. A **template class** can itself be used as a **template** argument, thus defining a nested **template** or a composed type.

Beware of inserting type, **class** or context dependencies into a **template** function or **class**. A developer of a **template** function or **class** generally has no knowledge of the variety of types, classes and applications to which the **template** will be applied. If the **template** function or **class** is not completely generic, compilation errors will result which can be very difficult to trace.

This chapter has examined several popular **template** data structures, such as Vector, Matrix, LinkedList and associated iterator **class** LinkedListIterator. In addition to linked list, vector and matrix data structures, other popular **template** data structures are the Stack and Queue, Trees (BinarySearchTree, AVL etc.), Polynomial, Graph and Hash (Budd, 1994; Ford and Topp, 1996). We shall examine Stack and Queue **template** classes in Chapter 15 when we discuss templates in connection with inheritance.

In conclusion, it is worth noting that the draft C++ standard adds the Standard Template Library (STL), which consists of container and iterator classes, and also adds the **dynarray** (dynamic array) and **ptrdynarray** (pointer to dynamic array) **template** classes to C++'s standard library of classes. The **dynarray** **template class** is similar in operation to the **template Vector** **class** presented in this chapter, but a lot better designed! The **ptrdynarray<T>** **template class** is derived (see Chapter 15) from **dynarray<void\*>**, and is used when working with an array of pointers to an object of type T. In Chapter 15 a **template class** called **VectorPtr** is presented which is similar in principle to **ptrdynarray**. For a detailed discussion of both **dynarray** and **ptrdynarray** refer to Plauger (1995).

## Exercises

13.1 Overload the `Max()` **template** function specifically for type `char*`:

```
template <class T>
T Max (T a, T b)
{
    return a > b ? a : b ;
}
```

13.2 For the following `Complex` **class**, define an `Abs()` **friend** function which returns the absolute value of a complex number:

```
template <class T>
class Complex
{
private:
    T re, im ;
public:
    Complex ()
        : re (0.0), im (0.0) {}
    Complex (T real, T imag)
        : re (real), im (imag) {}
};
```

13.3 Implement simple `Polygon` classes which encapsulate a polygon's `Point` vertices as a pointer to `Point`, `vector` and linked list using **templates** throughout. Consider the advantages and disadvantages of each implementation.

13.4 Change the `LinkedList` **class** presented in this chapter from a double-ended to a single-ended linked list so that `LinkedList` encapsulates only a `first` pointer.

13.5 The `VectorIterator` iterator **class** for `Vector` objects declared in `V_IT_C.H` is of the form:

```
template <class T>
class VectorIterator
{
private:
    int           c_elem ; // current element
    const Vector<T>*> vec ; // current Vector
//...
};
```

It was mentioned in the text that the `vec` data member could equally be declared a reference instead of a pointer. Modify `VectorIterator` so that `vec` is now a reference to `Vector`.

13.6 Implement a `Polynomial` **class** which models a polynomial such as:

$$p(x) = 7x^4 + 6x^2 + 4x + 1$$

To assist in the implementation of `Polynomial`, declare a **class** called `Term` which encapsulates coefficient and exponent data members. Encapsulate a linked list of `Terms` within `Polynomial`. Use templates throughout.

For further details of polynomial representations in C++ refer to Budd (1994), Wilt (1994) and Horowitz *et al.* (1995).

- 13.7 Develop an undirected graph **class**, Graph, which encapsulates lists of vertices and undirected edges. Provide a default constructor and a two-argument constructor which allows a graph object to be defined in terms of graph vertices and edges. Provide operations for adding and deleting either a vertex or an edge from the graph. Use **template** classes throughout.

For an excellent discussion of graph data structures in C++ refer to Horowitz *et al.* (1995), Weiss (1996) and Ford and Topp (1996).

# Exception Handling

*Errors are the burden of all programming. Run-time errors are particularly troublesome because of their associated difficulty in tracing. C++ supports a language-based mechanism called exception handling for handling run-time errors. When an abnormal situation arises at run-time, a program should terminate. If, however, before program termination occurs an exception or error is thrown to a section of the program which deals with exceptions, information can be gathered to assist in diagnosing the source of the problem which resulted in program termination.*

*The C++ exception handling mechanism is based on three keywords: try, catch and throw. If an exception occurs within a block of program statements that are tested for errors then the exception is thrown and caught by a program's exception handler. Functions and class member functions can have an associated exception specification which indicates a function's exception-handling capabilities.*

*C++ supports various exception handling functions and classes such as xmsg and xalloc. Alternatively, user-defined exception handling classes can be declared, and we will examine Index and Range classes, which are useful in the implementation of Vector and Matrix classes. The chapter finishes by taking a look at exception handling and the new operator. It will be shown that by default the new operator throws an xalloc exception in the case of unsuccessful memory allocations.*



## 14.1 try, catch and throw

Exception handling in C++ revolves around three keywords: **try**, **catch** and **throw**. A block of program code in which an exception may occur is prefixed by the **try** keyword:

```
try
{
    // block of statements monitored for exceptions
    //...
}
```

The block of statements within the opening and closing braces of the **try** statement is monitored for exceptions. If an exception occurs within this block, program flow is interrupted and the exception is thrown, using the **throw** keyword:

```
throw exception ; // throw point
```

and caught, using the **catch** keyword, to a part of the program which processes exceptions:

```
catch (T obj)
{
    // process exception
    //...
}
```

The scope of *obj* is local to the **catch** statement. The section of a program which deals with handling exceptions is frequently referred to as the *exception handler*.

If an exception occurs and an exception is thrown, the program searches for an associated exception handler. If a handler is found, program control is transferred to the handler. If no handler is found, the program will be terminated by calling the *terminate()* function; see later. Although several variations exist, the above discussion outlines the general procedure of exception handling in C++.

Let's see **try**, **catch** and **throw** in action:

```
// tct.cpp
// introduces the try, catch and throw keywords
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0), i ; // note: definition of i!

    cout << "enter a positive integer: " ;
    cin >> number ;

    // code may throw an exception
    try
    {
        if (number < 0)
            // throw exception to handler
            throw number ;
        else
            cout << "number entered: " << number << endl ;
    }
    // process exception
    catch (int i)
    {
        cout << "negative number entered!" << endl ;
    }
}
```

with some user interaction:

```
enter a positive integer: -7
negative number entered!
```

Note that the variable `i`, defined between the parentheses of the `catch` statement, is local to the `catch` statement block. This program prompts a user to enter a positive integer. If a positive integer is entered by the user the program executes normally. If the user enters a negative integer an exception is thrown from within the `try` statement and caught by the `catch` statement. The `try` statement block encompasses an `if-else` statement which tests whether the number entered is greater than zero or not. If the number entered is less than zero an exception is thrown by the unary `throw` expression:

```
throw number ;
```

The exception will be caught by its corresponding `catch` statement (assuming one exists). The type of the exception, in this case integer, determines which `catch` statement is used for processing the exception. The `catch` statement can only be used directly after a `try` statement or after another `catch` statement exception handler. Thus the following will generate a compilation error:

```
try
{
//...
}
int j (1) ;
//...
catch (int i)
{
//...
}
```

There can be any number of `catch` statements following a `try` statement:

```
try
{
//...
}
catch (T1 obj)
{
//...
}
catch (T2 obj)
{
//...
}

//...

catch (Tn obj)
{
//...
}
```

If an exception is thrown to which there is no corresponding **catch** statement, the program will abnormally terminate by calling the *terminate()* function. To illustrate abnormal termination, let us modify the program TCT.CPP as follows:

```
// term.cpp
// illustrates abnormal program termination
// by calling the terminate() function
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0) ;

    cout << "enter a positive integer: " ;
    cin >> number ;

    // code may throw an exception
    try
    {
        if (number < 0)
            // throw exception to handler
            throw number ;
        else
            cout << "number entered: " << number << endl ;
    }
    // process exception
    catch (double d)
    {
        cout << "negative number entered!" << endl ;
    }
}
```

with some user interaction:

```
enter a positive integer: -7
program aborted
```

If you are programming in a Windows (generic sense) environment you may alternatively see a dialog box displayed with the message 'Program Aborted'.

The key difference between programs TCT.CPP and TERM.CPP is that the **int** type in the **catch** statement has been changed to type **double**. When the program executes and a user enters a negative integer this exception will not be caught by the **catch(double d)** statement block. Each **catch** statement exception handler will only evaluate an exception that matches, or can be converted to, the data type or **class** specified in the **catch** statement argument list. If no match is found in the set of **catch** statements the *terminate()* function is called. In general, a **throw** expression of type T is a match for an exception handler of type T, **const** T, T& and **const** T&. Also, the same match sequence applies to a **throw** expression that is a pointer to a type that can be converted to the handler type by standard pointer conversion. This is illustrated in the following program:

```
// types.cpp
```

```

// illustrates different handler types
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0) ;
    //int* pi = new int (0) ;

    cout << "enter a positive integer: " ;
    cin >> number ;
    //cin >> *pi ;

    // code may throw an exception
    try
    {
        if (number < 0)
        //if (*pi < 0)
        // throw exception to handler
        throw number ;
        //throw pi ;
    else
        cout << "number entered: " << number << endl ;
        //cout << "number entered: " << *pi << endl ;
    }
    // process exception
    //catch (void* pv)
    //catch (int i)
    //catch (const int i)
    //catch (int& i)
    catch (const int& i)
    {
        cout << "negative number entered!" << endl ;
    }
}

```

Incidentally, if your program contains multiple **catch** statements that clash a compilation error will be generated:

```

try
{
    ...
}
catch (int i)
{
    ...
}
catch (const int& i) // error: duplicate exception
                      // handler
{
    ...
}

```

## 14.2 Multiple **catch** Statements

The previous section illustrated that multiple **catch** statements are allowed provided they do not duplicate one another:

```
// types.cpp
// illustrates multiple catch statements
#include <iostream.h> // C++ I/O

char NEG_NO[25] = "negative number entered!";

void main ()
{
    int number (0);

    cout << "enter a positive integer: " ;
    cin  >> number ;

    // code throws an exception
    try
    {
        if (number < 0)
            // throw char* exception to handler
            throw NEG_NO ;
        else
            // throw int exception to handler
            throw number ;
    }
    // process exceptions
    catch (int i)
    {
        cout << "number entered: " << i << endl ;
    }
    catch (char* string)
    {
        cout << string << endl ;
    }
}
```

If a user enters a positive integer then the type **int** expression is thrown, else a **char\*** expression is thrown. Each **catch** statement deals with a specific type of **throw** expression. Note that for each **catch** statement either an integer or a string is passed to the **catch** statement block as a temporary variable or object. The operand of the **throw** expression is identical in principle to an argument in a function call or the operand of a **return** statement. If the argument to the **catch** statement is not used it can be neglected:

```
//...
catch (int) // note: no variable
{
    cout << "negative number entered!" << endl ;
}
```

## 14.3 Handling all Exceptions

C++ supports a certain kind of **catch** statement which catches all exceptions regardless of type or **class**. This **catch** statement uses an ellipsis (...) in place of an exception declaration and is of the general syntax:

```
catch (...) // catch all exceptions
{
//...
}
```

This catch-all exception is illustrated in the following program:

```
// catch_all.cpp
// illustrates the catch-all statement
#include <iostream.h> // C++ I/O

char NEG_NO[25] = "negative number entered!" ;

void main ()
{
    int number (0) ;

    cout << "enter a positive integer: " ;
    cin >> number ;

    // code throws an exception
    try
    {
        if (number < 0)
            // throw char* exception to handler
            throw NEG_NO ;
        else
            // throw int exception to handler
            throw number ;
    }
    // process all exceptions
    catch (...)
    {
        cout << "I catch all exceptions" ;
    }
}
```

The catch-all statement is particularly useful for handling exceptions that are not explicitly dealt with in a set of **catch** statements. The catch-all statement is akin to the **default** case in a **switch** statement. The following program illustrates the combined use of an **int catch** statement and a catch-all statement:

```
// int&all.cpp
// illustrates both catch-int and catch-all statements
```

```
#include <iostream.h> // C++ I/O

void main ()
{
    int number (0) ;
    double d (0.0) ;

    cout << "enter a positive integer: "
    cin >> number ;

    // code throws an exception
    try
    {
        if (number < 0)
            // throw double exception to handler
            throw d ;
        else
            // throw int exception to handler
            throw number ;
    }
    // process exceptions
    catch (int i) // int exceptions
    {
        cout << "number entered: " << i << endl ;
    }
    catch (...) // all other exceptions
    {
        cout << "negative number entered!" ;
    }
}
```

Note that the **catch**(...) handler must be placed after all other handlers for a given **try** block. Thus, the following is illegal:

```
//...
catch (...) // all other exceptions
{
    cout << "negative number entered!" ;
}
catch (int i) // int exceptions
{
    cout << "number entered: " << i << endl ;
}
```

## 14.4 Functions and Exception Handling

Up to now we have seen the use of **try**, **catch** and **throw** restricted to a single function (namely the *main()* function), and all exceptions have been thrown from within the **try**

statement block. Let's now examine throwing an exception from a function other than *main()* that is outside a **try** block:

```
// out_try.cpp
// illustrates throwing an exception from outside a
// try block
#include <iostream.h> // C++ I/O

// throws an int exception
void Function (int i)
{
    throw i ;
}

void main ()
{
    try
    {
        Function (0) ;
        Function (1) ;
    }
    // process int exceptions
    catch (int i)
    {
        cout << "caught: " << i << endl ;
    }
}
```

with output:

```
caught: 0
```

Note that only one exception is thrown although there are two calls to *Function()*.

The **int** type exception thrown from *Function()* is outside the **try** block within *main()*. This is possible because the **throw** expression is in the body of a function that is called from within the **try** block.

The **try-catch-throw** setup can be transferred entirely to a function other than *main()*:

```
// tct_func.cpp
// illustrates the try-catch-throw setup from within a
function
#include <iostream.h> // C++ I/O

// try-catch-throw within function
void Function (int i)
{
    try
    {
        throw i ;
    }
```

```
    catch (int i)
    {
        cout << "caught number: " << i << endl ;
    }
}

void main ()
{
    Function (0) ;
    Function (1) ;
}
```

with output:

```
caught number: 0
caught number: 1
```

The output illustrates that each time *Function ()* is called the exception handling statements of the function are reset.

#### 14.4.1 Exception Specification

C++ allows a function declaration to specify exactly what set of exceptions the function can **throw** via an *exception specification*. The exception specification is used as a suffix to a function declarator and is of the general syntax:

```
return_type FunctionName (argument_list) throw (type_list)
{
//...
}
```

The exception specification is not part of a function's type. Both *argument\_list* and *type\_list* are optional, and an empty exception specification (i.e. **throw ()**) indicates that a function cannot **throw** any exception. A function with no exception specification is capable of throwing any exception. Each of the different types or classes that can be thrown by a function is separated by the comma operator in the *type\_list* argument. A function that attempts to **throw** an exception that is not listed in its exception specification will result in a call to the *unexpected()* function. The default behaviour of *unexpected()* is to call *abort()*, which in turn causes abnormal program termination. A description of *unexpected()* is given later.

The following program illustrates exception specifications:

```
// ex_spec.cpp
// illustrates exception specifications
#include <iostream.h> // C++ I/O

// throw only int & double
void FThrowIntDouble (int i) throw (int, double)
```

```

{
if (i==0)
    throw 0 ;      // throw int
if (i==1)
    throw 0.1 ;   // throw double
}

// throw no exceptions
void FThrowNone (int i) throw ()
{
// these statements can't be thrown!
if (i==0)
    throw 0 ;      // throw int
if (i==1)
    throw 0.1 ;   // throw double
}

// throw any exceptions
void FThrowAny (int i)
{
if (i==0)
    throw 0 ;      // throw int
if (i==1)
    throw 0.1 ;   // throw double
}

void main ()
{
try
{
//FThrowIntDouble (0) ; // o.k.
//FThrowIntDouble (1) ; // o.k.
//FThrowAny (1) ;       // o.k.
FThrowNone (0) ;           // abnormal program termination
}
catch (int i)
{
cout << "caught int: " << i << endl ;
}
catch (double d)
{
cout << "caught double: " << d << endl ;
}
}

```

The function *FThrowIntDouble()* can throw either **int** or **double** exceptions, but if an attempt is made to throw any other type of exception abnormal program termination will occur and the *unexpected()* function will be called. *FThrowNone()* cannot throw any exceptions and an attempt to do so will result in an abnormal program termination. *FThrowAny()* illustrates that a function without an exception specification can throw any exception.

## 14.5 Re-Throwing an Exception

An exception can be re-thrown. This is accomplished by using the **throw** keyword without an operand and causes the original exception to be caught by an outer **try-catch** block of statements. The following program illustrates re-throwing an exception within the *Function1()* function, which throws back the original exception that is caught by the **catch** statement within *Function0()*:

```
// rethrow.cpp
// illustrates re-throwing an exception
#include <iostream.h> // C++ I/O

// prototype
void Function1 (int i) ;

void Function0 (int i)
{
    try
    {
        Function1 (i) ;
    }
    catch (int i)
    {
        cout << "Function0(): " << i << endl ;
    }
}

void Function1 (int i)
{
    try
    {
        if (i== -1)
            throw i ;
    }
    catch (int i)
    {
        cout << "Function1(): " << i << endl ;
        // re-throw exception
        throw ;
    }
}

void main ()
{
    Function0 (-1) ;
}
```

with output:

```
Function1(): -1
Function0(): -1
```

## 14.6 EXCEPT.H Header File

The EXCEPT.H header file contains several function and **class** declarations for exception handling: *terminate()*, *set\_terminate()*, *unexpected()*, *set\_unexpected()* and classes *xalloc* and *xmsg*. Each of these functions and classes is briefly described below.

### 14.6.1 The *terminate()* Function

When an exception is thrown but not caught by an associated handler, the *terminate()* function is called:

```
void terminate () ; // EXCEPT.H
```

The default action taken by *terminate()* is to call the *abort()* function, which results in immediate abnormal program termination. Alternatively, if you do not want a program to terminate with a call to *abort()* you can define a terminate function which is called instead. The terminate function must be registered using the *set\_terminate()* function. If a terminate function has been registered, *terminate()* will execute the most recent function given as an argument to *set\_terminate()*.

The *abort()* C++ library function has the following signature:

```
void abort () ; // STDLIB.H
```

and is similar to the *exit()* function:

```
void exit (int status) ; // STDLIB.H
```

*exit()* terminates program execution. Before termination commences, all files are closed, buffered output is written and any registered exit functions are called. *status* can be assigned either of the following identifiers:

```
EXIT_SUCCESS (0) Normal program termination.  
EXIT_FAILURE (1) Abnormal program termination.
```

Unlike *exit()*, the *abort()* function does not return any termination information to the operating system.

### 14.6.2 The *set\_terminate()* Function

It was noted above that the default action taken by *terminate()* is to call *abort()*. Alternatively, *terminate()* calls the last function given as an argument to the *set\_terminate()* function:

```
typedef void (*terminate_function) () ; // EXCEPT.H  
terminate_function set_terminate (terminate_function t_func) ;
```

The *set\_terminate()* function gives you more control over the actions to be taken when a handler to an exception cannot be found. The sequence of actions to be taken when abnormal program termination occurs is defined in the function pointed to by *t\_func*. The return value of *set\_terminate()* is the previous function given as an argument to the function. The

definition of the function pointed to by `t_func` must terminate the program, since it is an error to return the function pointed to by `t_func` to its caller. Also, it is an error for the function pointed to by `t_func` to **throw** an exception.

The following program illustrates the `set_terminate()` function by installing an abnormal termination function called `MyTerminate()`, which displays a ‘program aborted’ message and then calls `exit()`:

```
// set_term.cpp
// illustrates the set_terminate() function
#include <iostream.h> // C++ I/O
#include <except.h> // exception handling, exit()

// abnormal termination function
void MyTerminate ()
{
    cout << "program aborted" << endl ;
    exit (EXIT_FAILURE) ;
}

void Function () throw (int)
{
    int number (0) ;

    cout << "enter a positive integer: " ;
    cin >> number ;

    if (number < 0)
        throw number ;
    else
        cout << "number entered: " << number << endl ;
}

void main ()
{
    // install MyTerminate() function
    set_terminate (MyTerminate) ;

    try
    {
        Function () ;
    }
    catch (double)
    {
        cout << "negative number entered!" << endl ;
    }
}
```

and some user interaction:

```
enter a positive integer: -7
program aborted
```

The `MyTerminate()` function is called because an `int` type exception is thrown to which there is no corresponding handler.

### 14.6.3 The `unexpected()` Function

The signature of the `unexpected()` function is:

```
void unexpected () ; // EXCEPT.H
```

The `unexpected()` function is called when a function throws an exception which is not declared in its exception specification. The default action taken by `unexpected()` is to call `terminate()`, which in turn calls `abort()`. Alternatively, if you do not want `unexpected()` to call `terminate()` you can define an `unexpected` function which is called instead. If an `unexpected` function has been registered, then `unexpected()` executes the last function given as an argument to the `set_unexpected()` function.

### 14.6.4 The `set_unexpected()` Function

The `set_unexpected()` function allows a programmer to install a function which defines program behaviour when a function throws an exception that is not in its exception specification. The signature of `set_unexpected()` is:

```
typedef void (*unexpected_function) () ; // EXCEPT.H
unexpected_function set_unexpected (unexpected_function
                                  u_func) ;
```

The sequence of actions to be taken when a function throws an exception not listed in its exception specification is defined in the function pointed to by `u_func`. The return value of `set_unexpected()` is the previous function given as an argument to the function. If a function is registered with `set_unexpected()` the `unexpected()` function is still called automatically and then control is passed to the registered function pointed to by `u_func`. Also, the function pointed to by `u_func` can call the functions `abort()`, `exit()` or `terminate()`.

The following program illustrates the `set_unexpected()` function:

```
// set_unx.cpp
// illustrates the set_unexpected() function
#include <iostream.h> // C++ I/O
#include <except.h> // exception handling, exit()

// abnormal unexpected exception specification function
void MyUnexpected ()
{
    cout << "program aborted" << endl ;
    exit (EXIT_FAILURE) ;
}

void Function () throw (double)
{
    int number (0) ;
```

```

cout << "enter a positive integer: " ;
cin  >> number ;

if (number < 0)
    throw number ;
else
    cout << "number entered: " << number << endl ;
}

void main ()
{
// install MyUnexpected() function
set_unexpected (MyUnexpected) ;

try
{
    Function () ;
}
catch (int)
{
    cout << "negative number entered!" << endl ;
}
}

```

and some user interaction:

```

enter a positive integer: -7
program aborted

```

The *MyUnexpected()* function is called because a type **int** exception is thrown from *Function()*, but the exception specification of *Function()* does not support this type.

#### 14.6.5 The **xmsg** and **xalloc** Classes

The **xmsg** **class** is declared in EXCEPT.H and is of the form:

```

class xmsg
{
private:
    string* str ;
public:
    xmsg (const string& msg) ;
    xmsg (const xmsg& msg) ;
    ~xmsg () ;
    const string& why() const { return *str ; }
    void raise() throw (xmsg) ;
    xmsg& operator = (const xmsg& src) ;
};

```

As the **class** name suggests, **xmsg** is used for reporting messages that are related to an exception. The constructor sets the **string** data member to a given message. The C++ library **string** **class** is defined in the **CSTRING.H** header file. The *raise()* member function causes an **xmsg** exception to be thrown, whereas *why()* returns the **string** data member.

The **class** **xalloc** reports an error associated with an allocation request:

```
class xalloc : public xmsg // xalloc inherited from xmsg
{
private:
    size_t size ;
public:
    xalloc (const string& msg, size_t size) ;
    size_t requested () const { return size ; }
    void raise () throw (xalloc) ;
};
```

**xalloc** is inherited from **class** **xmsg**. Inheritance is covered in more detail in the next chapter. The **xalloc** **class** constructor defines a **string** message which will be reported when a requested size allocation in bytes, **size**, cannot be allocated. The *raise()* member function causes a **xalloc** exception to be thrown, whereas *requested()* returns the value of the **size** data member.

The **xalloc** and **xmsg** classes will be examined further in the header file **VEC\_X1.H** below and in the next chapter, when we discuss inheritance with reference to the C++ **string** library **class**.

## 14.7 Constructors, Copy Constructors and Destructors

Exception handling generates its fair share of calls to a **class**'s constructors, copy constructor and destructor. Consider the following program:

```
// con&des.cpp
// illustrates calls to constructors,
// copy constructors and destructors with exception handling
#include <iostream.h> // C++ I/O

class X
{
private:
    int data ;
public:
    X ()
    { cout << "constructor called" << endl ;
      data = 0 ; }
    X (const X& x)
    { cout << "copy constructor called" << endl ;
      data = x.data ; }
    ~X ()
    { cout << "destructor called" << endl ; }
};
```

```
void Function (const X& x) throw (X)
{
    throw x ;
}

void main ()
{
    try
    {
        X x ;
        Function (x) ;
    }
    catch (X)
    {
        cout << "X exception caught" << endl ;
    }
}
```

with output:

```
constructor called
copy constructor called
copy constructor called
destructor called
destructor called
copy constructor called
X exception caught
destructor called
destructor called
```

If the **throw** expression, **throw x;**, is commented out in the above program, then the program output is as expected:

```
constructor called
destructor called
```

When an exception is thrown the copy constructor of the thrown object is called. The copy constructor call initialises a temporary object at the point where the exception is thrown (the **throw** point). When a program is interrupted by an exception, destructors are called for all objects created since the **try** block was first entered. An object that has only been partially created when an exception is thrown (e.g. an array of objects) will only have destructors called for the objects that have been fully constructed.

## 14.8 Exception Handling and a Vector **class**

Let us now examine the Vector **class** with respect to exception handling. On examining the Vector **class** declared and defined in the header file VEC\_TMP.H of Chapter 13 we observe that there are three key repeatable exceptions to handle: memory, size and range. Allocating memory dynamically using the **new** operator requires us to test the return pointer

against the NULL pointer to verify that the memory request was successful. The size (i.e. the number of elements) of a `Vector` object must be verified when a `Vector` object is defined and during assignment and arithmetical operations, such as the overloaded `=`, `+`, `-` and `*` operators. Each time an element of a `Vector` is indexed (using the `Value()` member functions or the overloaded `[]` operator of `VEC_TMP.H`) a range test must be performed. The `Vector` **class** of `VEC_TMP.H` relied heavily on the `assert()` macro. The `Vector` **class** below is a modification of `VEC_TMP.H` using the exception handling techniques outlined in the present chapter:

```
// vec_x.h
// template class Vector with exception handling

#ifndef _VEC_X_H // prevent multiple includes
#define _VEC_X_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...

#define NDEBUG (0) // debugging:ON

// template/exception handling Assert() function
template <class T, class X>
inline void Assert (T exp, X x)
{
    if (!NDEBUG) // compatible with assert() macro NDEBUG
        if (!exp)
            throw x ;
}

// template class Vector
template <class T>
class Vector
{
private:
    // data members
    T* array ;
    int n_elements ;
public:
    // exception handling classes
    class Memory {};
    class Size {};
    class Range {};
    // constructors
    Vector () { Allocate (3) ; }
    Vector (int n, T obj) ;
    Vector (const Vector& v) ;
    // destructor
    ~Vector () throw () { delete [] array ; array = NULL ;
                           n_elements = 0 ; }
    // member functions
    void Allocate (int n) throw (Vector<T>::Size,
```

```

                                Vector<T>::Memory) ;
int      NumberElements () const { return n_elements ; }
T&      Value (int index) throw (Vector<T>::Range) ;
const T& Value (int index) const
                                throw (Vector<T>::Range) ;
void    New (int new_n) ;
// overloaded operators
Vector& operator = (const Vector& v)
                                throw (Vector<T>::Size) ;
T&      operator [] (int index)
                                throw (Vector<T>::Range) ;
const T& operator [] (int index) const
                                throw (Vector<T>::Range) ;
Vector   operator + (const Vector& v)
                                throw (Vector<T>::Size) ;
Vector   operator - (const Vector& v)
                                throw (Vector<T>::Size) ;
Vector   operator * (const Vector& v)
                                throw (Vector<T>::Size) ;
// friend
friend ostream& operator << (ostream& s,
                                const Vector<T>& v) ;
}; // class Vector

// allocates memory for elements of Vector<T>
template <class T>
void Vector<T>::Allocate (int n) throw (Vector<T>::Size,
                                Vector<T>::Memory)
{
if (n <= 0)
    throw Size () ;

array = new T[n] ;
Assert (array!=NULL, Memory ()) ;
//if (array == NULL)
//    throw Memory () ;

n_elements = n ;
for (int i=0; i<n; i++)
    array[i] = T () ;
}

// 2 arg. constructor
template <class T>
Vector<T>::Vector (int n, T obj)
{
Allocate (n) ;
for (int i=0; i<n; i++)
    array[i] = obj ;
}

```

```

// copy constructor
template <class T>
Vector<T>::Vector (const Vector& v)
{
    Allocate (v.NumberElements()) ;
    for (int i=0; i<v.NumberElements(); i++)
        *(array+i) = v.Value (i) ;
}

// public member functions:

// returns element value (non-const object)
template <class T>
T& Vector<T>::Value (int index) throw (Vector<T>::Range)
{
    if (index < 0 || index >= n_elements)
        throw Range () ;
    return *(array+index) ;
}

// returns element value (const object)
template <class T>
const T& Vector<T>::Value (int index) const throw
(Vector<T>::Range)
{
    if (index < 0 || index >= n_elements)
        throw Range () ;
    return *(array+index) ;
}

// change number of elements
template <class T>
void Vector<T>::New (int new_n)
{
    delete [] array ;
    array = NULL ;
    n_elements = 0 ;

    Allocate (new_n) ;
}

// overloaded operators:

// assignment operator =
template <class T>
Vector<T>& Vector<T>::operator = (const Vector& v)
                                         throw (Vector<T>::Size)
{
    // verify that the two vectors are of equivalent dimensions
    if (n_elements != v.NumberElements())
        throw Size () ;
}

```

```
for (int i=0; i<v.NumberElements(); i++)
    *(array+i) = v.Value (i) ;
return *this ;
}

// subscript operator [] (non-const object)
template <class T>
T& Vector<T>::operator [] (int index) throw
    (Vector<T>::Range)
{
if (index < 0 || index >= n_elements)
    throw Range () ;
return *(array+index) ;
}

// subscript operator [] (const object)
template <class T>
const T& Vector<T>::operator [] (int index)
    const throw (Vector<T>::Range)
{
if (index < 0 || index >= n_elements)
    throw Range () ;
return *(array+index) ;
}

// addition operator +
template <class T>
Vector<T> Vector<T>::operator + (const Vector& v)
    throw (Vector<T>::Size)
{
// verify that the two vectors are of equivalent
// dimensions
if (n_elements != v.NumberElements())
    throw Size () ;

Vector<T> add (v.NumberElements(),T()) ;

for (int i=0; i<v.NumberElements(); i++)
    add.Value (i) = Value (i) + v.Value (i) ;
return add ;
}

// subtraction operator -
template <class T>
Vector<T> Vector<T>::operator - (const Vector& v)
    throw (Vector<T>::Size)
{
// verify that the two vectors are of equivalent
// dimensions
if (n_elements != v.NumberElements())
    throw Size () ;
```

```

Vector<T> sub (v.NumberElements(),T()) ;

for (int i=0; i<v.NumberElements(); i++)
    sub.Value (i) = Value (i) - v.Value (i) ;
return sub ;
}

// multiplication operator *
template <class T>
Vector<T> Vector<T>::operator * (const Vector& v)
                                         throw (Vector<T>::Size)
{
// verify that the two vectors are of equivalent dimensions
if (n_elements != v.NumberElements())
    throw Size () ;

Vector<T> mult (v.NumberElements(),T()) ;

for (int i=0; i<v.NumberElements(); i++)
    mult.Value (i) = Value (i) * v.Value (i) ;
return mult ;
}

// friend:

// overloaded insertion operator <<
template <class T>
ostream& operator << (ostream& s, const Vector<T>& v)
{
s << "[ " ;
for (int i=0; i<v.n_elements; i++)
{
s << setprecision (2)
  << setiosflags (ios::showpoint | ios::fixed)
  << v.Value (i) ;
i!=v.n_elements-1 ? s << ", " : s << " ] " ;
}
return s ;
}

#endif // _VEC_X_H

```

The following program tests the above Vector **template class**:

```

// vec_tst.cpp
// tests the exception handling features
// of the template class Vector
#include <iostream.h> // C++ I/O
#include "vec_x.h"      // template class Vector

void main ()

```

```

{
try
{
    Vector<int> v (5, 0) ;
    cout << "Vector<int> v: " << v << endl ;

    // run-time range error
    cout << "v[5]: " << v[5] << endl ;
}
catch (Vector<int>::Memory)
{
    cout << "insufficient memory for allocation" << endl ;
}
catch (Vector<int>::Size)
{
    cout << "incorrect Vector<T> size defined" << endl ;
}
catch (Vector<int>::Range)
{
    cout << "out-of-bounds indexing of Vector<T> elements"
        << endl ;
}
}
}

```

with output:

```

Vector<int> v: [0, 0, 0, 0, 0]
out-of-bounds indexing of Vector<T> elements

```

The above program code illustrates several new features which are worth examining separately.

### 14.8.1 Exception Classes

The **template class** Vector incorporates three **publicly** declared empty exception nested classes called Memory, Size and Range:

```

template <class T>
class Vector
{
//...
public:
    // exception handling classes
    class Memory {};
    class Size   {};
    class Range  {};
//...
};

```

The sole purpose of these three exception classes is to enable **template class** Vector to **throw** a memory, size or range exception to appropriate exception handlers. The three

exception classes are nested classes, because it is assumed at present that these types of exception are specific to Vector. Also, the three exception classes are **publicly** declared to allow the exception handlers access to the nested classes.

### 14.8.2 Constructors

Since constructors have no return type, it is difficult to report an exception to a caller when an object is defined. The C++ method of exception handling allows a construction error to be conveniently thrown out of the constructor and dealt with accordingly. This is illustrated above in VEC\_X.H, although indirectly, through the use of the *Allocate()* member function:

```
template <class T>
Vector<T>::Vector (int n, T obj)
{
    Allocate (n) ;
    //...
}
//...
template <class T>
void Vector<T>::Allocate (int n) throw (Vector<T>::Size,
                                            Vector<T>::Memory)
{
    if (n <= 0)
        throw Size () ;
    //...
}
```

If a user defines a Vector object with a size dimension less than or equal to zero, a *Size* exception is thrown allowing the user to **catch** the exception and deal with the error:

```
void main ()
{
    try
    {
        Vector<int> v(5, 0) ;
        //...
    }
    //...
    catch (Vector<int>::Size)
    {
        cout << "incorrect Vector<T> size defined" << endl ;
    }
    //...
}
```

### 14.8.3 *Assert()* **template** Function

The **template** Vector and Matrix classes of Chapter 13 (VEC\_TMP.H and MTRX\_TMP.H) made good use of the C++ *assert()* macro for testing a given assertion:

```
template <class T>
void Vector<T>::Allocate (int n)
{
    assert (n) ;

    array = new T[n] ;
    assert (array != NULL) ;
    //...
}
```

Alternatively, we can define a **template** exception-handling *Assert ()* function (Stroustrup, 1994, pp. 397–8):

```
template <class T, class X>
inline void Assert (T exp, X x)
{
    if (!NDEBUG) // compatible with assert() macro NDEBUG
        if (!exp)
            throw x ;
}
```

*Assert ()* throws an exception of type or **class** *X* if *exp* is logical-false. A typical use of *Assert ()* is given in the *Allocate ()* member function and tests whether the **new** operator cannot allocate the requested block of memory:

```
template <class T>
void Vector<T>::Allocate (int n) throw (Vector<T>::Size,
                                         Vector<T>::Memory)
{
    //...
    array = new T[n] ;
    Assert (array!=NULL, Memory ()) ;
    //...
}
```

If the memory request is unsuccessful a *Memory* exception is thrown.

#### 14.8.4 Declaring Exceptions

Several member functions and overloaded operator functions of **template class** *Vector* incorporate exception specifications as part of their function declarator:

```
template <class T>
class Vector
{
    //...
    void Allocate (int n) throw (Vector<T>::Size,
                                Vector<T>::Memory) ;
public:
    //..
    ~Vector () throw () { delete [] array ; array = NULL ; }
```

```

        n_elements = 0 ; }

//...
Vector& operator = (const Vector& v)
    throw (Vector<T>::Size) ;
//...
} ;

```

The declaration of **template class** Vector indicates that *Allocate()* can **throw** exceptions of classes Size and Memory, but no others. The Vector **class** destructor cannot **throw** any exception. Since a function declaration that has no exception specification can **throw** any type of exception, the above exception specifications could be viewed as redundant. However, explicitly declaring the types of exception that a member function can **throw** as part of a **class** declaration makes a **class** declaration much more expressive.

## 14.9 Passing Information with an Exception

The above use of exception handling with the Vector **template class** demonstrated throwing exceptions with the help of three exception classes: Memory, Size and Range. However, because these three exception classes are empty classes (i.e. no data members) no information is sent to the appropriate handlers, via the thrown objects, concerning the exact reason for an exception. Therefore, let us extend the Memory, Size and Range exception classes:

```

// vec_x1.h
// template class Vector with exception handling
//...
// template class Vector
template <class T>
class Vector
{
private:
    // data members
    T* array ;
    int n_elements ;
public:
    // exception handling classes
    class Memory
    {
public:
        xalloc xa_mem ;
        Memory (const string& msg, int alloc)
            : xa_mem (msg, (size_t)alloc) {}
    };
    class Size
    {
public:
        xalloc xa_size ;
        Size (const string& msg, int size)
            : xa_size (msg, (size_t)size) {}

```

```

    };
    class Range
    {
    public:
        xmmsg xm_range ;
        int index ;
        Range (const string& msg, int i)
            : xm_range (msg), index (i) {}
    };
    //...
};

// public member functions:

// allocates memory for elements of Vector<T>
template <class T>
void Vector<T>::Allocate (int n) throw (Vector<T>::Size,
                                            Vector<T>::Memory)
{
    if (n <= 0)
        throw Size (string("incorrect Vector<T> size defined"),
                    n) ;

    array = new T[n] ;
    Assert (array!=NULL,
            Memory (string("insufficient memory for
                            allocation"), n*sizeof(T))) ;

    n_elements = n ;
    for (int i=0; i<n; i++)
        array[i] = T () ;
}

// returns element value (non-const object)
template <class T>
T& Vector<T>::Value (int index) throw (Vector<T>::Range)
{
    if (index < 0 || index >= n_elements)
        throw Range (string("out-of-bounds indexing of Vector<T>
                            elements"), index) ;
    return *(array+index) ;
}
//...

// overloaded operators:

// assignment operator =
template <class T>
Vector<T>& Vector<T>::operator = (const Vector& v)
                                         throw (Vector<T>::Size)
{

```

```

// verify that the two vectors
// are of equivalent dimensions
if (n_elements != v.NumberElements())
    throw Size (string("Vector<T> objects must be equivalent
                    for assignment"),
                v.NumberElements()) ;

for (int i=0; i<v.NumberElements(); i++)
    *(array+i) = v.Value (i) ;
return *this ;
}

// subscript operator [] (non-const object)
template <class T>
T& Vector<T>::operator [] (int index) throw
(Vector<T>::Range)
{
    if (index < 0 || index >= n_elements)
        throw Range (string("out-of-bounds indexing of Vector<T>
                           elements"), index) ;
    return *(array+index) ;
}
//...
#endif // _VEC_X_H

```

Program VEC\_TST1.CPP tests the above **template** Vector **class**:

```

// vec_tst1.cpp
// tests the exception handling features of the template
// class Vector having modified the Memory, Size and Range
// exception classes
#include <iostream.h> // C++ I/O
#include <cstring.h> // C++ string class
#include "vec_x1.h" // template class Vector

void main ()
{
    try
    {
        Vector<int> v (5, 0) ;
        cout << "Vector<int> v: " << v << endl ;

        // run-time range error
        cout << "v[5]: " << v[5] << endl ;
    }
    catch (Vector<int>::Memory m)
    {
        cout << m.xa_mem.why () << endl
            << m.xa_mem.requested ()
            << " bytes requested" << endl ;
    }
}

```

```

catch (Vector<int>::Size s)
{
    cout << s.xa_size.why() << endl ;
}
catch (Vector<int>::Range r)
{
    cout << r.xm_range.why ()      << endl
        << "index: " << r.index << endl ;
}
}

```

with output:

```

Vector<int> v: [0, 0, 0, 0, 0]
out-of-bounds indexing of Vector<T> elements
index: 5

```

The nested classes **Memory** and **Size** both encapsulate a data member of **class** **xalloc**, and their constructors initialise the **str** and **size** (**string\*** **str** and **size\_t size**) data members of **class** **xalloc**. Nested **class** **Range** encapsulates data members of types **xmsg** and **int** (i.e. **xm\_range** and **index**), which are similarly initialised via **Range**'s two-argument constructor.

A **Memory**, **Size** or **Range** exception can now be thrown with the appropriate information relating to the exception rather than simply the type of exception:

```

template <class T>
void Vector<T>::Allocate (int n) throw (Vector<T>::Size,
                                         Vector<T>::Memory)
{
    if (n <= 0)
        throw Size (string("incorrect Vector<T> size defined"),
                    n) ;

    array = new T[n] ;
    Assert (array!=NULL,
            Memory (string("insufficient memory for
                           allocation"), n*sizeof(T))) ;
//...
}

```

A thrown object is examined by a handler by giving a name to the exception object:

```

void main ()
{
//...
catch (Vector<int>::Size s)
{
    cout << s.xa_size.why() << endl ;
}
//...
}

```

### 14.9.1 Range and Index Classes

Previous sections have illustrated the use of nested exception handling classes Memory, Size and Range for the **template class** Vector:

```
template <class T>
class Vector
{
//...
public:
    // exception handling classes
    class Memory
    {
    //...
    };
    class Size
    {
    //...
    };
    class Range
    {
    //...
    };
//...
};
```

However, classes Memory, Size and Range could equally be used by a variety of classes other than Vector, and not only as exception handling classes. As a result we shall now remove the nested Memory, Size and Range classes from the declaration of **template class** Vector and develop a Range **class**. Although the Memory and Size classes could equally be developed, we shall rely on the C++ library exception-handling classes xmsg and xalloc declared in the EXCEPT.H header file for handling both memory and size exceptions. In addition to the Range **class**, an Index **class** is developed to assist the user in accessing the elements of a Vector.

In working with arrays, vectors and matrices we are continually accessing elements within a bounded range. Thus, it would be nice to have a **class** which is specifically designed to work with ranges. Such a **class**, called Range, is declared below:

```
// range.h
// class Range

#ifndef _RANGE_H // prevent multiple includes
#define _RANGE_H

#include <iostream.h> // C++ I/O

enum Boolean { B_FALSE, B_TRUE };

class Range
{
private:
```

```

        int min, max ;
public:
    // constructors
    Range ()
        : min (0), max (0) {}
    Range (int _max)
        : min (0), max (_max) {}
    Range (int _min, int _max)
        : min (_min), max (_max) {}
    // copy constructor
    Range (const Range& r)
        : min (r.min), max (r.max) {}
    // member functions
    int Min () const { return min ; }
    int Max () const { return max ; }
    unsigned int Mag () const { return max - min + 1 ; }
    Boolean In (int i) { return i >= min && i <= max
                           ? B_TRUE : B_FALSE ; }
    Boolean Out (int i) { return i < min || i > max
                           ? B_TRUE : B_FALSE ; }
    // overloaded operators
    Range& operator = (const Range& r)
    { min = r.min ; max = r.max ; return *this ; }
    Boolean operator == (const Range& r)
    { return min==r.min && max==r.max
          ? B_TRUE : B_FALSE ; }
    Boolean operator != (const Range& r)
    { return min!=r.min || max!=r.max
          ? B_TRUE : B_FALSE ; }
    // friend
    friend ostream& operator << (ostream& s, const Range& r)
    { return s << "(" << r.min << ", " << r.max
      << ")" ; }
}; // class Range

#endif // _RANGE_H

```

Range encapsulates two **int** data members which represent the lower and upper bounds of a given range. The four constructors of Range allow a user to define a Range object with ranges 0:0, 0:max and min:max, and to initialise one Range object to another. The member functions *In()* and *Out()* test whether an **int** number is within or outside a range, respectively.

To assist the user in indexing elements of an array, vector or matrix within a given range, let us develop an **Index class**:

```

// index.h
// class Index

#ifndef _INDEX_H // prevent multiple includes
#define _INDEX_H

```

```

#include <iostream.h> // C++ I/O
#include "assert.h" // Assert()
#include "range.h" // class Range

class Index
{
private:
    Range range ;
    int index ;
public:
    // constructors
    Index (int _min, int _max)
        : range (_min, _max), index (_min) {}
    Index (const Range& r)
        : range (r), index (r.Min()) {}
    Index (const Range& r, int _index)
        : range (r), index (_index) {}
    // copy constructor
    Index (const Index& i)
        : range (i.range), index (i.index) {}
    // member functions
    Range GetRange () const
    {
        return range ;
    }
    int Min () const
    {
        return range.Min () ;
    }
    int Max () const
    {
        return range.Max () ;
    }
    int Mag () const
    {
        return range.Mag () ;
    }
    // overloaded operators
    Index& operator = (const Index& i)
    {
        range = i.range ; index = i.index ;
        return *this ;
    }
    int operator [] (const Index& i)
    {
        return i.index ;
    }
    Boolean operator == (int i)
    {
        return index == i ? B_TRUE : B_FALSE ;
    }
    Boolean operator != (int i)
    {
        return index != i ? B_TRUE : B_FALSE ;
    }
    Index operator ++ () throw (Index) ; // prefix
    Index operator ++ (int) throw (Index) ; // postfix
    // conversion function
    operator int () const
    {
        return index ;
    }
    // friend
    friend ostream& operator << (ostream& s, const Index& i)
    {
        return s << i.range << " " << i.index ;
    }
}; // class Index

Index Index::operator ++ () throw (Index) // prefix
{

```

```

Assert (! (index>range.Max()) , *this) ;
index++ ; return *this ;
}
Index Index::operator ++ (int) throw (Index) // postfix
{
Assert (! (index>range.Max()) , *this) ;
index++ ; return Index (range, index-1) ;
}

#endif // _INDEX_H

```

Index encapsulates both Range and int objects, range and index. The majority of **class** Index is fairly straightforward, but note the conversion function **Index::operator int()**, which is used to cast an Index object to an int object. The **class** Index also makes use of the **Assert () template** function, which is defined in the ASSERT.H header file.

Note that an Index is *a kind of* Range, and a more suitable implementation of Index would be to derive Index from Range:

```

class Range
{
protected:
    int min, max ;
//...
};

class Index : public Range // Index derived from Range
{
protected:
    int index ;
//...
};

```

In this case, Index would inherit the Range data members min and max and the *Min()*, *Max()* and *Mag()* member functions of Range, thus eliminating the present replication of these member functions in the Index **class** and the Index::range data member. However, since we have not yet covered the topic of inheritance we will encapsulate a Range object within an Index object:

```

class Index // Index contains a Range object
{
private:
    Range range ;
//...
};

```

With the Range and Index classes at our disposal we can now redesign the Vector **template class** of VEC\_X1.H as follows:

```

// rng_idx.cpp
// illustrates the use of Range and Index classes

```

```

// with a Vector template class

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <cstring.h> // C++ class string
#include <except.h> // exception handling

#define NDEBUG (0) // debugging:ON

#include "assert.h" // Assert()
#include "range.h" // class Range
#include "index.h" // class Index

// template class Vector
template <class T>
class Vector
{
private:
    // data members
    T* array ;
    Range range ;
public:
    // constructors
    Vector () ;
    Vector (const Range& r, T obj) ;
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () throw () { delete [] array ; array = NULL ; }
    // member functions
    void Allocate (const Range& r) throw (xalloc) ;
    int NumberElements () const
    { return range.Mag () ; }
    T& Value (int i) throw (Index) ;
    const T& Value (int i) const throw (Index) ;
    T& Value (const Index& i) throw (Index) ;
    const T& Value (const Index& i) const throw (Index) ;
    // overloaded operators
    Vector& operator = (const Vector& v) throw (xalloc) ;
    T& operator [] (int i) throw (Index) ;
    T& operator [] (const Index& i) throw (Index) ;
    Vector operator + (const Vector& v) throw (xmsg) ;
    Vector operator - (const Vector& v) throw (xmsg) ;
    Vector operator * (const Vector& v) throw (xmsg) ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Vector<T>& v) ;
}; // template class Vector

// default constructor
template <class T>

```

```

Vector<T>::Vector ()
{
    Allocate (Range (2)) ;
}

// 2 arg. constructor
template <class T>
Vector<T>::Vector (const Range& r, T obj)
{
    Allocate (r) ;
    for (Index i (r); i<=r.Max(); i++)
    {
        *(array+int(i)-range.Min()) = obj ;
    }
}

// copy constructor
template <class T>
Vector<T>::Vector (const Vector& v)
{
    Allocate (v.range) ;
    for (Index i (v.range); i<=v.range.Max(); i++)
        *(array+int(i)-range.Min()) = v.Value (i) ;
}

// public member functions:

// allocates memory for elements of Vector<T>
template <class T>
void Vector<T>::Allocate (const Range& r) throw (xalloc)
{
    Assert (! (r.Mag()<=0),
            xalloc ("incorrect Vector<T> size defined", r.Mag())) ;

    array = new T[r.Mag()] ;
    Assert (array!=NULL,
            xalloc ("insufficient memory for allocation",
                    r.Mag()*sizeof(T))) ;

    range = r ;
    for (Index i (r); i<=r.Max(); i++)
        *(array+int(i)-range.Min()) = T () ;
}

// returns element value (non-const object)
template <class T>
T& Vector<T>::Value (int i) throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+i-range.Min()) ;
}

```

```

// returns element value (const object)
template <class T>
const T& Vector<T>::Value (int i) const throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+i-range.Min()) ;
}

// returns element value (non-const object)
template <class T>
T& Vector<T>::Value (const Index& i) throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+int(i)-range.Min()) ;
}

// returns element value (const object)
template <class T>
const T& Vector<T>::Value (const Index& i)
    const throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+int(i)-range.Min()) ;
}

// overloaded operators:

// assignment operator =
template <class T>
Vector<T>& Vector<T>::operator = (const Vector& v)
    throw (xalloc)
{
    // verify that the two vectors are of equivalent dimensions
    Assert (!(range != v.range),
            xalloc ("Vector<T> objects must be equivalent for
                    assignment", v.NumberElements())) ;

    for (Index i (v.range); i<=v.range.Max(); i++)
        *(array+int(i)-range.Min()) = v.Value (i) ;
    return *this ;
}

// subscript operator [] (non-const object)
template <class T>
T& Vector<T>::operator [] (int i) throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+i-range.Min()) ;
}

// subscript operator [] (non-const object)

```

```

template <class T>
T& Vector<T>::operator [] (const Index& i) throw (Index)
{
    Assert (!range.Out(i), Index (range, i)) ;
    return *(array+int(i)-range.Min()) ;
}

// addition operator +
template <class T>
Vector<T> Vector<T>::operator + (const Vector& v)
                                         throw (xmsg)
{
    // verify that the two vectors are of equivalent dimensions
    Assert (!(range != v.range),
            xmsg ("Vector<T> objects must be equivalent for
                  addition")) ;

    Vector<T> add (v.range, T()) ;
    for (Index i (v.range); i<=v.range.Max(); i++)
        add.Value (i) = Value (i) + v.Value (i) ;
    return add ;
}

// subtraction operator -
template <class T>
Vector<T> Vector<T>::operator - (const Vector& v)
                                         throw (xmsg)
{
    // verify that the two vectors are of equivalent dimensions
    Assert (!(range != v.range),
            xmsg ("Vector<T> objects must be equivalent for
                  subtraction")) ;

    Vector<T> sub (v.range, T()) ;
    for (Index i (v.range); i<=v.range.Max(); i++)
        sub.Value (i) = Value (i) - v.Value (i) ;
    return sub ;
}

// multiplication operator *
template <class T>
Vector<T> Vector<T>::operator * (const Vector& v) throw (xmsg)
{
    // verify that the two vectors are of equivalent dimensions
    Assert (!(range != v.range),
            xmsg ("Vector<T> objects must be equivalent for
                  multiplication")) ;

    Vector<T> mult (v.range, T()) ;
    for (Index i (v.range); i<=range.Max(); i++)
        mult.Value (i) = Value (i) * v.Value (i) ;
}

```

```

return mult ;
}

// friend:

// overloaded insertion operator <<
template <class T>
ostream& operator << (ostream& s, const Vector<T>& v)
{
    s << "[ " ;
    for (Index i(v.range); i<=v.range.Max(); i++)
    {
        s << setprecision (2)
            << setiosflags (ios::showpoint | ios::fixed)
            << v.Value (i) ;
        i!=v.range.Max() ? s << ", " : s << "]" ;
    }
    return s ;
}

void main ()
{
    try
    {
        Vector<int> v (Range (6, 10), 0) ;
        cout << "Vector<int> v: " << v << endl ;

        v[11] = 1 ; // out-of bounds indexing
    }
    catch (xalloc x)
    {
        cout << x.why () << " " << x.requested () << endl ;
    }
    catch (xmmsg x)
    {
        cout << x.why () << endl ;
    }
    catch (Range r)
    {
        cout << "Range exception caught " << r << endl ;
    }
    catch (Index i)
    {
        cout << "Index exception caught " << i << endl ;
    }
}

```

with output:

```

Vector<int> v: [0, 0, 0, 0, 0]
Index exception caught (6, 10) 11

```

The elements of a Vector object can now be within a bounded range specified when a Vector object is defined, as illustrated above within the *main()* function:

```
Vector<int> v (Range (6, 10), 0);
```

Vector no longer encapsulates an *n\_elements* data member as in VEC\_X1.H, but instead a *range* data member. However, since the elements of a Vector object are no longer indexed relative to zero we have to be careful when defining member functions which access elements of a Vector object. For instance, consider the *Allocate()* member function:

```
template <class T>
void Vector<T>::Allocate (const Range& r) throw (xalloc)
{
    //...
    for (Index i (r); i<=r.Max(); i++)
        *(array+int(i)-range.Min()) = T () ;
}
```

The **for**-loop defines and initialises (via a Range object) a control object *i* of **class** Index for performing an iteration through the elements of a newly created Vector object. The single statement of the **for**-loop body initialises each Vector object element to the default constructor of type *T* for the specified range *r*.

## 14.10 Exception Handling and the new Operator

Let's conclude the present chapter by examining the **new** operator with reference to exception handling. To date, when requesting a block of memory using the **new** operator we have explicitly tested the pointer returned by **new** to verify whether the memory request was successful or not:

```
//...
X* x = new X ;
if (x == 0)
{
    // memory allocation unsuccessful
}
//...
```

or:

```
//...
X* x = new X ;
assert (x != NULL) ;
//...
```

Testing every pointer returned by the **new** operator can become laborious and is easily forgotten. Fortunately, the **new** operator, by default, throws an *xalloc* exception if a memory allocation is unsuccessful:

```

// x_new.cpp
// illustrates exception handling and the new operator
#include <iostream.h> // C++ I/O
#include <cstring.h> // C++ string class
#include <except.h> // exception handling

const unsigned int SIZ = 1e04 ;

class X
{
private:
    double* data ;
public:
    X ()
        : data (new double[SIZ]) {}
    ~X ()
        { delete [] data ; }
};

void main ()
{
try
{
    X* x = new X[SIZ] ;
}
catch (xalloc xa)
{
    cout << xa.why () << " " << xa.requested () << endl ;
}
}

```

with output:

```
Out of memory 0
```

Note that the value, `xalloc::size`, returned by `xalloc::requested()` is zero. If the `new` operator is unsuccessful in allocating the requested amount of memory it returns 0 or NULL.

Alternatively, and in the same vein as the `set_terminate()` and `set_unexpected()` functions, a user-defined `new_handler` function can be defined which is called if a memory allocation using the `new` operator is unsuccessful. The `new_handler` function has no arguments and returns `void` and is registered by passing the `new_handler` function name to the `set_new_handler()` function. The signature of `set_new_handler()` is:

```
typedef void (*new_handler) () throw (xalloc) ; // NEW.H
new_handler set_new_handler (new_handler) ;
```

and is declared in the `NEW.H` header file. `set_new_handler()` returns the previous handler if one was registered.

The `new_handler` function should specify an appropriate sequence of actions to be taken when the `new` operator cannot successfully allocate a requested block of memory. The

`new_handler` function should free previously allocated redundant memory, `throw` an `xalloc` exception or initiate program termination.

An example of a user-defined `new_handler` function is given in the following program:

```
// set_hand.cpp
// illustrates the set_new_handler() function
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()
#include <new.h> // set_new_handler()

const unsigned int SIZ = 1e04 ;

class X
{
private:
    double* data ;
public:
    X ()
        : data (new double[SIZ]) {}
    ~X ()
        { delete [] data ; }
};

// abnormal termination function
void MyHandler ()
{
    cout << "MyHandler () called" << endl ;
    exit (EXIT_FAILURE) ;
}

void main ()
{
    // install MyHandler() function
    set_new_handler (MyHandler) ;

    X* x = new X[SIZ] ;
}
```

with output:

```
MyHandler () called
```

## 14.11 Summary

Exception handling enables communication to take place between the developer and a user of a function, `class` or library: ‘The author of a library can detect run-time errors but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors but cannot detect them – or else they would have been handled in the user’s code’

*and not left for the library to find. The notion of an exception is provided to help deal with such problems'* (Stroustrup, 1991, p. 293).

There are several different methods that can be adopted for handling exceptions when programming. The C++ *termination* method of exception handling provides a programmer with a structured, language-based way of dealing with exceptions. Exceptions can be passed or thrown to a *non-local* part of a program whose sole purpose is to deal with exceptions and errors. Exception handling in C++ is based around the three keywords: **try**, **catch** and **throw**. A block of program statements that is to be monitored for exceptions is prefixed by the keyword **try**. If, at run-time, an exception occurs within this monitored block of statements, the exception is thrown, using the **throw** keyword, to an appropriate handler which deals with a certain type of exception. The exception is caught, using the **catch** keyword, and is then processed accordingly.

## Exercises

- 14.1 Define a function called *Sqrt()* which simply calls the C++ standard library function *sqrt()*, but which accounts for the possibility of calculating a negative square root by declaring an exception handling **class** called *ExpressionException* and an appropriate exception specification for *Sqrt()*.
- 14.2 Exercises 9.2, 10.2 and 11.4 have examined an **Int class**. Exercise 10.2 added an overloaded modulus operator % to **Int**, but did not cater for the possibility of division by zero. By declaring an exception-handling **class** called *DivisionByZero*, modify the **Int::operator%** () member function so that division by zero exceptions can be caught.
- 14.3 Use the *set\_terminate()* function to install your own function which deals with uncaught exceptions.
- 14.4 Program ND\_STR.CPP of Chapter 12 declared a simple **String class**:

```
class String
{
private:
    char* string ; // pointer to char
    int length ; // length of string
public:
    String (const char str[] = 0) ;
    ~String () ;
    //...
};
```

Declare an exception handling **class** called *InsufficientMemory* and modify the one-argument constructor of **String** so that it now has an exception specification of **class** *InsufficientMemory*. Also, ensure that **String**'s destructor can **throw** no exceptions.

- 14.5 The following is a simpler implementation of the **Vector class** declared in VEC\_X.H:

```
template <class T>
class Vector
{
private:
```

```
// data members
T* array ;
int n_elements ;
void Allocate (int n) ;
public:
//...
T& operator [] (int index) ;
};
```

Declare an exception handling **class** called **OutOfBounds** to handle out-of-bounds indexing of the elements of a **Vector** object using the overloaded **operator[]()** member function.

- 14.6 Previous implementations of the **Vector::Allocate()** member function used the **assert()** macro to test whether the requested size of a **Vector** object is greater than zero:

```
template <class T>
void Vector<T>::Allocate (int n)
    throw (NegativeVectorSize)
{
    assert (n) ;
    //...
}
```

By declaring an exception handling **class** called **NegativeVectorSize**, modify the **Allocate()** member function so that **assert()** is no longer required.

- 14.7 By using the implicit **xalloc** exception handling capabilities of the **new** operator, rewrite the following **Vector::Allocate()** member function:

```
template <class T>
void Vector::Allocate (int n)
{
    //...
array = new T[n] ;
assert (array != NULL) ;
//...
n_elements = n ;
}
```

## Inheritance

*Inheritance, in C++, is the ability of a class to inherit the features of one or more other classes. It is exactly this ability of deriving from an existing class or classes which makes inheritance such an important feature in the C++ language, not only from the point of view of object-oriented programming, but also in the design and development of reusable code. Similar classes with inherent relationships can be placed into class hierarchies to assist in the conceptualisation and organisation of program code.*

*C++ gives us complete control in specifying inheritance relationships between classes, through the use of the private, protected and public access specifiers. Just as an object can always maintain private data while supporting a public interface to other objects, a class can equally control the access of its data members to derived classes. A derived class can also be derived from by other classes, and a class can be derived from multiple base classes – multiple inheritance.*

*We shall address the issue of containment versus inheritance in the design of a class. Containment exercises the ‘has a’ relationship, in which a class has or contains an object of another class. Inheritance exercises the ‘is a’ relationship, in which a class is a kind of another class. These two differing approaches of containment and inheritance will be illustrated by way of developing Triangle and Tetrahedra classes, where a tetrahedron object can be viewed as a kind of triangle or as having four triangle faces.*

*C++ supports run-time polymorphism using inheritance by declaring base class member functions as virtual. Derived classes can specifically override these base class virtual functions. The use of virtual functions and abstract base classes will be illustrated with the help of a Shapes class hierarchy. Classes such as Polygon, Triangle, Quadrilateral, Circle and Tetrahedra are all derived from an abstract base class called Shape for the modelling of popular primitive geometric objects.*

*Several powerful classes are presented throughout the present chapter. As mentioned above, several Shapes classes are presented. Vector3D, RVector, PtrVector and SortedVector classes are derived from Vector for modelling a vector in three-dimensional space, an indexed range vector, an array of pointer memory addresses and a sorted vector. The Matrix and Windows memory classes presented in Chapter 13 are revisited, and Stack and Queue classes are derived from class LinkedList.*



## 15.1 Base and Derived Classes

Inheritance in C++ is achieved by creating a new or *derived class* from an existing or *base class*, and is introduced in the following program:

```
// b&d.cpp
// introduces base and derived classes
#include <iostream.h> // C++ I/O

// base class
class Base
{
private:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    int Get ()
    { return bdm ; }
    void Set (int i)
    { bdm = i ; }
};

// derived class
class Derived : public Base
{
};

void main ()
{
    cout << "before:" << endl ;

    // base class object
    Base base ;
    cout << "Base::bdm : " << base.Get () << endl ;

    // derived class object
    Derived derived ;
    cout << "Derived::bdm: " << derived.Get () << endl ;

    cout << "after: " << endl ;

    // alter
    derived.Set (1) ;

    cout << "Base::bdm : " << base.Get () << endl ;
    cout << "Derived::bdm: " << derived.Get () << endl ;
}
```

with output:

```

before:
Base::bdm    : 0
Derived::bdm: 0
after:
Base::bdm:   : 0
Derived::bdm: 1

```

The above program declares a base **class**, called, appropriately, **Base**, which encapsulates a **private** data member called **bdm**, a no-argument constructor which initialises **bdm** to zero and two **public** member functions which *get and set* the value of **bdm**. Another **class**, called **Derived**, is declared in the above program, and is **publicly** derived from the base **class**:

```

class Derived : public Base
{
};

```

Notice that **Derived** is an empty **class** that has no data members or member functions of its own. Although the **class** declaration of **Derived** encapsulates no data members or member functions, the following statement within **main()** illustrates that **Derived** has access to the **Base::Get()** member function:

```

//...
cout << "Derived::bdm: " << derived.Get () << endl ;

```

Similarly, **Derived** has access to the **Base::Set()** member function:

```

//...
derived.Set (1) ;

```

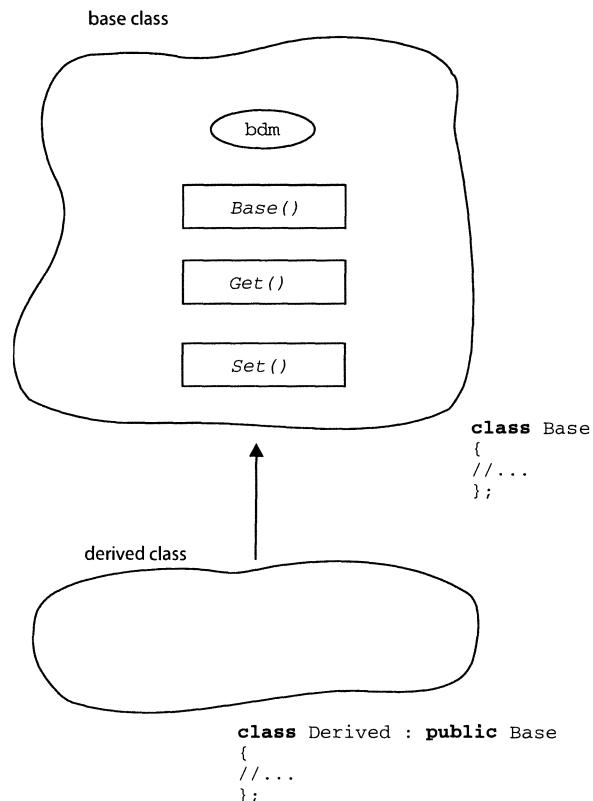
which is in fact assigning the constant value of 1 to the **bdm** data member of the object of **class** **Derived** and not **class** **Base**. This important point is emphasised in the program output, which illustrates that objects **base** and **derived** each have their own **bdm** data members and **derived** does not share the **bdm** data member with object **base**. Objects of **class** **Derived** behave as if we had declared **class** **Derived** as:

```

class Derived
{
private:
    int bdm ;
public:
    Derived ()
        : bdm (0) {}
    int Get ()
        { return bdm ; }
    void Set (int i)
        { bdm = i ; }
};

```

Figure 15.1 schematically illustrates **class** **Derived** inheriting the features of **class** **Base**. Note the direction of the inheritance arrow.



**Fig. 15.1** A derived **class** inheriting the features of a base **class**.

It may sound obvious, but a **class** cannot be derived from itself or from an unknown **class**. It is only possible to derive from a **class** which has previously been declared.

## 15.2 Inheritance by Declaration

Inheritance is specified by incorporating the base **class** in the declaration of the derived **class**. This was demonstrated in the above program for base **class** Base and derived **class** Derived:

```
class Derived : public Base
{
};
```

The general syntax of **class** inheritance is of the form<sup>1</sup>:

```
class DerivedClass : accessSpecifier BaseClass
```

<sup>1</sup> Note that it is illegal to derive a **class** from a C++ integral type such as **int**.

---

```
{
//...
};
```

The use of a single colon (:) and the optional access specifier in the declaration inform the compiler that DerivedClass is derived from BaseClass. The next section examines the different access specifiers available in C++.

## 15.3 Access Specifiers

Program B&D.CPP illustrated the use of the **public** access specifier for **class** Derived:

```
class Derived : public Base
{
};
```

The keyword **public** allows objects of **class** Derived access to Base's **public** member functions, but not access to Base's **private** data members. Alternatively, we can use the **private** keyword as the access specifier, which prevents objects of Derived from accessing Base's **public** member functions, thus preventing a Derived object from accessing any data member or member function of an object of **class** Base.

Note that an object of a base or derived **class** never has access to another **class**'s **private** data members or **private** member functions.

Consider the following program which examines the use of the **public** and **private** access specifiers for derived classes:

```
// acc_spec.cpp
// examines the public and private
// access specifiers for derived classes
#include <iostream.h> // C++ I/O

// base class
class Base
{
private:
    int pri_bdm ;
public:
    int pub_bdm ;
};

// publicly derived class
class PubDerived : public Base
{
private:
    int pri_ddm ;
public:
    void Function ()
    {
        pri_ddm = pri_bdm ; // error: not accessible
    }
}
```

```

        pri_ddm = pub_bdm ; // o.k.: accessible
    }
};

// privately derived class
class PriDerived : private Base
{
private:
    int pri_ddm ;
public:
    void Function ()
    {
        pri_ddm = pri_bdm ; // error: not accessible
        pri_ddm = pub_bdm ; // o.k.: accessible
    }
};

void main ()
{
// publicly derived class object
PubDerived pub_derv ;

pub_derv.pri_bdm ; // error: not accessible
pub_derv.pub_bdm ; // o.k.: accessible

// privately derived class object
PriDerived pri_derv ;

pri_derv.pri_bdm ; // error: not accessible
pri_derv.pub_bdm ; // error: not accessible
}
}

```

Functions of derived classes cannot access the **private** data members of a base **class**, whereas they can access **public** data or function members of a base **class**. Similarly, objects of a derived **class** cannot access the **private** data or function members of a base **class**. Objects of a **publicly** derived **class** can access the **public** data or function members of a base **class**, whereas objects of a **privately** derived **class** cannot access the **public** data or function members of a base **class**.

In addition, we shall see shortly that the keyword **protected** can also be used as an access specifier.

### 15.3.1 Default Specifier

If an access specifier is not used when declaring a derived **class**, then **private** is assumed:

```

class Derived : Base // privately derived class
{
//...
};

```

This is identical in principle to declaring a **class**'s data members, which we know are **private** by default:

```
class X
{
    // private data members
};
```

### 15.3.2 Access Declaration

There are occasions in C++ when a base **class** is **privately** inherited but it is required that certain data members retain their **public** specification. For such cases, C++ supports an *access declaration*, which is appropriately placed in the **public** section of a derived **class**.

The following program declares a base **class**, **Base**, which encapsulates two **public** data members. A **class**, **Derived**, is **privately** derived from **Base**, thus making **Base**'s **public** data members **private** data members of **Derived**:

```
// chng_acc.cpp
// illustrates changing an access specification
// associated with data members of a derived class
#include <iostream.h> // C++ I/O

// base class
class Base
{
public:
    int pub_bdm0 ;
    int pub_bdm1 ;
};

// privately derived class
class Derived : private Base
{
public:
    Base::pub_bdm1 ; // restore public pub_bdm1
};

void main ()
{
    Derived d_obj ;
    d_obj.pub_bdm0 ; // error: not accessible
    d_obj.pub_bdm1 ; // o.k.
}
```

The **public** section of **Derived** contains the following access declaration:

```
Base::pub_bdm1 ;
```

which restores the **public** specification of **Base** **class** data member **pub\_bdm1**. The statements within **main()** illustrate that without the use of the access declaration for

Base::pub\_bdm1 this data member would not be accessible to an object of **class** Derived.

Similarly, an access declaration can be used to restore a **protected** (to be discussed shortly) access specification for a derived **class** data member. Note that it is not possible to modify the original access specification of a base **class** data member using an access declaration, because this would effectively allow a base **class** declaration to be modified by a derived **class**.

When the C++ **namespace** is discussed in Chapter 19 it will be shown that the so-called *using declaration* offers a programmer a more general form of the access declaration, thus making the access declaration redundant.

## 15.4 Properties of Base and Derived Classes

Before examining the **protected** keyword, let's examine three obvious, but nevertheless important, properties of inheritance:

- Inheritance is a one-way process. A base **class** knows nothing of its derived class/es.
- A base **class** remains unchanged during inheritance.
- In theory, an infinite number of classes can be derived from a single base **class**.

The following program illustrates these three properties:

```
// props.cpp
// illustrates certain properties of inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
private:
    int bdm ;
public:
    int BaseGet ()
    { return bdm ; }
    void BFunction ()
    {
        bdm = DerivedGet () ; // error: undefined function
    }
};

// declare multiple derived classes from a single base class:

// derived class
class Derived0 : public Base
{
private:
    int ddm ;
public:
    int DerivedGet ()
```

---

```

    { return ddm ; }
void DFunction ()
{
    ddm = BaseGet () ; // o.k.: accessible
}
};

// derived class
class Derived1 : public Base
{
};

// derived class
class Derived2 : public Base
{
};

void main ()
{
    // base class remains unaltered by derived classes
    Base b_obj ;
}
```

## 15.5 Deriving from a Derived class

Once a derived **class** has been adequately declared, additional classes can be derived from the derived **class**:

```

// der_der.cpp
// illustrates deriving from a derived class
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    int Get ()
        { return bdm ; }
    void Set (int i)
        { bdm = i ; }
};

// derived class from Base
class Derived0 : public Base
{
};
```

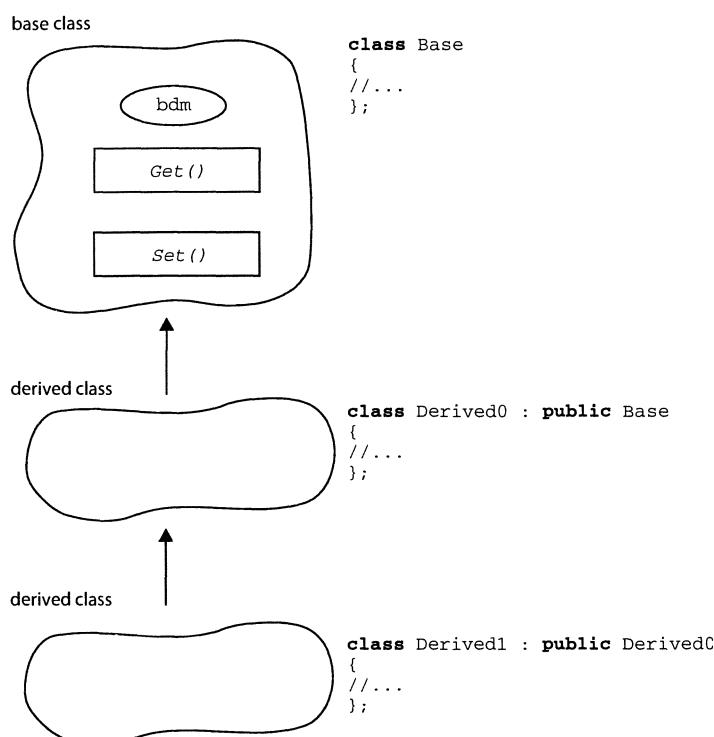
```
// derived class from Derived0
class Derived1 : public Derived0
{
};

void main ()
{
    Base b_obj ;
    b_obj.Set (0) ;
    cout << b_obj.Get () << endl ;

    Derived0 d0_obj ;
    d0_obj.Set (1) ;
    cout << d0_obj.Get () << endl ;

    Derived1 d1_obj ;
    d1_obj.Set (2) ;
    cout << d1_obj.Get () << endl ;
}
```

The above hierarchy is schematically illustrated in Fig. 15.2.



**Fig. 15.2** Deriving from a derived **class**.

## 15.6 protected Data Members

To date, all of the classes that we have seen have declared their data members and member functions as either **private** or **public**. C++ supports a third keyword, called **protected**, to declare members of a **class** when programming for inheritance. Declaring a data or function member of a **class** as **protected** allows the member to be accessed not only by other data or function members of the same **class**, but also by any **class** derived from it. Objects of a **class** do not have access to a **class's** **protected** members, just as if they were declared as **private**. Consider the program:

```
// proc.cpp
// illustrates the protected keyword
#include <iostream.h> // C++ I/O

// base class
class Base
{
private:
    int pri_bdm ;
protected:
    int pro_bdm ;
public:
    int pub_bdm ;
};

// derived class
class Derived : public Base
{
private:
    int pri_ddm ;
public:
    void Function ()
    {
        pri_ddm = pro_bdm ; // o.k.: accessible
    }
};

void main ()
{
    Base bobj ;

    bobj.pro_bdm ; // error: not accessible

    Derived dobj ;

    dobj.pro_bdm ; // error: not accessible
}
```

The **protected** data member of **class** **Base**, **pro\_bdm**, is accessible to **class** **Derived**, but not accessible to objects of **Base** or **Derived**.

## 15.7 protected Access Specifier

We saw in the previous section that the **private** and **public** keywords can be used as access specifiers. Also, the **protected** keyword can be used as an access specifier for derived classes. When a **class** is **protectedly** derived from a base **class**, all **protected** and **public** data or function members of the base **class** are **protected** members of the derived **class**:

```
// pro_accs.cpp
// examines the protected access specifier for
// derived classes
#include <iostream.h> // C++ I/O

// base class
class Base
{
private:
    int pri_bdm ;
protected:
    int pro_bdm ;
public:
    int pub_bdm ;
};

// protectedly derived class
class ProDerived : protected Base
{
private:
    int pri_ddm ;
public:
    void Function ()
    {
        pri_ddm = pri_bdm ; // error: not accessible
        pri_ddm = pro_bdm ; // o.k.: accessible
        pri_ddm = pub_bdm ; // o.k.: accessible
    }
};

void main ()
{
    // protectedly derived class object
    ProDerived pro_derv ;

    pro_derv.pri_bdm ; // error: not accessible
    pro_derv.pro_bdm ; // error: not accessible
    pro_derv.pub_bdm ; // error: not accessible
}
```

Both the **protected** and **public** data members, **pro\_bdm** and **pub\_bdm**, of the base **class**, **Base**, are **protected** data members of the derived **class**, **ProDerived**, thus preventing objects of **ProDerived** having access to **Base**'s **public** data member.

## 15.8 Base and Derived class Constructors and Destructors

### 15.8.1 No Derived class Constructors or Destructor

Several of the derived classes in the above programs did not explicitly define their own constructor or destructor. To examine exactly what initialisation takes place for an object of a derived **class** without a constructor or destructor explicitly defined, consider the program:

```
// no_d_c&d.cpp
// illustrates a derived class
// with no constructor or destructor
#include <iostream.h> // C++ I/O

// base class
class Base
{
public:
    Base ()
    { cout << "Base class constructor called"
      << endl ; }
    ~Base ()
    { cout << "Base class destructor called"
      << endl ; }
};

// derived class
class Derived : public Base
{
};

void main ()
{
    Derived d_obj ;
}
```

with output:

```
Base class constructor called
Base class destructor called
```

The output illustrates that if a constructor or destructor is not defined for **class** Derived the compiler will use the appropriate constructor or destructor of **class** Base.

### 15.8.2 Derived class Constructors and Destructor

Let's now extend the above program by defining a constructor and destructor for **class** Derived:

```
// d_c&d.cpp
// illustrates a derived class
```

```

// with constructor and destructor
#include <iostream.h> // C++ I/O

// base class
class Base
{
public:
    Base ()
    { cout << "Base class constructor called"
      << endl ; }
    ~Base ()
    { cout << "Base class destructor called"
      << endl ; }
};

// derived class
class Derived : public Base
{
public:
    Derived ()
    { cout << "Derived class constructor called"
      << endl ; }
    ~Derived ()
    { cout << "Derived class destructor called"
      << endl ; }
};

void main ()
{
    Derived d_obj ;
}

```

with output:

```

Base class constructor called
Derived class constructor called
Derived class destructor called
Base class destructor called

```

The program output illustrates that although a constructor is defined for **class** Derived, the constructor for **class** Base is called, and before Derived's constructor is called, when an object of Derived is defined. It makes sense to call a base **class** constructor before a derived **class** constructor, because generally the correct operation of a derived **class** object will depend on the data members of the base **class** being correctly initialised. As expected, the destructors are called in reverse order, i.e. the derived **class** destructor is called before the base **class** destructor.

Let's extend the above D\_C&D.CPP by declaring a data member of a given **class** in the derived **class**:

```

// d_c&d&d.cpp
// illustrates a derived class

```

```
// with constructor, destructor and data members
#include <iostream.h> // C++ I/O

class X
{
public:
    X ()
    { cout << "X class constructor called"
      << endl ; }
    ~X ()
    { cout << "X class destructor called"
      << endl ; }
};

// base class
class Base
{
public:
    Base ()
    { cout << "Base class constructor called"
      << endl ; }
    ~Base ()
    { cout << "Base class destructor called"
      << endl ; }
};

// derived class
class Derived : public Base
{
protected:
    X x ;
public:
    Derived ()
    { cout << "Derived class constructor called"
      << endl ; }
    ~Derived ()
    { cout << "Derived class destructor called"
      << endl ; }
};

void main ()
{
    Derived d_obj ;
```

with output:

```
Base class constructor called
X class constructor called
Derived class constructor called
Derived class destructor called
```

```
X class destructor called
Base class destructor called
```

The program output illustrates that if data members are declared for a derived **class**, the order in which constructors are called when an object of a derived **class** is defined is: base, data member and derived. The order is reversed for **class** destructors.

Even if a derived **class** constructor has a different number of arguments than its base **class** constructor, the above rule of calling base/derived constructors/destructor still applies:

```
// odd_c&d.cpp
// further illustrates a derived class
// with constructor and destructor
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0)
        { cout << "Base class constructor called"
          << endl ; }
    ~Base ()
        { cout << "Base class destructor called"
          << endl ; }
};

// derived class
class Derived : public Base
{
private:
    int ddm ;
public:
    Derived (int i)
        : ddm (i)
        { cout << "Derived class constructor called"
          << endl ; }
    ~Derived ()
        { cout << "Derived class destructor called"
          << endl ; }
};

void main ()
{
//Derived d_obj ; // error: no match found

    Derived d_obj (1) ;
}
```

---

with output identical to that generated by program D\_C&D.CPP:

```
Base class constructor called
Derived class constructor called
Derived class destructor called
Base class destructor called
```

Note that an object of Derived can only be defined using the one-argument constructor of Derived. The compiler will not use the no-argument constructor of Base. Also, if the default constructor of **class** Base is removed a compilation error will result, because objects of the base **class** Base cannot be initialised correctly when an object of Derived is defined.

Alternatively, what if we wanted to define an object of **class** Derived and at the same time initialise an inherited data member of **class** Base? For example, let's modify program ODD\_C&D.CPP to define a Derived object and simultaneously initialise the inherited **protected** data member, bdm, of **class** Base via the none and one-argument constructors:

```
// d_to_b.cpp
// illustrates passing constructor argument objects
// from a derived class constructor to a base class
// constructor

#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    ~Base () {}
    int Get ()
    { return bdm ; }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () {}
    Derived (int i) : Base (i), ddm (i) {}
    ~Derived () {}
};

void main ()
```

```

{
Derived no_arg_obj ;
Derived one_arg_obj (1) ;

cout << "no_arg_obj.bdm : " << no_arg_obj.Get () << endl
    << "one_arg_obj.bdm: " << one_arg_obj.Get () ;
}

```

with output:

```

no_arg_obj.bdm : 0
one_arg_obj.bdm: 1

```

The above program illustrates that C++ allows a derived **class** constructor to call its base **class** constructor and simultaneously to pass derived **class** constructor argument objects to its base **class** constructor using the general syntax:

```

DerivedClass (argument_list)
: BaseClass (argument_list)
{
//...
}

```

Note that if a non-zero-argument base **class** constructor is called by a derived **class** constructor, the default constructor of the base **class** is no longer called.

This syntax is identical to the data member initialisation list syntax that we have seen in previous chapters:

```

ClassName (typeSpecifier arg1, ..., typeSpecifier argn)
: data_member1 (arg1), ..., data_membern (argn)
{
//...
}

```

Hence a base **class** constructor acts just as if it is a data member of the derived **class** when initialising an object of a derived **class**.

Both of the constructor bodies for **class** Derived in program D\_TO\_B.CPP are empty:

```

class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () {}
    Derived (int i) : Base (i), ddm (i) {}
    //...
};

```

As well as passing constructor arguments to a base **class**, a derived **class** can also include statements, function calls and expressions involving the passed constructor arguments in the constructor function body:

```

// d_to_b1.cpp
// further illustrates passing constructor argument objects
// from a derived class constructor to a base class
// constructor

#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    ~Base () {}
    int Get ()
    { return bdm ; }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () {}
    Derived (int i) : Base (i), ddm (i)
        { cout << "der. con. arg. sq.: " << i*i << endl ; }
    ~Derived () {}
};

void main ()
{
    Derived one_arg_obj (2) ;
    cout << "one_arg_obj.bdm: " << one_arg_obj.Get () << endl ;
}

```

with output:

```

der. con. arg. sq.: 4
one_arg_obj.bdm: 2

```

### 15.8.3 Use of the **xalloc** and **xmsg** Classes with the C++ string Library **class**

As a further illustration of a derived **class** calling and passing argument objects to a base **class** constructor, consider the C++ string library **class** declared in the header file CSTRING.H:

```
class string
{
//...
public:
    class outofrange: public xmmsg
    {
public:
    outofrange () ;
};

class lengtherror : public xmmsg
{
public:
    lengtherror () ;
};

//...
string () throw (xalloc) ;
string (const string& s) throw (xalloc) ;
//...
~string () throw () ;
//...

};

//...
inline string::outofrange::outofrange ()
: xmmsg ("string reference out of range") ;
//...
inline string::lengtherror::lengtherror ()
: xmmsg ("string length error") ;
//...
```

Note the use of the exception specifications for the above-listed `string` constructors and destructor. The `class string` destructor throws no exceptions.

Within `class string` are two nested classes, `outofrange` and `lengtherror`, which both inherit from the C++ `xmmsg` library `class`<sup>2</sup>. The no-argument constructor definitions of both classes, `outofrange` and `lengtherror` initialise the one-argument constructor of `class xmmsg` to appropriate string messages.

## 15.9 Copy Constructors, Overloaded Assignment Operator and Inheritance

### 15.9.1 Copy Constructors

If a base `class` defines a copy constructor but a derived `class` does not, then the base `class` copy constructor is invoked when an object of the derived `class` is defined and initialised (either by assignment or copy constructor) to another object of the derived `class`. This is demonstrated in the following program:

---

2 For a discussion of the exception handling classes `xmmsg` and `xalloc` refer to Chapter 14. Classes `xmmsg` and `xalloc` are declared in the C++ library header file `EXCEPT.H`.

```

// cc_inh.cpp
// illustrates a base class copy constructor and
// inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0)
    { cout << "Base::Base() called" << endl ; }
    // copy constructor
    Base (const Base& b)
        : bdm (b.bdm)
    { cout << "Base::Base(const Base&) called" << endl ; }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived ()
        : ddm (0)
    { cout << "Derived::Derived() called" << endl ; }
};

void main ()
{
    Derived d0 ;      // define
    Derived d1 = d0 ; // define and initialise by assignment
    //Derived d1 (d0) ; // define and initialise by copy con.
}

```

with output:

```

Base::Base() called
Derived::Derived() called
Base::Base(const Base&) called

```

The first two lines displayed are a result of defining object `d0`. The final line displayed is due to defining and initialising object `d1` to object `d0`. The data members of both the base and derived classes are initialised by memberwise assignment.

If a derived **class** defines a copy constructor, it will be invoked when a derived **class** object is defined and initialised (either by assignment or copy constructor) to another object of the derived **class**. This is demonstrated in the program below, which simply adds a copy constructor to **class** `Derived`:

```

// cc_inh1.cpp
// illustrates the use of both base and derived class
// copy constructors and inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
//...
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived ()
        : ddm ()
    { cout << "Derived::Derived() called" << endl ; }
    // copy constructor
    Derived (const Derived& d)
        : Base (d), ddm (d.ddm)
    { cout << "Derived::Derived(const Derived&) called"
        << endl ; }
};

void main ()
{
    Derived d0 ;           // define
    Derived d1 = d0 ;      // define and initialise by assignment
    //Derived d1 (d0) ;    // define and initialise by copy con.
}

```

with output:

```

Base::Base() called
Derived::Derived() called
Base::Base(const Base&) called
Derived::Derived(const Derived&) called

```

Note that the Derived copy constructor invokes the Base copy constructor, since it is defined and it is the responsibility of a derived **class** copy constructor to ensure that data members of a base **class** are appropriately initialised:

```

Derived (const Derived& d)
    : Base (d), ddm (d.ddm) { /*...*/ }

```

The program output illustrates that, as with normal constructors, a base **class** copy constructor is invoked before a derived **class** copy constructor. If the base **class** copy constructor

is not defined but the derived **class** copy constructor is defined, the compiler will perform the necessary default memberwise initialisation of the base **class** data members.

### 15.9.2 Overloaded Assignment Operator

The operation of the overloaded assignment **operator=()** function is similar to the operation of the copy constructor:

```
// op_inh.cpp
// illustrates the use of a base class
// overloaded assignment operator and inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    // constructors
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    Base& operator = (const Base& b)
    {
        cout << "Base::operator =() called" << endl ;
        bdm = b.bdm ; return *this ;
    }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
    void Display ()
    {
        cout << "Derived::bdm: " << bdm
            << ", Derived::ddm: " << ddm << endl ; }
};

void main ()
{
    Base b1 (1), b2 (2) ;
    b2 = b1 ;
```

```

Derived d1 (1), d2 (2) ;
d2 = d1 ;

d2.Display () ;
}

```

with output:

```

Base::operator=() called
Base::operator=() called
Derived::bdm: 1, Derived::ddm: 1

```

The output illustrates that if a base **class** defines an **operator=()** function, but a derived **class** does not define **operator=()**, the base **class** **operator=()** function is invoked when one derived **class** object is assigned to another derived **class** object. The output also illustrates that the default assignment of objects of a derived **class** which does not overload **operator=()** is memberwise assignment.

If a derived **class** also defines the **operator=()** function, then it is invoked when one derived **class** object is assigned to another. This is illustrated in the following program:

```

// op_inh1.cpp
// illustrates the use of both base and derived class
// overloaded assignment operator and inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
//...
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
    Derived& operator = (const Derived& d)
    {
        cout << "Derived::operator=() called" << endl ;
        Base::operator = (d) ;
        ddm = d.ddm ; return *this ;
    }
};

void main ()
{
    Base b1 (1), b2 (2) ;

```

---

```
b2 = b1 ;

Derived d1 (1), d2 (2) ;
d2 = d1 ;
}
```

with output:

```
Base::operator=() called
Derived::operator=() called
Base::operator=() called
```

Note that the Derived **operator=()** function invokes the Base **operator=()** function, since it is defined and it is the responsibility of a derived **class operator=()** function to ensure that data members of a base **class** are appropriately assigned:

```
Derived& operator = (const Derived& d)
{
    Base::operator = (d) ;
    ddm = d.ddm ; return *this ;
}
```

When both a base and a derived **class** define **operator=()** functions, and an object of a derived **class** is assigned to an object of a base **class**, the **operator=()** function of the **class** on the left-hand side of the assignment operator is called:

```
// op_inh2.cpp
// illustrates base and derived class
// overloaded assignment operator functions and inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
//...
Base& operator = (const Base& b) { /*...*/ }
};

// derived class
class Derived : public Base
{
//...
Derived& operator = (const Derived& d) { /*...*/ }
};

void main ()
{
    Base    b (0) ;
    Derived d (1) ;

    b = d ; // o.k. : assign derived object to base object
```

---

```
//d = b ; // error: assign base object to derived object
}
```

with output:

```
Base::operator=() called
```

Assigning a derived **class** object to a base **class** object is often referred to as *slicing*. Note that the reverse situation of assigning a base **class** object to a derived **class** object is illegal.

Let us conclude this section by examining a program in which both base and derived classes define a copy constructor and an **operator=()** function:

```
// op_inh3.cpp
// illustrates the use of a copy constructor and overloaded
// assignment operator function for both base and derived
// classes
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    // constructors
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    Base (const Base& b)
        : bdm (b.bdm)
    { cout << "Base::Base(const Base&) called"
        << endl ; }
    Base& operator = (const Base& b)
    {
        cout << "Base::operator=() called" << endl ;
        bdm = b.bdm ; return *this ;
    }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
    Derived (const Derived& d)
        : Base (d), ddm (d.ddm)
```

```

    { cout << "Derived::Derived(const Derived&) called"
      << endl ; }
    Derived& operator = (const Derived& d)
    {
      cout << "Derived::operator=() called" << endl ;
      Base::operator = (d) ;
      ddm = d.ddm ; return *this ;
    }
};

void main ()
{
  Derived d0 (0) ;

  cout << "Derived d1=d0:" << endl ;
  Derived d1 = d0 ;

  cout << "d1=d0:" << endl ;
  d1 = d0 ;
}

```

with output:

```

Derived d1=d0:
Base::Base(const Base&) called
Derived::Derived(const Derived&) called
d1=d0:
Derived::operator=() called
Base::operator=() called

```

Pay particular attention to the distinction between the use of the assignment operator for initialisation of an object and assignment of one object to another.

## 15.10 new and delete Operators and Inheritance

If the **new** and **delete** operators are overloaded for a base **class**, then a derived **class** inherits the overloaded **new** and **delete** operators. This is illustrated in the following program:

```

// nd_inh.cpp
// illustrates overloading the new and delete operators
// with respect to inheritance
#include <iostream.h> // C++ I/O
#include <alloc.h> // malloc(), free()

// base class
class Base
{

```

```

protected:
    int bdm ;
public:
    // constructors
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    void* operator new (size_t size)
    {
        cout << "Base::operator new() called, size: "
        << size << endl ;
        return malloc (size) ;
    }
    void operator delete (void* ptr)
    {
        cout << "Base::operator delete() called" << endl ;
        free (ptr) ;
    }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
};

void main ()
{
    Base* pb (new Base) ;
    Derived* pd (new Derived) ;

    delete pb ;
    delete pd ;
}

```

with output:

```

Base::operator new() called, size: 2
Base::operator new() called, size: 4
Base::operator delete() called
Base::operator delete() called

```

The interesting point to note is that the size of a Derived object is larger than the size of a Base object, and yet the **new** and **delete** operators allocate and deallocate the memory appropriately. If **new** and **delete** are overloaded for a base **class**, correct allocation and deallocation is inherited for all derived classes.

## 15.11 Overriding class Member Functions

Let's now examine base and derived classes that have member functions with identical function names:

```
// override.cpp
// illustrates derived class member functions
// which override base class member functions
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    ~Base () {}
    void Display ()
    {
        cout << "Base::bdm: " << bdm << endl ;
    }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
    ~Derived () {}
    void Display ()
    {
        cout << "Derived::bdm: " << bdm << endl
            << "Derived::ddm: " << ddm << endl ;
    }
};

void main ()
{
    Base     b_obj ;
    Derived d_obj ;

    b_obj.Display () ;
    d_obj.Display () ;
}
```

with output:

```
Base::bdm: 0
Derived::bdm: 0
Derived::ddm: 0
```

Both **Base** and **Derived** classes define a member function called *Display()*, which simply displays each **class's protected** data members. The output illustrates that for objects of a derived **class** the compiler will choose the member function of the derived **class** if the same function name exists in both base and derived classes. For objects of a base **class**, the compiler has no other option than to choose the member defined in the base **class**, since the base **class** knows nothing of the derived **class**. The *Display()* member function in **class Derived** is said to *override* the *Display()* member function in **class Base**. This is a convenient and frequently used property of derived classes, because it allows the same function name to be used for objects of base and derived classes.

What about overriding base **class** overloaded member functions? Consider the following program, which overloads the *Display()* member function in **class Base**:

```
// oo.cpp
// illustrates overriding base class overloaded member
// functions
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    ~Base () {}
    void Display ()
    {
        cout << "Base::bdm: " << bdm << endl ;
    }
    void Display (char* str)
    {
        cout << str << bdm << endl ;
    }
};

// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
```

```

Derived (int i) : Base (i), ddm (i) {}
~Derived () {}
void Display ()
{
    cout << "Derived::bdm: " << bdm << endl
    << "Derived::ddm: " << ddm << endl ;
}
};

void main ()
{
Base     b_obj ;
Derived  d_obj ;

b_obj.Display () ;           // Base::Display()
b_obj.Display ("Base::bdm: ") ; // Base::Display(char*)
d_obj.Display () ;
d_obj.Display ("Derived::bdm: ") ; // error: undefined
                                    // function
}

```

Objects of **class** `Base` have access to both of the overloaded `Display()` member functions. However, objects of **class** `Derived` do not have access to `Base::Display(char*)`, and the compiler will issue an ‘undefined function’ compilation error if an attempt is made to call this member function via a `Derived` **class** object.

The next program illustrates calling a base **class** overridden member function using the scope resolution operator (`::`):

```

// scope_res.cpp
// illustrates the use of the scope resolution operator to
// access an overridden base class member function from a
// derived class
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    ~Base () {}
    void Display ()
    {
        cout << "Base::bdm: " << bdm << endl ;
    }
};

```

```
// derived class
class Derived : public Base
{
protected:
    int ddm ;
public:
    Derived () : Base () { ddm = 0 ; }
    Derived (int i) : Base (i), ddm (i) {}
    ~Derived () {}
    void Display ()
    {
        Base::Display () ;
        cout << "Derived::ddm: " << ddm << endl ;
    }
};

void main ()
{
    Derived d_obj ;

    d_obj.Display () ;
}
```

with output:

```
Base::bdm: 0
Derived::ddm: 0
```

Rather than defining `Derived::Display()` as:

```
void Display ()
{
    cout << "Derived::bdm: " << bdm << endl
    << "Derived::ddm: " << ddm << endl ;
}
```

as was the case in program OVERRIDE.CPP, SCPE\_RES.CPP makes use of the scope resolution operator to call `Base::Display()` from within `Derived::Display()`:

```
void Display ()
{
    Base::Display () ;
    cout << "Derived::ddm: " << ddm << endl ;
}
```

If the `Base` **class** name and the scope resolution operator were not used, the compiler would view the call to `Display()` within `Derived::Display()` as a call to itself, and thus generate an infinite recursive loop.

## 15.12 Overloaded Operator Functions and Inheritance

Except for the `operator=()` function, all other overloaded operator functions of a base `class` are inherited by a derived `class`.

## 15.13 Friendship and Inheritance

Class friendship is not inherited:

```
// fr_inh.cpp
// illustrates friendship and inheritance
#include <iostream.h> // C++ I/O

class X
{
private:
    int x_data ;
public:
    // friend
    friend class Y ;
};

// friend of X
class Y
{
public:
    void YFunction (X x)
    {
        x.x_data++ ; // o.k.
    }
};

// Z derived from Y
class Z : public Y
{
public:
    void ZFunction (X x)
    {
        x.x_data++ ; // error: X::x_data not accessible
    }
};

void main ()
{}
```

The `class Y` is a `friend` of `class X` and therefore has access to the `private` data member `X::x_data`. However, just because `class Z` is derived from `class Y` it does not similarly

allow members of **class** Z access to the **private** data member of **class** X. Class friendship is not inherited.

A **friend** function of a derived **class** has access to a **protected static** data member of a base **class**. In the case of non-**static** data members, a derived **class friend** function only has access to a non-**static** data member of a base **class** via an object, pointer or reference to the derived **class**:

```
// fr_inh1.cpp
// further illustrates friendship and inheritance
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int ns_bdm ;
    static int s_bdm ;
};

// derived class
class Derived : public Base
{
public:
    friend void Function0 (Base vb, Base* pb, Base& rb)
    {
        vb.s_bdm++ ; // o.k.

        vb.ns_bdm++ ; // error: not accessible
        pb->ns_bdm++ ; // error: not accessible
        rb.ns_bdm++ ; // error: not accessible
    }
    friend void Function1 (Derived vd, Derived* pd,
                           Derived& rd)
    {
        vd.ns_bdm++ ; // o.k.
        pd->ns_bdm++ ; // o.k.
        rd.ns_bdm++ ; // o.k.
    }
};

void main ()
{
}
```

## 15.14 Increased Functionality and Code Reusability

Let's now examine how an existing **class** can be derived from to create a related **class** which has increased functionality. The base **class** that we will use is the **Point class** discussed

in previous chapters, and is basically identical to previous versions of `Point` except that the `x`, `y` and `z` data members are now declared **protected**:

```
// point.h
// class Point

#ifndef _POINT_H // prevent multiple includes
#define _POINT_H

#include <iostream.h> // C++ I/O

// class Point
class Point
{
protected:
    // protected data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (double d)
        : x (d), y(d), z (d) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    double& X () { return x; } // non-const objects
    double& Y () { return y; }
    double& Z () { return z; }
    const double& X () const { return x; } // const objects
    const double& Y () const { return y; }
    const double& Z () const { return z; }
    // overloaded operator
    Point& operator = (const Point& p);
    // friends
    friend ostream& operator << (ostream& s,
                                    const Point& p);
    friend istream& operator >> (istream& s, Point& p);
}; // class Point

#endif // _POINT_H
```

The implementation file for `class Point` is:

```
// point.cpp
// implementation file for class Point
#include "point.h" // header file
```

```

// overloaded operator:

// assignment operator =
inline Point& Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}

// overloaded operators/friends:

// insertion operator <<
ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
}

// extraction operator >>
istream& operator >> (istream& s, Point& p)
{
    return s >> p.x >> p.y >> p.z ;
}

```

The above Point **class** is a non-**template** and non-exception-handling **class** in order to help focus our attention on design considerations relevant to inheritance. The following program extends the Point **class** by **publicly** deriving from it a NumberedPoint **class**:

```

// num_pt.cpp
// a numbered Point class
#include <iostream.h> // C++ I/O
#include "point.h" // Point class

// class NumberedPoint
class NumberedPoint : public Point
{
protected:
    // protected data member
    int number ;

public:
    // constructors
    NumberedPoint ()
        : Point (), number (0) {}
    NumberedPoint (double x_arg, double y_arg,
                  double z_arg=0.0, int n=0)
        : Point (x_arg, y_arg, z_arg), number (n) {}
    NumberedPoint (double d, int n=0)
        : Point (d), number (n) {}
    // copy constructor
    NumberedPoint (const NumberedPoint& p)
        : Point (p), number (p.number) {}

```

```

// member functions
// non-const objects
int& Number () { return number ; }
// const objects
const int& Number () const { return number ; }
// overloaded operator
NumberedPoint& operator = (const NumberedPoint& p) ;
// friends
friend ostream& operator << (ostream& s,
                                const NumberedPoint& p) ;
friend istream& operator >> (istream& s,
                                NumberedPoint& p) ;
}; // class NumberedPoint

// overloaded operator:

// assignment operator =
inline NumberedPoint& NumberedPoint::operator =
                                (const NumberedPoint& p)
{
    Point::operator = (p) ;
    number = p.number ;
    return *this ;
}

// overloaded operators/friends:

// insertion operator <<
inline ostream& operator << (ostream& s,
                                const NumberedPoint& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z
                << ", " << p.number << ")" ;
}

// extraction operator >>
inline istream& operator >> (istream& s, NumberedPoint& p)
{
    return s >> p.x >> p.y >> p.z >> p.number ;
}

void main ()
{
    Point p (1.0, 2.0) ;
    cout << "p: " << p << endl ;

    NumberedPoint np ;
    cout << "np: " << np << endl ;

    NumberedPoint nq (3.0, 4.0, 0.0, 1) ;
    cout << "nq: " << nq << endl ;
}

```

```

cout << "nq.X(): " << nq.X () << endl ;
cout << "nq.Number (): " << nq.Number () << endl ;
}

```

with output:

```

p: (1, 2, 0)
np: (0, 0, 0, 0)
nq: (3, 4, 0, 1)
nq.X (): 3
nq.Number (): 1

```

`NumberedPoint` adds further functionality to `Point` by an additional **protected** data member called `number`, which assigns an integer number to a given `Point`. The `number` data member is declared **protected** to allow further classes to be derived from `NumberedPoint` while maintaining data protection. `NumberedPoint` defines similar constructors, including a copy constructor, to `Point`. For instance:

```

//...
NumberedPoint ()
: Point (), number (0) {}

```

and the copy constructor of `NumberedPoint` invokes the `Point` copy constructor and initialises the `number` data member:

```

//...
NumberedPoint (const NumberedPoint& p)
: Point (p), number (p.number) {}

```

Note that, when initialising the data members of a derived **class**, a base **class** constructor acts just as if it is a data member of the derived **class**. `NumberedPoint` adds a single access member function called `Number ()` for both constant and non-constant objects which simply returns the value of the `number` data member. The three overloaded operator functions `=`, `<<` and `>>` of **class** `Point` must be redefined for **class** `NumberedPoint`. The **operator= ()** function is overloaded for **class** `NumberedPoint` as:

```

inline NumberedPoint& NumberedPoint::operator =
(const NumberedPoint& p)
{
    Point::operator = (p) ;
    number = p.number ;
    return *this ;
}

```

which first calls the function `Point::operator= ()`, passing a `NumberedPoint` object argument, then assigns object `p`'s `number` data member to `number`, and finally returns a reference to the object operated on by the `operator= ()` function. Also, the overloaded functions `operator<< ()` and `operator>> ()` are **friend** functions, and as a result must be redefined for **class** `NumberedPoint`, since they are not inherited.

You may be thinking that developing the derived **class** `NumberedPoint` has generated a great deal of extra unnecessary work when we could have simply added the `number` data

---

member to **class** Point, plus the additional modifications to Point's constructors, member functions and overloaded **operator()** functions:

```
class Point
{
private:
    // private data members
    double x, y, z ;
    int number ;

public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0), number (0) {}
    //...
};
```

This is not always possible, and in certain cases is inadvisable. The Point **class** may have been supplied as part of a **class** library for which we do not have access to the implementation files. In modifying the Point **class** we lose the original structure of the **class** and may in the process of modification damage the original **class**. Also, there is a knock-on affect associated with modifying **class** Point, in that any previously developed classes which use the original Point **class** would have to be modified. These issues help to emphasise the key advantages of inheritance and of developing derived classes rather than modifying existing classes. Inheritance permits *code reusability* by allowing a programmer to add increased functionality to a **class** by reusing existing classes and at the same time guaranteeing that existing code remains unaltered. Aside from code reusability, inheritance assists a programmer in the conceptualisation of a problem and in structuring the design of a program. General low-level base classes can be developed which allow derived classes to add more and more functionality, through specialisation, to base classes.

## 15.15 A Vector **class** for Three-dimensional Space

Let us continue our examination of inheritance by developing a Vector3D **class** which is derived from **class** Vector. A Vector **class** has been discussed in some detail in previous chapters, with reference to the topics of pointers, templates and exception handling. However, the Vector **class** listed in the present section is a non-**template** and non-exception-handling **class** based on the **class** declared in VECTOR.H in Chapter 12. A non-**template**/exception-handling approach is adopted to help focus our attention on inheritance. Template classes RVector (range vectors) and SortedVector (sorted vectors) are presented towards the end of the chapter, when templates and inheritance are discussed.

Class Vector encapsulates an array of n\_elements **double** floating-point numbers (array[0],array[1],...,array[n\_elements-1]):

```
// vec.h
// header file for Vector class

#ifndef _VEC_H // prevent multiple includes
#define _VEC_H
```

```

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...
#include <assert.h> // assert()

class Vector
{
protected:
    // data members
    double* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector () { Allocate (3) ; }
    Vector (int n) { Allocate (n) ; }
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                  n_elements = 0 ; }
    // member functions
    int NumberElements () const
    { return n_elements ; }
    double& Value (int index) ;
    const double& Value (int index) const ;
    void New (int new_n) ;
    // overloaded operators
    Vector& operator = (const Vector& v) ;
    double& operator [] (int index) ;
    const double& operator [] (int index) const ;
    Vector operator + (const Vector& v) ;
    Vector operator - (const Vector& v) ;
    Vector operator += (const Vector& v) ;
    Vector operator -= (const Vector& v) ;
    // friend
    friend ostream& operator << (ostream& s,
                                       const Vector& v) ;
}; // class Vector

#endif // _VEC_H

```

with member function definitions given in VEC.CPP. Note that Vector's data members are declared **protected** so as to make Vector *ready for inheritance*.

Although **class** Vector allows us to work in n\_elements-dimensional space, we are frequently interested in using pairs or triples of numbers to locate points in a plane or three-dimensional space when performing geometric modelling. Hence a **class** Vector3D is **publicly** derived from **class** Vector which specifically caters for points in two- and three-dimensional space:

```
// vec3d.h
```

```
// header file class Vector3D

#ifndef _VEC3D_H // prevent multiple includes
#define _VEC3D_H

#include <math.h> // sqrt()
#include "vec.h" // class Vector

class Vector3D : public Vector
{
public:
    // constructors
    Vector3D ()
        : Vector () {}
    Vector3D (double arg1, double arg2, double arg3=0.0)
        : Vector ()
        { array[0]=arg1; array[1]=arg2; array[2]=arg3; }
    // copy constructor
    Vector3D (const Vector3D& v)
        : Vector (v) {}
    // member functions
    double Norm () ;
    void Normalise () ;
    double DotProduct (const Vector3D& v) ;
    Vector3D CrossProduct (const Vector3D& v) ;
}; // class Vector3D

#endif // _VEC3D_H
```

with member function definitions given in VEC3D.CPP:

```
// vec3d.cpp
// implementation file for Vector3D class

#include "vec3d.h"

const double TOLERANCE = 1e-06 ;

// member functions:

// returns the norm or magnitude (i.e. length) of a vector
double Vector3D::Norm ()
{
    return sqrt ( array[0]*array[0] +
                  array[1]*array[1] +
                  array[2]*array[2] ) ;
}

// returns a normalised vector
void Vector3D::Normalise ()
{
```

```

double n = this->Vector3D::Norm () ;
assert (n > TOLERANCE) ;
array[0]/=n ; array[1]/=n ; array[2]/=n ;
}

// returns the dot product of two vectors
double Vector3D::DotProduct (const Vector3D& v)
{
return ( array[0]*v.array[0] +
array[1]*v.array[1] +
array[2]*v.array[2] ) ;
}

// returns the cross product of two vectors (right-hand
// screw rule)
Vector3D Vector3D::CrossProduct (const Vector3D& v)
{
return Vector3D ( array[1]*v.array[2] - array[2]*v.array[1],
array[2]*v.array[0] - array[0]*v.array[2],
array[0]*v.array[1] -
array[1]*v.array[0] ) ;
}

```

Vector3D adds no new data members to Vector, only member functions. Both of Vector3D's constructors call the default Vector constructor, which encapsulates a vector with three components. The Vector3D copy constructor simply calls the Vector copy constructor, since Vector3D does not add additional data members to Vector:

```

//...
Vector3D (const Vector3D& v)
: Vector (v) {}

```

In addition, Vector3D does not define a destructor but calls, by default, the Vector destructor.

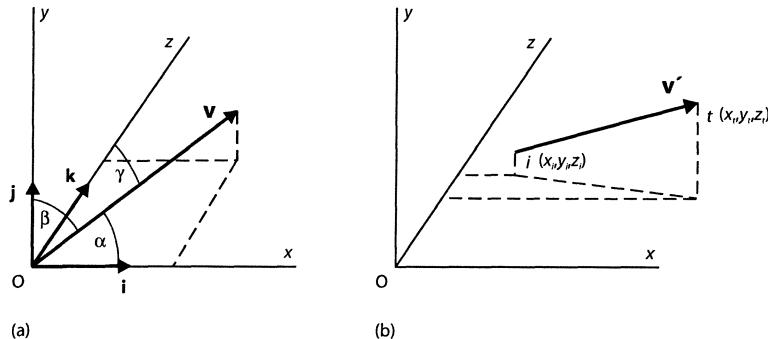
Note that the overloaded functions **operator=()**, **operator+()**, **operator-()** and **operator<<()** are not specifically overridden for **class** Vector3D, since Vector3D adds no new data members to **class** Vector.

Vector3D defines four additional member functions, called *Norm()*, *Normalise()*, *DotProduct()* and *CrossProduct()*, for manipulating Vector3D objects. To understand the workings of these functions let's first examine an arbitrary vector  $\mathbf{v}(a, b, c)$  in a three-dimensional space shown in Fig. 15.3(a). Constants  $a, b$  and  $c$  are the scalar components of  $\mathbf{v}$ . Figure 15.3(a) shows the initial point of  $\mathbf{v}$  to be at the origin O of the Cartesian coordinate system  $(x, y, z)$ , although the initial point of a vector can, in general, be positioned arbitrarily: Fig. 15.3(b).

The length or *norm* of a vector  $\mathbf{v}(a, b, c)$  follows from Pythagoras' theorem and is given by:

$$\|\mathbf{v}\| = (a^2 + b^2 + c^2)^{1/2}$$

Note the distinction between the norm of a vector,  $\mathbf{v}$ , which is denoted by  $\|\mathbf{v}\|$ , and the absolute value of a scalar,  $s$ , which is denoted by  $|s|$ . A vector of norm 1 is called a *unit* vector, and Fig. 15.3(a) illustrates the three unit vectors  $\mathbf{i}, \mathbf{j}$  and  $\mathbf{k}$  directed along the positive  $x$ -,  $y$ - and  $z$ -axes,



**Fig. 15.3** (a) A vector  $\mathbf{v}$  ( $a, b, c$ ) in a three-dimensional space ( $x, y, z$ ) with initial point at the global origin O. (b) A vector  $\mathbf{v}'$  ( $x_t - x_i, y_t - y_i, z_t - z_i$ ) in a three-dimensional space ( $x, y, z$ ) with initial point  $i$  at  $(x_i, y_i, z_i)$  and terminal point  $t$  at  $(x_t, y_t, z_t)$ .

respectively. Member function `Vector3D::Norm()` returns the norm of a `Vector3D` object:

```
double Vector3D::Norm ()
{
    return sqrt ( array[0]*array[0] +
                  array[1]*array[1] +
                  array[2]*array[2] ) ;
}
```

Normalising a vector is performed by multiplying a non-zero norm vector,  $\mathbf{v}$ , by the reciprocal of its norm to obtain a resultant vector of norm 1 or a unit vector:

$$\frac{1}{\|\mathbf{v}\|} \mathbf{v}$$

which is implemented as:

```
void Vector3D::Normalise ()
{
    double n = this->Vector3D::Norm () ;
    assert (n > TOLERANCE) ;
    array[0]/=n ; array[1]/=n ; array[2]/=n ;
}
```

The assertion confirms that the `Vector3D` object operated on by the member function is of non-zero norm to eliminate division by zero.

Two important properties of vectors are the dot and cross product. The dot product of two non-zero norm vectors  $\mathbf{u}(u_1, u_2, u_3)$  and  $\mathbf{v}(v_1, v_2, v_3)$  is denoted by  $\mathbf{u} \cdot \mathbf{v}$  and given by:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta \quad \text{for} \quad \|\mathbf{u}\| \neq 0 \text{ and } \|\mathbf{v}\| \neq 0$$

where  $\theta$  is the angle between  $\mathbf{u}$  and  $\mathbf{v}$ . In terms of components it is easily found that:

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + u_3 v_3$$

The dot product of two Vector3D objects is implemented as:

```
double Vector3D::DotProduct (const Vector3D& v)
{
    return ( array[0]*v.array[0] +
              array[1]*v.array[1] +
              array[2]*v.array[2] ) ;
}
```

Above, the norm of a vector was determined in terms of the vector components. Alternatively, we could have defined the norm of a vector  $\mathbf{v}$  in terms of a dot product  $\|\mathbf{v}\| = (\mathbf{v} \cdot \mathbf{v})^{1/2}$ , since  $\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2$ . The dot product of two non-zero norm vectors,  $\mathbf{u}$  and  $\mathbf{v}$ , is particularly useful for determining the angle,  $\theta$  ( $0 \leq \theta \leq \pi$ ), between  $\mathbf{u}$  and  $\mathbf{v}$ . Angle  $\theta$  is acute if  $\mathbf{u} \cdot \mathbf{v} > 0$ , obtuse if  $\mathbf{u} \cdot \mathbf{v} < 0$  and equal to  $\pi/2$  if  $\mathbf{u} \cdot \mathbf{v} = 0$ . Thus, two vectors  $\mathbf{u}$  and  $\mathbf{v}$  are orthogonal (i.e. perpendicular) if and only if  $\mathbf{u} \cdot \mathbf{v} = 0$ .

The cross product of two vectors  $\mathbf{u}(u_1, u_2, u_3)$  and  $\mathbf{v}(v_1, v_2, v_3)$  is denoted by  $\mathbf{u} \times \mathbf{v}$  and defined by:

$$\mathbf{u} \times \mathbf{v} = \left( \begin{vmatrix} u_2 & u_3 \\ v_2 & v_3 \end{vmatrix}, - \begin{vmatrix} u_1 & u_3 \\ v_1 & v_3 \end{vmatrix}, \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \right) = (u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2 - u_2 v_1)$$

Note that the cross product of two vectors  $\mathbf{u}$  and  $\mathbf{v}$  is a vector that is orthogonal to  $\mathbf{u}$  and  $\mathbf{v}$ . Also, it is worth noting that the cross product of two vectors is anticommutative; i.e.  $\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$ . The cross product of two Vector3D objects is defined as:

```
Vector3D Vector3D::CrossProduct (const Vector3D& v)
{
    return Vector3D ( array[1]*v.array[2] - array[2]*v.array[1] ,
                      array[2]*v.array[0] - array[0]*v.array[2] ,
                      array[0]*v.array[1] -
                      array[1]*v.array[0] ) ;
}
```

VEC\_TST.CPP illustrates typical uses of classes Vector and Vector3D and member functions Vector3D::DotProduct() and Vector3D::CrossProduct():

```
// vec_tst.cpp
// tests base and derived classes Vector and Vector3D
#include <iostream.h> // C++ I/O
#include "vec.h" // Vector class
#include "vec3d.h" // Vector3D class

void main ()
{
    Vector v (5) ;
    cout << "Vector v: " << v << endl ;

    Vector3D v3d (1.0, 2.0), u3d (4.0, 3.0) ;
    cout << "Vector3D v3d: " << v3d << " ; "
```

---

```

<< "Vector3D u3d: " << u3d << endl ;

double vu_dot = v3d.DotProduct (u3d) ;
Vector3D vu_cross = v3d.CrossProduct (u3d) ;

cout << "v3d.u3d: " << vu_dot << endl
    << "v3dxu3d: " << vu_cross << endl ;
}

```

with output:

```

Vector v: [0.00, 0.00, 0.00, 0.00, 0.00]
Vector3D v3d: [1.00, 2.00, 0.00]; Vector3D u3d: [4.00, 3.00, 0.00]
uv3d.u3d: 10.00
v3dxu3d: [0.00, 0.00, -5.00]

```

Before examining Triangle and Tetrahedra classes, which make good use of Vector3D objects, let's revisit Fig. 15.3. The angles  $\alpha$ ,  $\beta$  and  $\gamma$  between vector  $v(a, b, c)$  and the unit vectors  $i, j$  and  $k$  are called the direction angles of  $v$ , and  $\cos \alpha$ ,  $\cos \beta$  and  $\cos \gamma$  are the direction cosines of  $v$ , given by:

$$\cos\alpha = \frac{a}{\|v\|}, \quad \cos\beta = \frac{b}{\|v\|}, \quad \cos\gamma = \frac{c}{\|v\|}$$

and have the following property:  $\cos^2 \alpha + \cos^2 \beta + \cos^2 \gamma = 1$ . In addition, the normalised unit vector  $v / \|v\|$  is equal to:

$$\frac{v}{\|v\|} = (\cos\alpha, \cos\beta, \cos\gamma)$$

which proves useful when evaluating the unit normal vector to a plane or surface at a given point, since the components of the vector are in fact the direction cosines of the vector.

Although the functions `Norm()`, `Normalise()` and `DotProduct()` are specific to **class** Vector3D, they could equally be member functions of **class** Vector, since each function can be generalised for  $n$ -dimensional Euclidean space.

## 15.16 Containment Versus Inheritance

This section examines the development of Triangle, Tetrahedra and associated classes such as Point and Line. The Tetrahedra **class** is developed using the two different approaches of containment and inheritance. However, before we examine containment versus inheritance, let's first discuss the Vector3D, Point and Line classes used in the design of the Triangle and Tetrahedra classes. The Vector3D **class** used below is identical to the Vector3D **class** declared and implemented in VEC3D.H/.CPP. The Point **class** is similar to that of POINT.H/.CPP, but does add some extra operations:

```

// pt.h
// class Point

```

```
#ifndef _PT_H // prevent multiple includes
#define _PT_H

#include <iostream.h> // C++ I/O
#include <math.h> // fabs(), sqrt()
#include "boolean.h" // Boolean enum
#include "vec3d.h" // Vector3D class

// class Point
class Point
{
protected:
    // protected data members
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (double d)
        : x (d), y(d), z (d) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    double& X () { return x; } // non-const objects
    double& Y () { return y; }
    double& Z () { return z; }
    const double& X () const { return x; } // const objects
    const double& Y () const { return y; }
    const double& Z () const { return z; }
    double DistanceToOrigin () ;
    // overloaded operator
    Point& operator = (const Point& p) ;
    double& operator [] (int n) ;
    const double& operator [] (int n) const ;
    Vector3D operator - (const Point& p) ;
    Boolean operator == (const Point& p) ;
    Boolean operator != (const Point& p) ;
    // friends
    friend ostream& operator << (ostream& s,
                                    const Point& p) ;
    friend istream& operator >> (istream& s, Point& p) ;
}; // class Point

#endif // _PT_H

// pt.cpp
// implementation file for Point class
```

```
#include "pt.h"  header file

const double TOLERANCE = 1e-06 ;

// member function:

// returns distance from Point to origin
double Point::DistanceToOrigin ()
{
    return sqrt (x*x + y*y + z*z) ;
}

// overloaded operator:

// assignment operator =
Point& Point::operator = (const Point& p)
{
    x = p.x ; y = p.y ; z = p.z ;
    return *this ;
}

// subscript operator [] (non-const objects)
double& Point::operator [] (int n)
{
    assert (n >= 0 && n < 3) ;
    if (n == 0)
        return x ;
    else if (n == 1)
        return y ;
    else
        return z ;
}

// subscript operator [] (const objects)
const double& Point::operator [] (int n) const
{
    assert (n >= 0 && n < 3) ;
    if (n == 0)
        return x ;
    else if (n == 1)
        return y ;
    else
        return z ;
}

// subtraction operator -
Vector3D Point::operator - (const Point& p)
{
    return Vector3D (x-p.x, y-p.y, z-p.z) ;
}
```

```

// compares two Points
Boolean Point::operator == (const Point& p)
{
    if (fabs(x-p.x)<TOLERANCE &&
        fabs(y-p.y)<TOLERANCE &&
        fabs(z-p.z)<TOLERANCE)
        return TRUE ;
    return FALSE ;
}

// compares two Points
Boolean Point::operator != (const Point& p)
{
    if (fabs(x-p.x)<TOLERANCE &&
        fabs(y-p.y)<TOLERANCE &&
        fabs(z-p.z)<TOLERANCE)
        return FALSE ;
    return TRUE ;
}

// overloaded operators/friends:

// insertion operator <<
ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
}

// extraction operator >>
istream& operator >> (istream& s, Point& p)
{
    return s >> ws >> p.x >> p.y >> p.z ;
}

```

The Point **class** listed above overloads the subscript operator for both constant and non-constant Point objects, the subtraction operator and the == and != relational operators. Note that the subtraction of two Point objects is a Vector3D object. A Line **class** is as follows:

```

// line.h
// header file for Line class

#ifndef _LINE_H // prevent multiple includes
#define _LINE_H

#include "pt.h" // Point class

class Line
{
protected:
    Point p0, p1 ;

```

```

public:
    // constructors
    Line ()
        : p0 (), p1 () {}
    Line (const Point& p, const Point& q)
        : p0 (p), p1 (q) {}
    // copy constructor
    Line (const Line& l)
        : p0 (l.p0), p1 (l.p1) {}
    // member functions
    Point& P0 () { return p0 ; }
    const Point& P0 () const { return p0 ; }
    Point& P1 () { return p1 ; }
    const Point& P1 () const { return p1 ; }
    double Length () ;
    // overloaded operators
    Line& operator = (const Line& l) ;
    Point& operator [] (int n) ;
    const Point& operator [] (int n) const ;
    // friend
    friend ostream& operator << (ostream& s, const Line& l) ;
}; // class Line

#ifndef _LINE_H

// line.cpp
// implementation file for class Line
#include "line.h" // header file

// returns the length of a Line
double Line::Length ()
{
    double x = p1.X () - p0.X () ;
    double y = p1.Y () - p0.Y () ;
    double z = p1.Z () - p0.Z () ;

    return sqrt (x*x + y*y + z*z) ;
}

// overloaded operators:

// assignment operator =
Line& Line::operator = (const Line& l)
{
    p0 = l.p0 ; p1 = l.p1 ;
    return *this ;
}

// subscript operator [] (non-const objects)
Point& Line::operator [] (int n)
{

```

```
assert (n >= 0 && n < 2) ;
return n == 0 ? p0 : p1 ;
}

// subscript operator [] (const objects)
const Point& Line::operator [] (int n) const
{
assert (n >= 0 && n < 2) ;
return n == 0 ? p0 : p1 ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Line& l)
{
return s << "[" << l.p0 << ", " << l.p1 << "] " ;
}
```

The **class** Line encapsulates two Point data members which represent the end-points of a three-dimensional straight line segment.

### 15.16.1 Containment

A **class** declaration can encapsulate a data member which is an object of another **class**:

```
class X
{
//...
};

class Y
{
//...
X x; // object x of class X
};
```

**class** Y has an X object. Such a relationship is frequently referred to as a *has a* relationship. The **class** Line *has a* relationship with **class** Point, since Line has two Point data members:

```
class Line
{
protected:
    Point p0, p1 ;
//...
};
```

Let's now examine the slightly more complicated Triangle **class**, which models an arbitrary planar triangle with straight-line edges in three-dimensional space:

```
// tri_con.h
// header file for Triangle class to illustrate containment
```

---

```

#ifndef _TRI_CON_H // prevent multiple includes
#define _TRI_CON_H

#include "pt.h"    // class Point
#include "line.h"   // class Line

class Triangle
{
private:
    Point* verts ;
    Line* edges ;
    int number ;
    void Allocate () ;

public:
    // constructors
    Triangle () ;
    Triangle (const Point& p0, const Point& p1,
               const Point& p2, int n=0) ;
    // copy constructor
    Triangle (const Triangle& t) ;
    // destructor
    ~Triangle () ;
    // member functions
    int& Number () { return number ; }
    const int& Number () const { return number ; }
    Point& Vertex (int n) ;
    const Point& Vertex (int n) const ;
    Line& Edge (int n) ;
    const Line& Edge (int n) const ;
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    Vector3D Normal () ;
    // overloaded operator
    Triangle& operator = (const Triangle& t) ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Triangle& t) ;
}; // class Triangle

#endif // _TRI_CON_H

```

Triangle declares three **private** data members: verts, edges and number. The verts data member is a pointer to an array of three Point objects which represent the three vertices of a triangle. edges is a pointer to an array of three Line objects which represent the three edges of a triangle. number is an integer constant assigned to a Triangle object and is zero by default. Triangle::Allocate() requests sufficient memory from the operating system for the arrays of vertices and edges. Note that it is not necessary to encapsulate both Point and Line data members within a Triangle **class**, and significant savings in memory can be made if we simply declare a pointer to an array of Point objects. However, declaring a

pointer to an array of `Line` objects greatly simplifies the operations performed on a `Triangle` object.

The member, `operator=()` and `friend` functions are defined as:

```
// tri_con.cpp
// implementation file for Triangle class
#include "tri_con.h" // header file

// allocates memory for a Triangle
void Triangle::Allocate ()
{
    verts = new Point[3] ;
    assert (verts != NULL) ;
    edges = new Line[3] ;
    assert (edges != NULL) ;
}

// constructors:

Triangle::Triangle ()
{
    Allocate () ;
    number = 0 ;
}

Triangle::Triangle (const Point& p0, const Point& p1,
                    const Point& p2, int n)
{
    Allocate () ;
    verts[0] = p0 ; verts[1] = p1 ; verts[2] = p2 ;
    edges[0].P0() = p0 ; edges[0].P1() = p1 ;
    edges[1].P0() = p1 ; edges[1].P1() = p2 ;
    edges[2].P0() = p2 ; edges[2].P1() = p0 ;
    number = n ;
}

// copy constructor
Triangle::Triangle (const Triangle& t)
{
    Allocate () ;
    verts[0] = t.verts[0] ; verts[1] = t.verts[1] ;
    verts[2] = t.verts[2] ;
    edges[0] = t.edges[0] ; edges[1] = t.edges[1] ;
    edges[2] = t.edges[2] ;
    number = t.number ;
}

// destructor
Triangle::~Triangle ()
{
    delete [] verts ;
```

```
delete [] edges ;
verts = NULL ;
edges = NULL ;
number = 0 ;
}

// member functions:

Point& Triangle::Vertex (int n)
{
    assert (n >= 0 && n < 3) ;
    return verts[n] ;
}

const Point& Triangle::Vertex (int n) const
{
    assert (n >= 0 && n < 3) ;
    return verts[n] ;
}

Line& Triangle::Edge (int n)
{
    assert (n >= 0 && n < 3) ;
    return edges[n] ;
}

const Line& Triangle::Edge (int n) const
{
    assert (n >= 0 && n < 3) ;
    return edges[n] ;
}

// returns the centroid of a Triangle
Point Triangle::Centroid ()
{
    Point cent ;

    cent.X () = (verts[0].X () + verts[1].X () +
                 verts[2].X ()) / 3.0 ;
    cent.Y () = (verts[0].Y () + verts[1].Y () +
                 verts[2].Y ()) / 3.0 ;
    cent.Z () = (verts[0].Z () + verts[1].Z () +
                 verts[2].Z ()) / 3.0 ;
    return cent ;
}

// returns the perimeter of a Triangle
double Triangle::Perimeter ()
{
    return (edges[0].Length () + edges[1].Length () +
            edges[2].Length ()) ;
```

```

}

// returns the surface area of a Triangle
double Triangle::SurfaceArea ()
{
    Vector3D u, v, cross ;
    // use vertex v0 as 'local' origin
    u = verts[1] - verts[0] ;
    v = verts[2] - verts[0] ;
    cross = u.CrossProduct (v) ;
    return cross.Norm () / 2.0 ;
}

// returns the normalised normal of a Triangle
Vector3D Triangle::Normal ()
{
    Vector3D u, v, normal ;
    // use vertex v0 as 'local' origin
    u = verts[1] - verts[0] ;
    v = verts[2] - verts[0] ;
    normal = u.CrossProduct (v) ;
    normal.Normalise () ;
    return normal ;
}

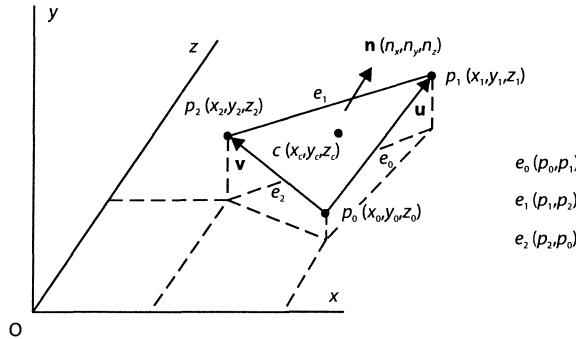
// overloaded operator:

// assignment operator =
Triangle& Triangle::operator = (const Triangle& t)
{
    verts[0] = t.verts[0] ; verts[1] = t.verts[1] ;
    verts[2] = t.verts[2] ;
    edges[0] = t.edges[0] ; edges[1] = t.edges[1] ;
    edges[2] = t.edges[2] ;
    number = t.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Triangle& t)
{
    s << "number : " << t.number << endl
    << "vertices: " << t.verts[0] << ", " << t.verts[1]
                                << ", " << t.verts[2] << endl
    << "edges     : " << t.edges[0] << ", " << t.edges[1]
                                << ", " << t.edges[2] ;
    return s ;
}

```

Member functions *Number()*, *Vertex()* and *Edge()* allow a user of **class Triangle** to get and set the number, vertices and edges of a **Triangle** object. **Triangle** also defines



**Fig. 15.4** An arbitrary planar triangle in a three-dimensional space.

four geometric member functions called *Centroid()*, *Perimeter()*, *SurfaceArea()* and *Normal()* which allow basic operations to be performed on a *Triangle* object. Let's begin examining these member functions by considering the centroid of a triangle. The centroid  $c(x_c, y_c, z_c)$  of a triangle defined by the three points  $p_0(x_0, y_0, z_0)$ ,  $p_1(x_1, y_1, z_1)$  and  $p_2(x_2, y_2, z_2)$  is given by (see Fig. 15.4):

$$(x_c, y_c, z_c) = \left( \frac{x_0 + x_1 + x_2}{3}, \frac{y_0 + y_1 + y_2}{3}, \frac{z_0 + z_1 + z_2}{3} \right)$$

and implemented as:

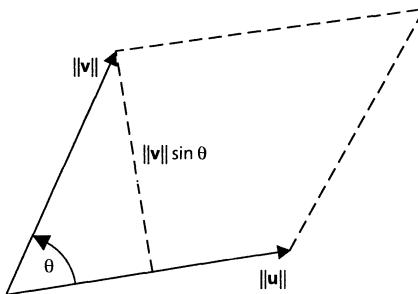
```
Point Triangle::Centroid ()
{
    Point cent ;
    cent.X () = (verts[0].X() + verts[1].X() + verts[2].X()) / 3.0 ;
    cent.Y () = (verts[0].Y() + verts[1].Y() + verts[2].Y()) / 3.0 ;
    cent.Z () = (verts[0].Z() + verts[1].Z() + verts[2].Z()) / 3.0 ;
    return cent ;
}
```

The perimeter of a triangle (Fig. 15.4), is simply the sum of the lengths of the three triangle edges  $e_0$ ,  $e_1$  and  $e_2$ , and is implemented as:

```
double Triangle::Perimeter ()
{
    return (edges[0].Length () + edges[1].Length () +
            edges[2].Length ()) ;
}
```

The area,  $A$ , of a triangle is one half the area of a parallelogram determined by two vectors  $\mathbf{u}$  and  $\mathbf{v}$  (Fig. 15.5):

$$A = \frac{1}{2} \|\mathbf{u}\| \|\mathbf{v}\| \sin\theta = \frac{1}{2} \|\mathbf{u} \times \mathbf{v}\|$$



**Fig. 15.5** A parallelogram formed by the two vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

and is implemented as:

```
double Triangle::SurfaceArea ()
{
    Vector3D u, v, cross ;
    // use vertex v0 as 'local' origin
    u = verts[1] - verts[0] ;
    v = verts[2] - verts[0] ;
    cross = u.CrossProduct (v) ;
    return cross.Norm () / 2.0 ;
}
```

The first vertex of the `Triangle` object is used as a *local* origin to form two `Vector3D` objects  $\mathbf{u}$  and  $\mathbf{v}$ .

It was noted earlier that the cross product of two vectors  $\mathbf{u}$  and  $\mathbf{v}$  is another vector and is orthogonal to both  $\mathbf{u}$  and  $\mathbf{v}$ . Thus, the normal  $\mathbf{n}(n_x, n_y, n_z)$  vector of a triangle is easily determined from the cross product of two vectors  $\mathbf{u}(p_1-p_0)$  and  $\mathbf{v}(p_2-p_0)$ ; see Fig. 15.4. Note that  $-\mathbf{n}=\mathbf{v}\times\mathbf{u}$ . The normal of a `Triangle` object is implemented as:

```
Vector3D Triangle::Normal ()
{
    Vector3D u, v, normal ;
    // use vertex v0 as 'local' origin
    u = verts[1] - verts[0] ;
    v = verts[2] - verts[0] ;
    normal = u.CrossProduct (v) ;
    normal.Normalise () ;
    return normal ;
}
```

Before a normal vector is returned by `Normal()` it is first normalised ( $\mathbf{n}/\|\mathbf{n}\|$ ) so that the components of the normal vector are the direction cosines of the vector. The normal vector to a plane or surface at a point is conventionally written as a unit vector, since the magnitude of  $\mathbf{n}$  is generally not required, only its direction.

It is worth noting in the above declaration and member function definitions just how much `class Triangle` relies on the classes `Point`, `Line`, `Vector3D` and how naturally they are used. Also, the above declaration/definition of `Triangle` helps to emphasise the importance

of good robust design of classes (particularly *primitive* classes), because classes can quickly become highly integrated.

The Tetrahedra **class** declaration is:

```
// tet_con.h
// header file for Tetrahedra class to illustrate containment

#ifndef _TET_CON_H // prevent multiple includes
#define _TET_CON_H

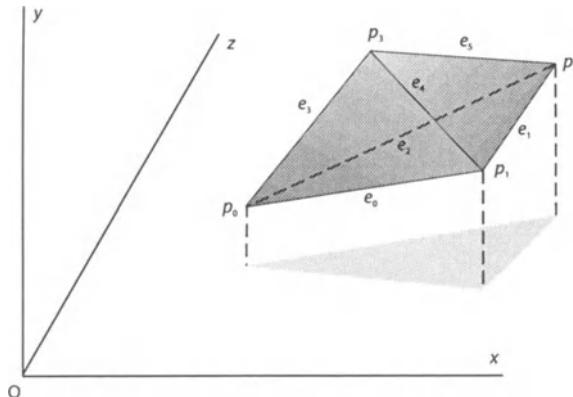
#include "tri_con.h" // class Triangle

class Tetrahedra
{
private:
    Triangle* faces ;
    int number ;
    void Allocate () ;

public:
    // constructors
    Tetrahedra () ;
    Tetrahedra (const Triangle& t0, const Triangle& t1,
                const Triangle& t2, const Triangle& t3,
                int n=0) ;
    // copy constructor
    Tetrahedra (const Tetrahedra& t) ;
    // destructor
    ~Tetrahedra () ;
    // member functions
    int& Number () { return number ; }
    const int& Number () const { return number ; }
    Triangle& Face (int n) ;
    const Triangle& Face (int n) const ;
    Point* Vertices () ;
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    double Volume () ;
    // overloaded operator
    Tetrahedra& operator = (const Tetrahedra& t) ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Tetrahedra& t) ;
}; // class Tetrahedra

#endif // _TET_CON_H
```

Figure 15.6 illustrates an arbitrary tetrahedron in three-dimensional space consisting of four vertices, six straight-line edges and four planar triangle faces. Tetrahedra models a three-dimensional tetrahedron by using containment to encapsulate a pointer data member, faces, to Triangle. The member function definitions of Tetrahedra are:



**Fig. 15.6** An arbitrary tetrahedron in three-dimensional space.

```

// tet_con.cpp
// implementation file for Tetrahedra class
#include "tet_con.h" // header file

// allocates memory for a Tetrahedra
void Tetrahedra::Allocate ()
{
    faces = new Triangle[4] ;
    assert (faces != NULL) ;
}

// constructors:

Tetrahedra::Tetrahedra ()
{
    Allocate () ;
    number = 0 ;
}

Tetrahedra::Tetrahedra (const Triangle& t0, const Triangle& t1,
                      const Triangle& t2, const Triangle& t3,
                      int n)
{
    Allocate () ;
    faces[0] = t0 ; faces[1] = t1 ;
    faces[2] = t2 ; faces[3] = t3 ;
    number = n ;
}

// copy constructor
Tetrahedra::Tetrahedra (const Tetrahedra& t)
{
    Allocate () ;
    faces[0] = t.faces[0] ; faces[1] = t.faces[1] ;
}

```

```
faces[2] = t.faces[2] ; faces[3] = t.faces[3] ;
number = t.number ;
}

// destructor
Tetrahedra::~Tetrahedra ()
{
delete [] faces ;
faces = NULL ;
number = 0 ;
}

// member functions:

Triangle& Tetrahedra::Face (int n)
{
assert (n >= 0 && n < 4) ;
return faces[n] ;
}

const Triangle& Tetrahedra::Face (int n) const
{
assert (n >= 0 && n < 4) ;
return faces[n] ;
}

// return a pointer to the four vertices of a Tetrahedra
Point* Tetrahedra::Vertices ()
{
Point* v_array = new Point[4] ;

// use faces[0] as base Triangle
v_array[0] = faces[0].Vertex (0) ;
v_array[1] = faces[0].Vertex (1) ;
v_array[2] = faces[0].Vertex (2) ;

// find fourth vertex
Point pt4 ;

for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
        if ( (pt4 = faces[1].Vertex(i))!=(faces[0].Vertex(j)) )
            break ; // found 4th. vertex
v_array[3] = pt4 ;

return v_array ;
}

// returns the centroid of a Tetrahedra
Point Tetrahedra::Centroid ()
{
```

```

Point cent ; // centroid
Point* verts = Vertices () ; // get 4 vertices of
                           // Tetrahedra

cent.X() = (verts[0].X()+verts[1].X()+verts[2].X()+
            verts[3].X())/4.0 ;
cent.Y() = (verts[0].Y()+verts[1].Y()+verts[2].Y()+
            verts[3].Y())/4.0 ;
cent.Z() = (verts[0].Z()+verts[1].Z()+verts[2].Z()+
            verts[3].Z())/4.0 ;
delete [] verts ; // delete memory associated with 4
                     // vertices
return cent ;
}

// returns the perimeter of a Tetrahedra
// uses vertices of a Tetrahedra to find the perimeter
// length since a Tetrahedra edge is common to 2 Triangles
double Tetrahedra::Perimeter ()
{
    Point* verts = Vertices () ; // get 4 vertices of
                               // Tetrahedra
    double perm = (verts[1]-verts[0]).Norm () +
                  (verts[2]-verts[0]).Norm () +
                  (verts[3]-verts[0]).Norm () +
                  (verts[3]-verts[1]).Norm () +
                  (verts[2]-verts[1]).Norm () +
                  (verts[3]-verts[2]).Norm () ;
    delete [] verts ; // delete memory associated with 4
                      // vertices
    return perm ;
}

// returns the surface area of a Tetrahedra
double Tetrahedra::SurfaceArea ()
{
    return (faces[0].SurfaceArea () + faces[1].SurfaceArea () +
             faces[2].SurfaceArea () + faces[3].SurfaceArea ()) ;
}

// returns the volume of a Tetrahedra
double Tetrahedra::Volume ()
{
    Point* verts = Vertices () ; // get 4 vertices of Tetrahedra
    // vol=(1/6)|a.(bxc)|
    // use vertex v0 as 'local' origin
    return fabs ( (verts[1]-verts[0]).DotProduct ( (verts[2]-
                                                    verts[0]).CrossProduct (verts[3]-verts[0]) )
                 ) / 6.0 ;
}

```

```

// overloaded operator:

// assignment operator =
Tetrahedra& Tetrahedra::operator = (const Tetrahedra& t)
{
    faces[0] = t.faces[0] ; faces[1] = t.faces[1] ;
    faces[2] = t.faces[2] ; faces[3] = t.faces[3] ;
    number = t.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Tetrahedra& t)
{
    s << "number : " << t.number << endl
        << "faces : " << endl << t.faces[0] << endl
                                << t.faces[1] << endl
                                << t.faces[2] << endl
                                << t.faces[3] ;

    return s ;
}

```

Because Tetrahedra uses containment to model a tetrahedron as four Triangle objects, and each Triangle face shares its three Point vertices with the other three Triangle faces, a *Vertices()* member function is required which returns a pointer to an array of the four Point vertices. *Vertices()* loops through the vertices of the first two faces of a Tetrahedra object to find the four unique vertices of a Tetrahedra object.

The centroid  $c(x_c, y_c, z_c)$  of a tetrahedron defined by the four points  $p_0(x_0, y_0, z_0)$ ,  $p_1(x_1, y_1, z_1)$ ,  $p_2(x_2, y_2, z_2)$  and  $p_3(x_3, y_3, z_3)$  is given by:

$$(x_c, y_c, z_c) = \left( \frac{x_0 + x_1 + x_2 + x_3}{4}, \frac{y_0 + y_1 + y_2 + y_3}{4}, \frac{z_0 + z_1 + z_2 + z_3}{4} \right)$$

and implemented as:

Note that the memory associated with the vertices is deallocated before the centroid is returned.

The perimeter length of a tetrahedron is the sum of its six edges and is implemented as follows:

```
double Tetrahedra::Perimeter ()
{
    Point* verts = Vertices () ; // get 4 vertices of
                                // Tetrahedra
    double perm = (verts[1]-verts[0]).Norm () +
                  (verts[2]-verts[0]).Norm () +
                  (verts[3]-verts[0]).Norm () +
                  (verts[3]-verts[1]).Norm () +
                  (verts[2]-verts[1]).Norm () +
                  (verts[3]-verts[2]).Norm () ;
    delete [] verts ; // delete memory associated with 4
                       // vertices
    return perm ;
}
```

The surface area of a tetrahedron is easily found from the sum of the area its four triangular faces:

```
double Tetrahedra::SurfaceArea ()
{
    return (faces[0].SurfaceArea () + faces[1].SurfaceArea () +
             faces[2].SurfaceArea () + faces[3].SurfaceArea ()) ;
}
```

Tetrahedra does not define a *Normal()* member function, since the normal vector is undefined for a tetrahedron. However, the normal of face n of a Tetrahedra object *tet\_obj* is easily found:

```
tet_obj.Face (n).Normal () ;
```

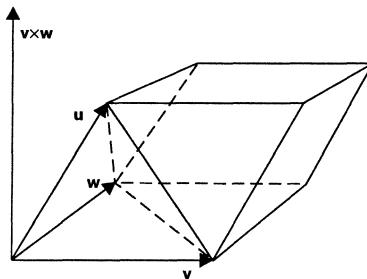
A tetrahedron does have volume and this is determined from considering the volume of a parallelepiped determined by the three vectors *u*, *v* and *w*, shown in Fig. 15.7. The volume, *V*, of a tetrahedron determined by *u*, *v* and *w* is one-sixth the volume of the parallelepiped:

$$V = \frac{1}{6} \|\mathbf{v} \times \mathbf{w}\| \frac{|\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})|}{\|\mathbf{v} \times \mathbf{w}\|} = \frac{1}{6} |\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})|$$

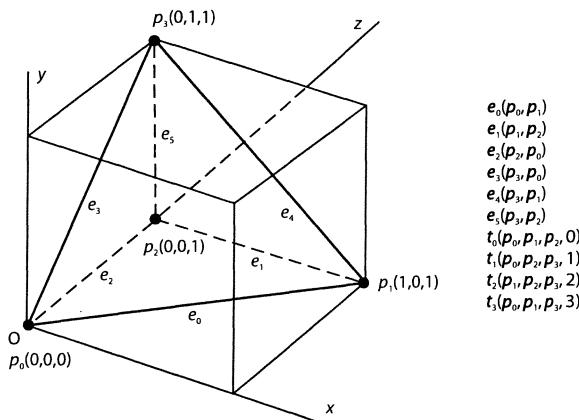
and implemented as:

```
double Tetrahedra::Volume ()
{
    Point* verts = Vertices () ; // get 4 vertices of
                                // Tetrahedra
    // vol=(1/6)/a.(bxc) /
    // use vertex v0 as 'local' origin
    return fabs ( (verts[1]-verts[0]).DotProduct ( (verts[2]-

```



**Fig. 15.7** A parallelepiped formed by the three vectors  $\mathbf{u}$ ,  $\mathbf{v}$  and  $\mathbf{w}$ .



**Fig. 15.8** A tetrahedron inside a unit cube.

```

    verts[0]).CrossProduct (verts[3]-verts[0]) )
) / 6.0 ;
}

```

Point and Vector3D classes allow a very concise definition of Tetrahedra::Volume().

File TTT\_CON.CPP tests the above Triangle and Tetrahedra classes for the tetrahedron described by the four points  $p_0, p_1, p_2$  and  $p_3$  shown in Fig. 15.8:

```

// ttt_con.cpp
// tests the Triangle and Tetrahedra classes
#include <iostream.h> // C++ I/O
#include "pt.h" // Point class
#include "tri_con.h" // Triangle class
#include "tet_con.h" // Tetrahedra class

void main ()
{
    Triangle tri_def ;
    cout << "tri_def: " << endl << tri_def << endl ;

```

```

Point p0 (0.0, 0.0, 0.0), p1 (1.0, 0.0, 1.0) ;
Point p2 (0.0, 0.0, 1.0), p3 (0.0, 1.0, 1.0) ;

Triangle t0 (p0, p1, p2, 0), t1 (p0, p2, p3, 1) ;
Triangle t2 (p1, p2, p3, 2), t3 (p0, p1, p3, 3) ;

Tetrahedra tet (t0, t1, t2, t3, 0) ;
cout << "tet: " << endl << tet
    << "tet.centroid" : " << tet.Centroid() << endl
    << "tet.perimeter" : " << tet.Perimeter () << endl
    << "tet.surface_area" : " << tet.SurfaceArea () << endl
    << "tet.volume" : " << tet.Volume () << endl ;

/*
Triangle* t_array = new Triangle[4] ;

t_array[0] = t0 ; t_array[1] = t1 ;
t_array[2] = t2 ; t_array[3] = t3 ;

for (int i=0; i<4; i++)
    cout << "t_array[" << i << "] " << endl
        << t_array[i] << endl ;

delete [] t_array ;
*/
}

```

with output:

```

tri_def:
number : 0
vertices: (0, 0, 0), (0, 0, 0), (0, 0, 0)
edges   : [(0, 0, 0), (0, 0, 0)], [(0, 0, 0), (0, 0, 0)],
           [(0, 0, 0), (0, 0, 0)]

tet:
number: 0
faces:
number : 0
vertices: (0, 0, 0), (1, 0, 1), (0, 0, 1)
edges   : [(0, 0, 0), (1, 0, 1)], [(1, 0, 1), (0, 0, 1)],
           [(0, 0, 1), (0, 0, 0)]
number : 1
vertices: (0, 0, 0), (0, 0, 1), (0, 1, 1)
edges   : [(0, 0, 0), (0, 0, 1)], [(0, 0, 1), (0, 1, 1)],
           [(0, 1, 1), (0, 0, 0)]
number : 2
vertices: (1, 0, 1), (0, 0, 1), (0, 1, 1)
edges   : [(1, 0, 1), (0, 0, 1)], [(0, 0, 1), (0, 1, 1)],
           [(0, 1, 1), (1, 0, 1)]
number : 3

```

---

```

vertices: (0, 0, 0), (1, 0, 1), (0, 1, 1)
edges   : [(0, 0, 0), (1, 0, 1)], [(1, 0, 1), (0, 1, 1)],
          [(0, 1, 1), (0, 0, 0)]
tet.centroid    : (0.25, 0.25, 0.75)
tet.perimeter   : 7.242264
tet.surface_area: 2.36303
tet.volume       : 0.166667

```

The `main()` function of TTT\_CON.CPP also contains a commented-out array, `t_array`, of four `Triangle` objects.

## 15.16.2 Inheritance

When a `class` Y is derived from a `class` X, Y is a kind of X, and such a relationship is frequently referred to as a *isa* relationship:

```

class X
{
//...
};

class Y : public X
{
//...
};

```

`class` Y is a kind of X. The `class` `Vector3D` *isa* relationship with `class` `Vector` since `Vector3D` is a kind of `Vector`:

```

class Vector3D : public Vector
{
//...
};

```

Let us now derive `class` `Tetrahedra` from `class` `Triangle`, thus making Tetrahedra a kind of Triangle. First, we need to modify the data members of `Triangle` in TRI\_CON.H from `private` to `protected` to enable them to be inherited:

```

// tri_inh.h
// header file for Triangle class to illustrate inheritance
//...
class Triangle
{
protected:
    Point* verts ;
    Line* edges ;
    int number ;
    void Allocate () ;
public:
    //...
};

```

with corresponding implementation file TRI\_INH.CPP.

TRI\_INH.H/.CPP are identical to TRI\_CON.H/.CPP except that the data members of Triangle in TRI\_INH.H are declared **protected**. The declaration of Tetrahedra is now given in TET\_INH.H:

```
// tet_inh.h
// header file for Tetrahedra class to illustrate inheritance

#ifndef _TET_INH_H // prevent multiple includes
#define _TET_INH_H

#include "tri_inh.h" // class Triangle

class Tetrahedra : public Triangle
{
protected:
    void Allocate () ;
public:
    // constructors
    Tetrahedra () ;
    Tetrahedra (const Point& p0, const Point& p1,
                const Point& p2, const Point& p3, int n=0) ;
    // copy constructor
    Tetrahedra (const Tetrahedra& t) ;
    // member functions
    Triangle Face (int n) ;
    Point* Vertices () ;
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    double Volume () ;
    // overloaded operator
    Tetrahedra& operator = (const Tetrahedra& t) ;
    // friend
    friend ostream& operator << (ostream& s,
                                    const Tetrahedra& t) ;
}; // class Tetrahedra

#endif // _TET_INH_H
```

Tetrahedra does not add any additional data members to those inherited from Triangle, and consequently does not need to define its own destructor but simply calls, by default, the destructor of Triangle. The implementation file TET\_INH.CPP for Tetrahedra is:

```
// tet_inh.cpp
// implementation file for Tetrahedra class
#include "tet_inh.h" // header file

// allocates memory for a Tetrahedra
void Tetrahedra::Allocate ()
{
```

```

verts = new Point[4] ;
assert (verts != NULL) ;
edges = new Line[6] ;
assert (edges != NULL) ;
}

// constructors:

Tetrahedra::Tetrahedra ()
{
Tetrahedra::Allocate () ;
number = 0 ;
}

Tetrahedra::Tetrahedra (const Point& p0, const Point& p1,
const Point& p2, const Point& p3,
int n)
{
Tetrahedra::Allocate () ;
verts[0] = p0 ; verts[1] = p1 ; verts[2] = p2 ; verts[3] = p3 ;
edges[0].P0() = p0 ; edges[0].P1() = p1 ;
edges[1].P0() = p1 ; edges[1].P1() = p2 ;
edges[2].P0() = p2 ; edges[2].P1() = p0 ;
edges[3].P0() = p0 ; edges[3].P1() = p3 ;
edges[4].P0() = p1 ; edges[4].P1() = p3 ;
edges[5].P0() = p2 ; edges[5].P1() = p3 ;
number = n ;
}

// copy constructor
Tetrahedra::Tetrahedra (const Tetrahedra& t)
{
Tetrahedra::Allocate () ;
verts[0] = t.verts[0] ; verts[1] = t.verts[1] ;
verts[2] = t.verts[2] ; verts[3] = t.verts[3] ;
edges[0] = t.edges[0] ; edges[1] = t.edges[1] ;
edges[2] = t.edges[2] ; edges[3] = t.edges[3] ;
edges[4] = t.edges[4] ; edges[5] = t.edges[5] ;
number = t.number ;
}

// member functions:

// returns a Triangle face object
Triangle Tetrahedra::Face (int n)
{
assert (n >= 0 && n<4) ;
if (n == 0)
    return Triangle (verts[0], verts[1], verts[2], 0) ;
else if (n == 1)
    return Triangle (verts[0], verts[2], verts[3], 1) ;
}

```



```

// returns the volume of a Tetrahedra
double Tetrahedra::Volume ()
{
    // vol=(1/6)/a.(bxc) /
    // use vertex v0 as 'local' origin
    return fabs ( (verts[1]-verts[0]).DotProduct ( (verts[2]-
        verts[0]).CrossProduct (verts[3]-verts[0]) )
    ) / 6.0 ;
}

// overloaded operator:

// assignment operator =
Tetrahedra& Tetrahedra::operator = (const Tetrahedra& t)
{
    verts[0] = t.verts[0] ; verts[1] = t.verts[1] ;
    verts[2] = t.verts[2] ; verts[3] = t.verts[3] ;
    edges[0] = t.edges[0] ; edges[1] = t.edges[1] ;
    edges[2] = t.edges[2] ; edges[3] = t.edges[3] ;
    edges[4] = t.edges[4] ; edges[5] = t.edges[5] ;
    number = t.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Tetrahedra& t)
{
    s << "number : " << t.number << endl
    << "vertices: " << t.verts[0] << ", " << t.verts[1] << ", "
    << t.verts[2] << ", " << t.verts[3] << endl
    << "edges : " << t.edges[0] << ", " << t.edges[1] << ", "
    << t.edges[2] << ", " << t.edges[3] << ", "
    << t.edges[4] << ", " << t.edges[5] ;
    return s ;
}

```

`Tetrahedra::Allocate()` overrides `Triangle::Allocate()` by allocating sufficient memory for four vertices and six edges rather than the three vertices and edges of a `Triangle` object. The member functions `Centroid()`, `Perimeter()`, `SurfaceArea()` and `Volume()` are similar in principle to those described above for `TET_CON.CPP`. However, `TET_INH.CPP` does define a new member function called `Face()`, which returns one of the four `Triangle` faces of a `Tetrahedra`.

A program which tests the `Triangle` and `Tetrahedra` classes declared and defined in `TRI_INH.H/.CPP` and `TET_INH.H/.CPP` is:

```

// ttt_inh.cpp
// tests the Triangle and Tetrahedra classes
#include <iostream.h> // C++ I/O
#include "pt.h"        // Point class
#include "tri_inh.h"   // Triangle class
#include "tet_inh.h"   // Tetrahedra class

```

```

void main ()
{
    Point p0 (0.0, 0.0, 0.0), p1 (1.0, 0.0, 1.0) ;
    Point p2 (0.0, 0.0, 1.0), p3 (0.0, 1.0, 1.0) ;

    Tetrahedra tet (p0, p1, p2, p3, 0) ;
    cout << "tet: " << endl << tet << endl
        << "tet.centroid : " << tet.Centroid() << endl
        << "tet.perimeter : " << tet.Perimeter () << endl
        << "tet.surface_area: " << tet.SurfaceArea () << endl
        << "tet.volume : " << tet.Volume () << endl ;
}

```

with output similar to that generated by TTT\_CON.CPP.

### 15.16.3 Containment or Inheritance?

In view of the above, how do we choose between containment and inheritance? Unfortunately, there is no simple answer to this question, and generally the choice between containment and inheritance depends on the application. From a user's perspective, both approaches operate in a similar manner and can generally be designed to operate identically. In the present case, it is tempting to assume that `Tetrahedra` is derived from `Triangle`. However, on closer examination we see that `Tetrahedra` is not a kind of `Triangle` but instead has four `Triangle` objects. In other words, `Tetrahedra` *has a* relationship to `Triangle`.

A frequently adopted method of choosing between containment and inheritance is to consider whether or not a derived **class** can have two or more base **class** objects – the so-called ‘can it have two?’ question. If the answer is yes, then use containment rather than inheritance. In the case of a `Tetrahedra`, it is much easier to visualise a tetrahedron as having four triangular faces rather than a tetrahedron being a kind of triangle. Similarly, a `Line` was modelled as having two `Point` data members and a `Triangle` as having three `Points` and three `Lines`.

In addition, when choosing between containment and inheritance bear in mind the implicit conversion from a derived **class** to a base **class**. Clearly, a conversion from a `Tetrahedra` to a `Triangle` is meaningless.

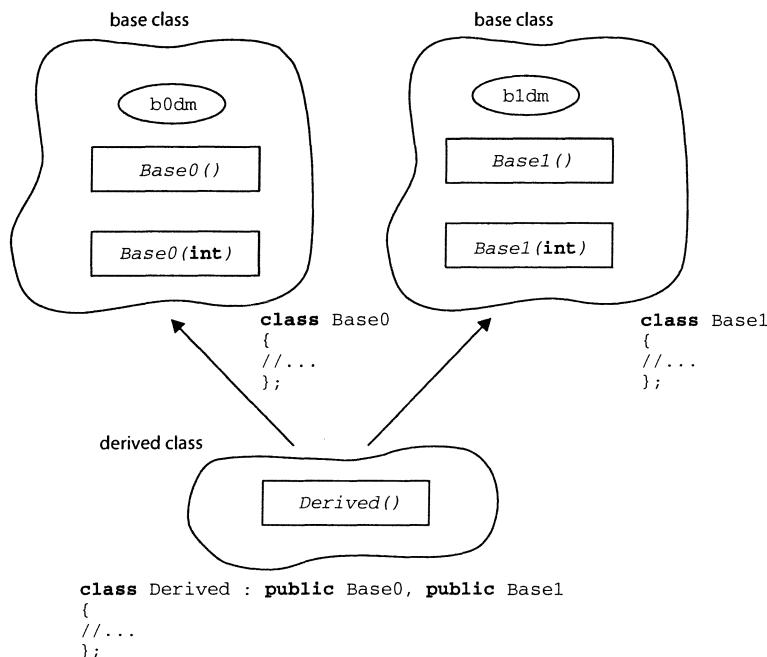
### 15.17 Multiple Inheritance

So far, all of the derived classes discussed have been derived from a single base **class**. C++ also allows us to declare a **class** which is derived from multiple base classes. Deriving a **class** from more than one base **class** is referred to as *multiple inheritance*. Figure 15.9 schematically illustrates multiple inheritance in which **class** `Derived` is derived from classes `Base0` and `Base1`. The following program demonstrates multiple inheritance:

```

// mult_inh.cpp
// illustrates multiple inheritance
#include <iostream.h> // C++ I/O

```



**Fig. 15.9** A derived **class** inheriting the features of two base classes.

```

// base class
class Base0
{
protected:
    int b0dm ;
public:
    // constructors
    Base0 ()
        : b0dm (0) {}
    Base0 (int i)
        : b0dm (i) {}
};

// base class
class Base1
{
protected:
    int b1dm ;
public:
    // constructors
    Base1 ()
        : b1dm (0) {}
    Base1 (int i)
        : b1dm (i) {}
};

```

```
// derived class from classes Base0 and Base1
class Derived : public Base0, public Base1
{
public:
    Derived (int i)
        : Base0 (i), Base1 (i) {}
    friend ostream& operator << (ostream& s,
                                    const Derived& d)
    { return s << d.b0dm << " " << d.b1dm ; }
};

void main ()
{
    Derived d (1) ;
    cout << d ;
}
```

with output:

1 1

The declaration of **class** Derived is:

```
class Derived : public Base0, public Base1
{
//...
```

and is of the general form:

```
class DerivedClass : accessSpecifier BaseClass0,
                     ..., accessSpecifier BaseClassn
{
//...
};
```

The multiple base classes are separated by the comma operator. Note that each base **class** inherited is accompanied by its own access specifier and not, for instance:

```
class Derived : public Base0, Base1
{
//...
};
```

### 15.17.1 Constructors, Destructors and Multiple Inheritance

Let's now examine the execution of constructors and destructors for multiple inheritance more closely. First of all we'll declare an empty derived **class** with no constructor(s) or destructor explicitly defined:

```

// m_no_c&d.cpp
// illustrates a class with no constructor and destructor
// derived from two base classes
#include <iostream.h> // C++ I/O

// base class
class Base0
{
public:
    Base0 ()
    { cout << "Base0 class constructor called"
      << endl ; }
    ~Base0 ()
    { cout << "Base0 class destructor called"
      << endl ; }
};

// base class
class Base1
{
public:
    Base1 ()
    { cout << "Base1 class constructor called"
      << endl ; }
    ~Base1 ()
    { cout << "Base1 class destructor called"
      << endl ; }
};

// derived class from classes Base0 and Base1
class Derived : public Base0, public Base1
{
};

void main ()
{
    Derived d_obj ;
}

```

with output:

```

Base0 class constructor called
Base1 class constructor called
Base1 class destructor called
Base0 class destructor called

```

The program output illustrates that if a constructor and destructor are not defined for Derived then the appropriate constructor or destructor of the base classes Base0 and Base1 is called. The order in which the base **class** constructors are called matches the order in which they are specified in the derived **class**'s inheritance list. As with single inheritance, base **class** destructors are called in reverse order.

Let's now extend the above program by simply defining a constructor and destructor for **class Derived**:

```
// m_c&d.cpp
// illustrates a class with a constructor and destructor
// derived from two base classes
#include <iostream.h> // C++ I/O

// base class
class Base0
{
public:
    Base0 ()
    { cout << "Base0 class constructor called"
      << endl ; }
    ~Base0 ()
    { cout << "Base0 class destructor called"
      << endl ; }
};

// base class
class Basel
{
public:
    Basel ()
    { cout << "Basel class constructor called"
      << endl ; }
    ~Basel ()
    { cout << "Basel class destructor called"
      << endl ; }
};

// derived class from classes Base0 and Basel
class Derived : public Base0, public Basel
{
public:
    Derived ()
    { cout << "Derived class constructor called"
      << endl ; }
    ~Derived ()
    { cout << "Derived class destructor called"
      << endl ; }
};

void main ()
{
    Derived d_obj ; }
```

with output:

```

Base0 class constructor called
Base1 class constructor called
Derived class constructor called
Derived class destructor called
Base1 class destructor called
Base0 class destructor called

```

The output illustrates that base **class** constructors are called in the order that they are specified in the derived **class** inheritance list prior to the derived **class** constructor, and vice versa for **class** destructors.

As with single inheritance, derived **class** constructor arguments can be passed to base **class** constructors using the following general syntax:

```

DerivedClass (argument_list)
  : BaseClass0 (argument_list), ..., BaseClassn (argument_list)
{
//...
}

```

This is demonstrated in the following program:

```

// d_to_bs.cpp
// illustrates passing constructor argument objects
// from a derived class constructor to base class constructors
#include <iostream.h> // C++ I/O

// base class
class Base0
{
protected:
    int b0dm ;
public:
    Base0 ()
        : b0dm (0) {}
    Base0 (int i)
        : b0dm (i) {}
};

// base class
class Base1
{
protected:
    int b1dm ;
public:
    Base1 ()
        : b1dm (0) {}
    Base1 (int i)
        : b1dm (i) {}
};

// derived class from classes Base0 and Base1

```

```

class Derived : public Base0, public Base1
{
public:
    Derived (int i, int j)
        : Base0 (i), Base1 (j) {}
};

void main ()
{
    Derived d_obj (0, 1);
}

```

## 15.18 virtual Base Classes

When a **class** is derived from two or more base classes ambiguities can arise when accessing data members and member functions. Consider the following program:

```

// m_prob.cpp
// illustrates ambiguous accessing of data members by a
// derived class
#include <iostream.h> // C++ I/O

// base class
class Base0
{
protected:
    int bdm ;
public:
    void Display () { cout << bdm << endl ; }
};

// base class
class Base1
{
protected:
    int bdm ;
public:
    void Display () { cout << bdm << endl ; }
};

// derived class from classes Base0 and Base1
class Derived : public Base0, public Base1
{
public:
    void Function (int i)
    {
        bdm = i;           // error: bdm is ambiguous
        Base1::bdm = i;   // o.k.: ambiguity resolved
    }
}

```

```

};

void main ()
{
    Derived d_obj ;

    d_obj.Display () ;           // error: Display() is ambiguous

    d_obj.Base0::Display () ;   // o.k.: ambiguity resolved
    d_obj.Base1::Display () ;   // o.k.: ambiguity resolved
}

```

Base classes Base0 and Base1 have a data member called bdm and a member function called *Display()*. The **class** Derived is derived from both base classes, Base0 and Base1, and Derived does not override any of its base **class** members. The function *Derived::Function()* illustrates an ambiguous reference to the bdm data member, since the compiler has no way of knowing which of the bdm data members of classes Base0 and Base1 is being referred to. Similarly, the object d\_obj of **class** Derived calls the *Display()* member function – but which one? The solution to these access problems is through the use of the scope resolution operator. If ambiguity exists, simply prefix the data member or member function by the name of its associated **class**.

Let us consider another slightly different program in which access ambiguity can occur:

```

// m_prob1.cpp
// further illustrates ambiguous accessing
// of data members by a derived class
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    void Display () { cout << bdm << endl ; }
};

// derived class from class Base
class Derived0 : public Base
{
};

// Derived class from class Base
class Derived1 : public Base
{
};

// derived class from classes Derived0 and Derived1
class Derived2 : public Derived0, public Derived1
{
};

```

```

void main ()
{
    Derived2 d_obj ;

    d_obj.Display () ;                                // error: Display() is
                                                    // ambiguous

    d_obj.Derived0::Display () ; // o.k.: ambiguity resolved
    d_obj.Derived1::Display () ; // o.k.: ambiguity resolved
}

```

In this program, **class** Derived2 is derived from classes Derived0 and Derived1 which are in turn both derived from **class** Base. As a result, there are two copies of **class** Base present in an object of **class** Derived2, and unless the problem of scope is resolved the compiler will not know which copy of Base to use.

What if we required that **class** Derived2 did not inherit two copies of Base but only one? In this instance C++ allows a derived **class** to be derived from **virtual** base classes. Let's modify the above program M\_PROB1.CPP by using **virtual** base classes to ensure that Derived2 contains only a single copy of Base:

```

// v_b_clss.cpp
// illustrates virtual base classes
#include <iostream.h> // C++ I/O

// base class
class Base
{
protected:
    int bdm ;
public:
    void Display () { cout << bdm << endl ; }
};

// derived class from class Base
class Derived0 : virtual public Base
{
};

// Derived class from class Base
class Derived1 : virtual public Base
{
};

// derived class from classes Derived0 and Derived1
class Derived2 : public Derived0, public Derived1
{
};

void main ()
{
    Derived2 d_obj ;
}

```

---

```
d_obj.Display () ; // o.k.: unambiguous
}
```

With the help of the keyword **virtual**, classes Derived0 and Derived1 declare **class Base** as **virtual** in each of their inheritance lists:

```
class Derived0 : virtual public Base
{
};

class Derived1 : virtual public Base
{
};
```

Any multiple inheritance involving classes Derived0 and Derived1 will result in a single copy of Base being made in the derived **classes** Derived0 and Derived1.

## 15.19 **virtual** Functions

Previous chapters have examined function and operator overloading. These are examples of compile-time polymorphism in which a function name can operate in a variety of different ways. The term *polymorphism* refers to the general concept that a single name may denote a variety of different objects of different derived classes which are related by a common base **class**. Each object denoted by a common name will respond differently to a given set of instructions.

Run-time polymorphism exists when dynamic binding and inheritance combine. C++ supports run-time polymorphism through the use of pointers to derived classes and **virtual** functions<sup>3</sup>. Virtual functions allow a function of a base **class** to be redefined in a derived **class**.

Polymorphism is one of the most powerful features of an object-oriented programming language and is one of the key features that distinguishes an object-oriented language from other languages. Polymorphism allows a programmer to design a highly generalised **class** with operations that are common to all classes derived from the **class** and at the same time allow derived classes to add additional operations specific to the derived classes. Thus, polymorphism gives a programmer the opportunity to design a common interface based on a base **class** which dictates the general operation of the interface, while derived classes define multiple specialised operations. This is why polymorphism is frequently referred to as ‘one interface, multiple methods’. An interface can be developed relatively independently of derived classes, and new derived classes can be easily added without affecting the overall design of the interface.

The use of **virtual** functions relies heavily on pointers, so let’s begin by examining a program which accesses non-**virtual** member functions using pointers:

```
// non_virt.cpp
// illustrates the use of pointers to non-virtual member
// functions
```

---

<sup>3</sup> A **class** containing one or more **virtual** functions is often referred to as a *polymorphic class*.

```
#include <iostream.h> // C++ I/O

class Base
{
public:
    void Display ()
    { cout << "Base::Display() called" << endl ; }

class Derived0 : public Base
{
public:
    void Display ()
    { cout << "Derived0::Display() called" << endl ; }

class Derived1 : public Base
{
public:
    void Display ()
    { cout << "Derived1::Display() called" << endl ; }

void main ()
{
    Base* b_ptr (NULL) ;

    Derived0 d0_obj ;
    Derived1 d1_obj ;

    b_ptr = &d0_obj ; // b_ptr pointer points to d0_obj object
    b_ptr->Display () ;

    b_ptr = &d1_obj ; // b_ptr pointer points to d1_obj object
    b_ptr->Display () ;
}
```

with output:

```
Base::Display() called
Base::Display() called
```

The `main()` function of NON\_VIRT.CPP defines a pointer to **class** `Base`, `b_ptr`, and two objects of classes `Derived0` and `Derived1`, `d0_obj` and `d1_obj`. The memory address of object `d0_obj` is then assigned to the `Base` **class** pointer:

```
b_ptr = &d0_obj ;
```

This mixing of classes is a valid assignment in the present case because C++ allows a base **class** pointer to point to an object of a **class** derived from the base **class**. However, the output illustrates that although a base **class** pointer may point to objects of a derived **class**,

the overridden member functions are still called according to the **class** or type of the pointer, and not what it points to.

A base **class** pointer can be used to access data members or member functions of a derived **class** by explicitly casting the base **class** pointer to the appropriate derived **class**. For instance, the `Derived0::Display()` member function could be accessed by `b_ptr` as:

```
((Derived0*)b_ptr)->Display () ;
```

An important point to note when using a pointer to a base **class** is that the pointer should generally not be incremented or decremented. This is because the pointer will be incremented or decremented relative to the base **class**, but the base **class** pointer could be pointing to a derived **class** object!

Let us now modify the above program by declaring the `Base::Display()` member function as a **virtual** member function by using the **virtual** keyword:

```
// virt.cpp
// illustrates the use of virtual member functions
#include <iostream.h> // C++ I/O

class Base
{
public:
    virtual void Display ()
    { cout << "Base::Display() called" << endl ; }

class Derived0 : public Base
{
public:
    void Display ()
    { cout << "Derived0::Display() called" << endl ; }

class Derived1 : public Base
{
public:
    void Display ()
    { cout << "Derived1::Display() called" << endl ; }

void main ()
{
    Base* b_ptr (NULL) ;

    Derived0 d0_obj ;
    Derived1 d1_obj ;

    b_ptr = &d0_obj ; // b_ptr pointer points to d0_obj object
    b_ptr->Display () ;

    b_ptr = &d1_obj ; // b_ptr pointer points to d1_obj object
```

```
b_ptr->Display () ;  
}
```

with output:

```
Derived0::Display() called  
Derived1::Display() called
```

The output now illustrates that the member function of the derived **class** pointed to by the base **class** pointer is called. The particular member function called is determined by the **class** or type of the object pointed to by the base **class** pointer. Thus, the following function-call statement:

```
b_ptr->Display () ;
```

is now capable of executing different functions with the same name, with each function performing its own set of operations. This is an example of run-time polymorphism. The type of polymorphism is run-time rather than compile-time because at the time of compilation the **class** or type operated on by the above function-call statement is undetermined.

A member function is declared as **virtual** in a base **class** using the **virtual** keyword:

```
class BaseClass  
{  
//...  
virtual return_type_specifier MemberFunction /*...*/ ;  
//...  
};
```

When *MemberFunction()* is redefined in a **class** or **classes** derived from **BaseClass**, the **virtual** keyword is not repeated. Repeating the **virtual** keyword in an overriding derived **class** member function declaration is legal but redundant.

There are a number of important properties of **virtual** functions which are worth highlighting:

- **virtual** functions are not simply overloaded functions with the keyword **virtual** attached. An overloaded function must differ in either or both of its **class** or type and number of arguments. A redefined **virtual** function must have exactly the same **class** or type and number of arguments. If two member functions have different **class**, type or number of arguments they are considered different functions and the **virtual** function mechanism is ignored. Generally, an overriding member function declaration cannot differ only by return type from an overridden member function. The exception to this rule is when the overridden **virtual** member function returns a pointer or reference to the base **class** and the overriding member function returns a pointer or a reference to the derived **class**.
- **virtual** functions are inherited and hierarchical irrespective of the depth of inheritance.
- A **virtual** function must be defined for a **class** in which it is first declared. The exception to this is when the function is declared as a *pure virtual* function. Pure **virtual** functions are discussed later.
- If a derived **class** does not override a **virtual** function then the **virtual** function defined in the base **class** is used.

- A **virtual** function must be a member function of a given **class** and not a **friend**.
- A **virtual** function can be declared a **friend** of another **class**.
- **virtual** functions cannot be **static** member functions.
- **virtual** functions can be declared as **inline**.
- Destructor functions can be **virtual** functions.
- Constructor functions cannot be **virtual** functions.

If we consider the first of the above list of points, the following program code illustrates declaring a function called *Display()* in a derived **class** which has a different number of arguments from the base **class** **virtual** function *Display()*:

```
class Base
{
//...
virtual void Display () ;
//...
};

class Derived : public Base
{
//...
// warning: Derived::Display() hides Base::Display()
void Display (int i) ;
};
```

Most compilers will issue a compilation warning message indicating that *Derived::Display()* hides the **virtual** function *Base::Display()*. However, the following is illegal because the overriding **virtual** function *Derived::Display()* conflicts with base **class** *Base*, owing to the two functions having different return types:

```
class Base
{
//...
virtual void Display () ;
//...
};

class Derived : public Base
{
//...
// error: Derived::Display() conflicts with Base::Display()
int Display () ;
};
```

whereas the following program code is legal because *Base::Function()* is returning a reference to *Base* and *Derived::Function()* is returning a reference to *Derived*:

```
class Base
{
//...
virtual Base& Display () ;
```

```
// ...
};

class Derived : public Base
{
//...
Derived& Display () ; // o.k.
};
```

To illustrate that **virtual** functions are hierarchical, consider the following program:

```
// v_hier.cpp
// illustrates that virtual functions are hierarchical
#include <iostream.h> // C++ I/O

class Base
{
public:
    virtual void Display ()
    { cout << "Base::Display() called" << endl ; }

// derived from Base
class Derived0 : public Base
{
public:
    void Display ()
    { cout << "Derived0::Display() called" << endl ; }

// derived from Derived0
class Derived1 : public Derived0
{
};

void main ()
{
    Base* b_ptr (NULL) ;

    Derived0 d0_obj ;
    Derived1 d1_obj ;

    b_ptr = &d0_obj ; // b_ptr pointer points to d0_obj object
    b_ptr->Display () ;

    b_ptr = &d1_obj ; // b_ptr pointer points to d1_obj object
    b_ptr->Display () ;
}
```

with output:

---

```
Derived0::Display() called
Derived0::Display() called
```

The **class** Derived1 is now derived from Derived0 and not Base. The output illustrates that when object d1\_obj is assigned to the Base **class** pointer and *Display()* is called, it is the function Derived0::*Display()* that is called and not Base::*Display()*. This is because Derived0 is hierarchically closer to Derived1 than is Base.

Generally, it is necessary to ensure that destructors are **virtual**. Consider the following program:

```
// v_dest.cpp
// illustrates the need for virtual destructors

#include <iostream.h> // C++ I/O

class Base
{
private:
    int* int_p ;
public:
    Base ()
        : int_p (new int[10]) {}
    ~Base ()
    {
        cout << "~Base()" << endl ;
        delete [] int_p ;
    }
};

class Derived : public Base
{
private:
    int* int_q ;
public:
    Derived ()
        : int_q (new int[10]) {}
    ~Derived ()
    {
        cout << "~Derived()" << endl ;
        delete [] int_q ;
    }
};

void main ()
{
    Base* b_ptr (new Derived) ;
    delete b_ptr ;
}
```

with output:

```
~Base ()
```

Both Base and Derived encapsulate pointers to **int** which allocate memory upon object definition. Although the base **class** pointer, **b\_ptr**, is pointing to Derived, the call to **delete** will only invoke the Base destructor at run-time. As a consequence, the memory allocated by the Derived constructor will not be deallocated. If the Base destructor is made **virtual**, then at run-time the appropriate destructor will be called for the object pointed to by the base **class** pointer:

```
~Derived()  
~Base()
```

### 15.19.1 Pure virtual Functions and Abstract Classes

If a **virtual** function is not defined in the base **class** in which it is first declared, then it is referred to as a *pure virtual* function. The declaration syntax of a pure **virtual** function is:

```
virtual return_typeSpecifier MemberFunction /*...*/ = 0 ;
```

A **virtual** function is made pure by the initialiser =0. If a **class** declares one or more pure **virtual** functions it is referred to as an *abstract class*. It is illegal to define an object of an abstract **class**, although pointers of an abstract **class** can still be defined:

```
// pure&abs.cpp  
// illustrates pure virtual functions and abstract classes  
#include <iostream.h> // C++ I/O  
  
// abstract base class  
class Base  
{  
    public:  
        virtual void Display () = 0 ; // pure virtual function  
};  
  
class Derived0 : public Base  
{  
    public:  
        void Display ()  
        { cout << "Derived0::Display() called" << endl ; }  
};  
  
class Derived1 : public Base  
{  
    public:  
        void Display ()  
        { cout << "Derived1::Display() called" << endl ; }  
};  
  
void main ()  
{
```

---

```

Base* b_ptr (NULL) ;

Derived0 d0_obj ;
Derived1 d1_obj ;

Base b_obj ;           // error: abstract class objects not
                      // allowed

b_ptr = &d0_obj ;    // b_ptr pointer points to d0_obj object
b_ptr->Display () ;

b_ptr = &d1_obj ;    // b_ptr pointer points to d1_obj object
b_ptr->Display () ;
}

```

The **class** Base is an abstract **class**, since Base::Display() is declared as a pure **virtual** function:

```

class Base
{
public:
    virtual void Display () = 0 ; // pure virtual function
};

```

Note that a pure **virtual** function in a base **class** which is not defined by a derived **class** remains a pure **virtual** function in the derived **class** and therefore results in the derived **class** being an abstract **class**.

Pure **virtual** functions are useful in preventing a user from defining an object of a generic base **class**, since it is frequently meaningless to define an object of such a **class**.

To help discuss inheritance, **virtual** functions, pure **virtual** functions and abstract classes further, let's revisit an example discussed in Chapter 1 for developing a series of shape classes using inheritance. First, we require some primitive classes, such as Position, Colour and Size, to model the position, colour and size of a given shape:

```

class Position
{
protected:
    int x, y ; // (x, y) coordinates in space
public:
    Position () ;
    //...
};

class Colour
{
protected:
    int red, green, blue ; // 3 primary components of colour
public:
    Colour () ;
    //..
};

```

```
class Size
{
protected:
    int bottom_left, bottom_right, top_left, top_right ;
public:
    Size () ;
    //...
};
```

Position encapsulates integer coordinates of a point in two-dimensional space. Colour encapsulates the red, green and blue primary components of light. Size encapsulates the size of an object as a two-dimensional, planar bounding box.

With classes such as Position, Colour and Size available we are able to develop a **class** which encapsulates the general properties of all shapes:

```
class FuzzyShape
{
protected:
    Colour colour ;
    Size size ;
    Position position ;
    //...
public:
    // constructors & destructor
    FuzzyShape () ;
    //...
    virtual ~FuzzyShape () ;
    // member functions
    Position GetPosition () const ;
    void SetPosition (const Position& p) ;
    //...
    // pure virtual member functions
    virtual void Draw () = 0 ;
    //...
};
```

The FuzzyShape **class** captures the position, colour, size etc. features of all objects by declaring appropriate **protected** data members. Note that FuzzyShape is an abstract **class**, since member functions such as FuzzyShape::Draw() are declared as pure **virtual** member functions. Thus it is illegal and meaningless to define an object of **class** FuzzyShape because it defines pure **virtual** functions, and an object of FuzzyShape has no physical equivalent.

We are now in a position to derive specific shape classes from FuzzyShape using inheritance:

```
class Triangle : public FuzzyShape
{
protected:
    Position p0, p1, p2 ;
public:
    Triangle () ;
```

```
//...
~Triangle () ;
void Draw () ;
//...
};

class Rectangle : public FuzzyShape
{
protected:
    double height, width ;
public:
    Rectangle () ;
//...
~Rectangle () ;
void Draw () ;
//...
};

class Circle : public FuzzyShape
{
protected:
    double radius ;
public:
    Circle () ;
//...
~Circle () ;
void Draw () ;
//...
};
//...
```

A typical interface could be:

```
//...
void Function ()
{
    FuzzyShape* fs_array[N] ;
//...
    Triangle t_obj ;
    Rectangle r_obj ;
    Circle c_obj ;
//...
    fs_array[0] = &t_obj ;
    fs_array[1] = &r_obj ;
//...
    fs_array[N-1] = &s_obj ;
//...
// draw all shape objects using a single function call
for (int i=0; i<N; i++)
    fs_array[i]->Draw () ;
//...
```

```
}
```

### 15.19.2 An Application

To further illustrate typical applications of single and multiple inheritance, non-pure and pure **virtual** functions, and abstract classes let's revisit the Triangle, Tetrahedra and related classes previously discussed. To begin with let's declare a Shape **class** which plays the role of base **class** to all shapes:

```
// shape_v.h
// header file for Shape class

#ifndef _SHAPE_V_H // prevent multiple includes
#define _SHAPE_V_H

#include "pt.h" // class Point

class Shape
{
protected:
    int number ;
public:
    Shape ()
        : number (0) {}
    Shape (int n)
        : number (n) {}
    // destructor
    virtual ~Shape () { number = 0 ; }
    // member functions
    int& Number () { return number ; }
    const int& Number () const { return number ; }
    // pure virtual member functions
    virtual Point Centroid () = 0 ;
    virtual double Perimeter () = 0 ;
    virtual double SurfaceArea () = 0 ;
    virtual double Volume () = 0 ;
}; // class Shape

#endif // _SHAPE_V_H
```

Rather than simply encapsulating a number, **class** Shape could encapsulate a whole list of properties, such as colour, size, position and surface, as in the FuzzyShape example discussed in the previous subsection. The abstract **class** Shape declares four pure **virtual** functions, *Centroid()*, *Perimeter()*, *SurfaceArea()* and *Volume()*, since it is assumed that all shapes have these properties but that they are specific to each shape. Note the **virtual** destructor for Shape.

Another **class** which will act as a base **class** for numerous derived classes is **class** Plane:

```
// plane_v.h
// header file for Plane class
```

---

```

#ifndef _PLANE_V_H // prevent multiple includes
#define _PLANE_V_H

#include "vec3d.h" // class Vector3D
#include "pt.h" // class Point

class Plane
{
protected:
    Vector3D normal ; // A, B, C of plane
    double d ; // D of plane
public:
    // constructors
    Plane () : d (0.0) {}
    Plane (double a, double b, double c, double d) ;
    Plane (Point* p_array) ;
    // copy constructor
    Plane (const Plane& p) ;
    // destructor
    ~Plane () ;
    // member functions
    Vector3D Normal () { return normal ; }
    // overloaded operator
    Plane& operator = (const Plane& p) ;
    // friend/overloaded operator
    friend ostream& operator << (ostream& s,
                                    const Plane& p) ;
}; // class Plane

#endif // _PLANE_V_H

```

with definitions residing in PLANE\_V.CPP:

```

// plane_v.cpp
// implementation file for Plane class

#include "plane_v.h" // header file

// Ax+By+Cz+D=0
Plane::Plane (double a, double b, double c, double d_arg)
{
    normal[0] = a ; normal[1] = b ; normal[2] = c ;
    d = d_arg ;
    normal.Normalise () ;
}

// plane through three points
Plane::Plane (Point* p_array)
{
    Vector3D u, v ;

```

```

// use vertex v0 as 'local' origin
u = p_array[1] - p_array[0] ;
v = p_array[2] - p_array[0] ;

normal = u.CrossProduct (v) ;
normal.Normalise () ;
d = - (p_array[0].X()*normal[0] +
       p_array[0].Y()*normal[1] +
       p_array[0].Z()*normal[2]) ;
}

// copy constructor
Plane::Plane (const Plane& p)
: normal (p.normal), d (p.d) {}

// destructor
Plane::~Plane ()
{}

// overloaded operator:
Plane& Plane::operator = (const Plane& p)
{
    normal = p.normal ;
    d = p.d ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Plane& p)
{
    return s << "normal: " << p.normal << ", d: " << p.d ;
}

```

Plane encapsulates the characteristics of a general first-degree planar surface which is given by the following linear equation in  $x, y$  and  $z$ :

$$Ax + By + Cz + D = 0$$

where the constant coefficients  $A, B, C$  and  $D$  are specific to a particular plane in three-dimensional space. The normal vector,  $\mathbf{n}$ , to the plane is conveniently given in terms of the coefficients  $A, B$  and  $C$  ( $\mathbf{n}(A, B, C)$ ), and if the normal is normalised then  $D$  is the distance from the origin to the nearest point on the plane; see Fig. 15.10. Since the coefficients  $A, B$  and  $C$  of a plane are conveniently stored in terms of the unit normal vector to the plane, the **protected** data members of **class** Plane are:

```

class Plane
{
protected:
    Vector3D normal ;      // A, B, C of plane
    double     d ;          // D of plane
}

```

```
//...
};
```

and not:

```
class Plane
{
protected:
    double a, b, c, d; // A, B, C and D of plane
//...
};
```

The four- and one-argument constructors of `Plane` allow a plane to be defined explicitly in terms of the coefficients  $A, B, C$  and  $D$  or in terms of three points. It is worth highlighting that `Plane` is not derived from `Shape`, because an unbounded plane having an undefined centroid and infinite perimeter and surface area is not considered to be a *real* shape.

Now, let's declare a `Polygon` **class** to model a straight-edged plane polygon of arbitrary number of vertices in a three-dimensional space:

```
// poly_v.h
// header file for Polygon class

#ifndef _POLY_V_H // prevent multiple includes
#define _POLY_V_H

#include "vec3d.h" // class Vector3D
#include "pt.h" // class Point
#include "line.h" // class Line
#include "shape_v.h" // class Shape
#include "plane_v.h" // class Plane

class Polygon : public Shape, public Plane
{
private:
    void Allocate (int n) ;
protected:
    Point* verts ;
    Line* edges ;
    int vertices ;
public:
    // constructors
    Polygon (int n=3) ;
    Polygon (Point* p_array, int n_verts, int n=0) ;
    // copy constructor
    Polygon (const Polygon& p) ;
    // destructor
    ~Polygon () ;
    // member functions
    int Vertices () const { return vertices ; }
    Point& Vertex (int n) ;
    const Point& Vertex (int n) const ;
```

```

Line&      Edge (int n) ;
const Line& Edge (int n) const ;
// overridden member functions
Point     Centroid () ;
double    Perimeter () ;
double    SurfaceArea () ;
double    Volume () ;
// overloaded operator
Polygon& operator = (const Polygon& p) ;
// friends/overloaded operators
friend ostream& operator << (ostream& s,
                                const Polygon& p) ;
friend istream& operator >> (istream& s, Polygon& p) ;
}; // class Polygon

#endif // _POLY_V_H

```

and implementation file:

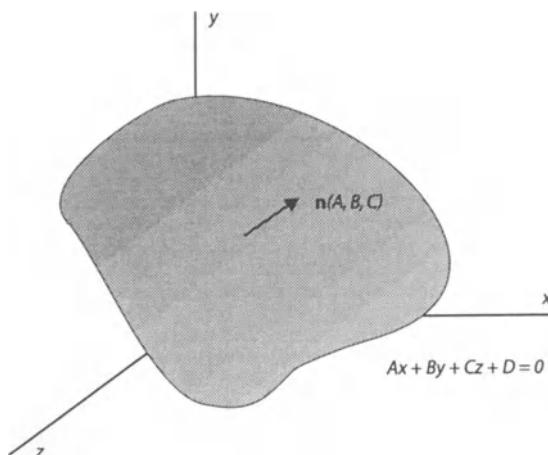
```

// poly_v.cpp
// implementation file for Polygon class

#include "poly_v.h" // header file

// allocates memory for a Polygon
void Polygon::Allocate (int n)
{
    verts = new Point[n] ;
    assert (verts != NULL) ;
    edges = new Line[n] ;
    assert (edges != NULL) ;
}

```



**Fig. 15.10** A general planar surface in a three-dimensional Cartesian coordinate system.

```
// default constructor
// default Polygon is a triangle
Polygon::Polygon (int n)
    : Shape (0), Plane (), vertices (n)
{
    Polygon::Allocate (n) ;
}

// constructor
Polygon::Polygon (Point* p_array, int n_verts, int n)
    : Shape (n), Plane (p_array), vertices (n_verts)
{
    Polygon::Allocate (n_verts) ;
    for (int i=0; i<vertices; i++)
    {
        verts[i]      = p_array[i] ;
        edges[i].P0() = p_array[i] ;
        edges[i].P1() = p_array[i+1] ;
        if (i == vertices-1) // final edge
        {
            verts[vertices-1]      = p_array[vertices-1] ;
            edges[vertices-1].P0() = p_array[vertices-1] ;
            edges[vertices-1].P1() = p_array[0] ;
        }
    }
}

// copy constructor
Polygon::Polygon (const Polygon& p)
    : Shape (p.number), Plane (p), vertices (p.vertices)
{
    Allocate (p.vertices) ;
    for (int i=0; i<vertices; i++)
    {
        verts[i] = p.verts[i] ;
        edges[i] = p.edges[i] ;
    }
}

// destructor
Polygon::~Polygon ()
{
    Plane::~Plane () ;
    Shape::~Shape () ;
    delete [] verts ;
    delete [] edges ;
    verts = NULL ;
    edges = NULL ;
    vertices = 0 ;
}
```

```
// member functions:

Point& Polygon::Vertex (int n)
{
    assert (n >= 0 && n < vertices) ;
    return verts[n] ;
}

const Point& Polygon::Vertex (int n) const
{
    assert (n >= 0 && n < vertices) ;
    return verts[n] ;
}

Line& Polygon::Edge (int n)
{
    assert (n >= 0 && n < vertices) ;
    return edges[n] ;
}

const Line& Polygon::Edge (int n) const
{
    assert (n >= 0 && n < vertices) ;
    return edges[n] ;
}

// returns the centroid of a Polygon
Point Polygon::Centroid ()
{
    Point cent ;

    for (int i=0; i<vertices; i++)
    {
        cent.X () += verts[i].X () ;
        cent.Y () += verts[i].Y () ;
        cent.Z () += verts[i].Z () ;
    }
    return Point (cent.X()/vertices, cent.Y()/vertices,
                  cent.Z()/vertices) ;
}

// returns the perimeter of a Polygon
double Polygon::Perimeter ()
{
    double perm (0.0) ;
    for (int i=0; i<vertices; i++)
        perm += edges[i].Length () ;
    return perm ;
}

// returns the surface area of a convex or concave Polygon
```

```
double Polygon::SurfaceArea ()
{
    Vector3D vi1v0, vi2v0, sum_vector ;

    for (int i=0; i<vertices; i++)
    {
        // use vertex v0 as 'local' origin
        vi1v0 = verts[i+1] - verts[0] ;
        vi2v0 = verts[i+2] - verts[0] ;
        sum_vector += vi1v0.CrossProduct (vi2v0) ;
    }
    return sum_vector.Norm () / 2.0 ;
}

// returns the volume of a Polygon
double Polygon::Volume ()
{
    return 0.0 ;
}

// overloaded operator:

Polygon& Polygon::operator = (const Polygon& p)
{
    // verify that the two Polygon's have equivalent no. of
    // vertices
    assert(vertices == p.vertices) ;
    Plane::operator = (p) ;

    for (int i=0; i<vertices; i++)
    {
        verts[i] = p.verts[i] ;
        edges[i] = p.edges[i] ;
    }
    number = p.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Polygon& p)
{
    s << "number : " << p.number << endl
    << "vertices: " ;
    for (int i=0; i<p.vertices-1; i++)
        s << p.verts[i] << ", " ;
    s << p.verts[p.vertices-1] << endl ;
    for (int j=0; j<p.vertices-1; j++)
        s << p.edges[i] << ", " ;
    s << p.edges[p.vertices-1] << endl ;
    return s ;
}
```

```

// friend/overloaded extraction operator >>
istream& operator >> (istream& s, Polygon& p)
{
    int n_verts (0), n (0) ;

    s >> ws >> n_verts ;

    Point* pa = new Point[n_verts] ;

    for (int i=0; i<n_verts; i++)
        s >> ws >> pa[i] ;
    s >> ws >> n ;

    p = Polygon (pa, n_verts, n) ;
    delete [] pa ;

    return s ;
}

```

Polygon steals many of its features from the Triangle **class**, discussed in previous sections, which was developed to model a straight-edge planar triangle in three-dimensional space. As we shall see shortly, a Polygon **class** greatly simplifies the development of a whole range of planar shapes, such as triangles and quadrilaterals. Also, by ‘taking a step backwards’ and developing a Polygon **class** we shall eliminate a great deal of repetition in program code, and more importantly help to categorise similar characteristics of different shapes. For instance, the *Vertex()* and *Edge()* member functions are not overloaded for classes Triangle and Quadrilateral but are inherited from Polygon.

Multiple inheritance is used to derive **class** Polygon from Shape and Plane:

```

class Polygon : public Shape, public Plane
{
private:
    void Allocate (int n) ;
protected:
    Point* verts ;
    Line* edges ;
    int vertices ;
//...
};

```

The **protected** data members *verts* and *edges* are pointers to arrays of the vertices and edges of a polygon while *vertices* is the number of vertices of the polygon. The default Polygon object is a triangle and is described by the one-argument constructor, whereas the three-argument constructor describes a polygon assigned the number *n* of *n\_verts* vertices given in the Point array *p\_array*:

```
Polygon (Point* p_array, int n_verts, int n=0) ;
```

If a Vector **template class** had been used in preference to C++ arrays we could alternatively have written:

---

```
Polygon (Vector<Point> p, int n=0) ;
```

The overridden member functions *Centroid()*<sup>4</sup>, *Perimeter()* and *Volume()* are fairly straightforward and similar to those developed for **class** Triangle, but *SurfaceArea()* deserves a closer examination:

```
double Polygon::SurfaceArea ()
{
    Vector3D vi1v0, vi2v0, sum_vector ;

    for (int i=0; i<vertices; i++)
    {
        // use vertex v0 as 'local' origin
        vi1v0 = verts[i+1] - verts[0] ;
        vi2v0 = verts[i+2] - verts[0] ;
        sum_vector += vi1v0.CrossProduct (vi2v0) ;
    }
    return sum_vector.Norm () / 2.0 ;
}
```

With reference to Fig. 15.11(a) and Triangle::SurfaceArea() we know that the area,  $A$ , of a triangle described by the three non-collinear points  $p_0, p_1$  and  $p_2$  is given by:

$$A = \frac{1}{2}(p_1 - p_0) \times (p_2 - p_0)$$

where point  $p_0$  is taken as a reference vertex. Similarly, the area of a general convex or concave polygon of  $n$  vertices is given by:

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (p_{i+1} - p_0) \times (p_{i+2} - p_0)$$

since there are  $n-2$  signed triangular areas for an  $n$ -vertex polygon; see Fig. 15.11(b).

Deriving **class** Triangle from Polygon is now easily implemented as:

```
// tri_v.h
// header file for Triangle class

#ifndef _TRI_V_H // prevent multiple includes
#define _TRI_V_H

#include "pt.h"      // class Point
#include "poly_v.h"  // class Plane

class Triangle : public Polygon
{
public:
```

---

<sup>4</sup> Beware of using the present algorithm (i.e. taking the mean of the  $x$ -,  $y$ - and  $z$ -coordinates of the polygon vertices) for determining the centroid of a polygon when clustering of vertices exists (Bashein and Detmer, 1994).

```
// constructors
Triangle () ;
Triangle (Point* p_array, int n=0) ;
// copy constructor
Triangle (const Triangle& t) ;
// destructor
~Triangle () ;
// overloaded operator
Triangle& operator = (const Triangle& t) ;
// friend/overloaded operator
friend istream& operator >> (istream& s, Triangle& t) ;
} // class Triangle

#endif // _TRI_V_H

// tri_v.cpp
// implementation file for Triangle class
#include "tri_v.h" // header file

// constructors:
Triangle::Triangle ()
: Polygon () {}

Triangle::Triangle (Point* p_array, int n)
: Polygon (p_array, 3, n) {}

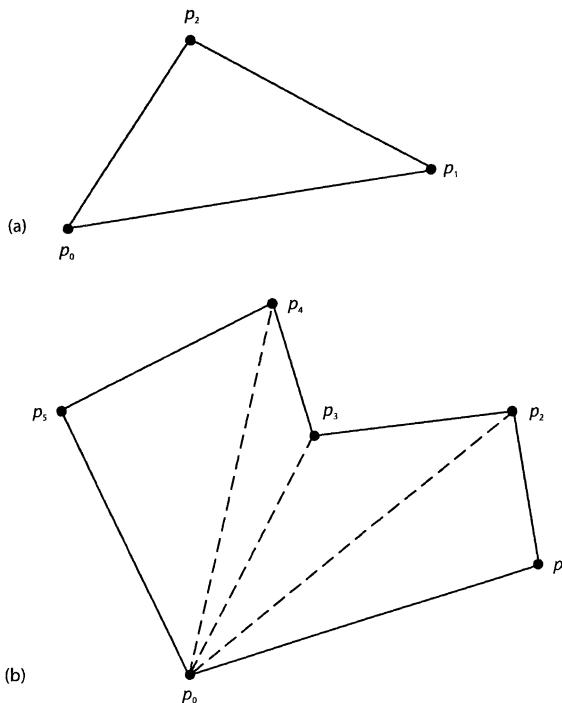
// copy constructor
Triangle::Triangle (const Triangle& t)
: Polygon (t) {}

// destructor
Triangle::~Triangle ()
{
    Polygon::~Polygon () ;
}

// overloaded operator:

// assignment operator =
Triangle& Triangle::operator = (const Triangle& t)
{
    Polygon::operator = (t) ;
    return *this ;
}

// friend/overloaded extraction operator >>
istream& operator >> (istream& s, Triangle& t)
{
    Point* pa = new Point[3] ;
    int number (0) ;
```



**Fig. 15.11** (a) A three-vertex polygon. (b) A six-vertex concave polygon consisting of four triangles.

```

s >> ws >> pa[0] >> pa[1] >> pa[2] >> number ;
t.verts[0] = pa[0] ; t.verts[1] = pa[1] ; t.verts[2] = pa[2] ;
t.number = number ;

delete [] pa ;

t.edges[0].P0() = t.verts[0] ; t.edges[0].P1() = t.verts[1] ;
t.edges[1].P0() = t.verts[1] ; t.edges[1].P1() = t.verts[2] ;
t.edges[2].P0() = t.verts[2] ; t.edges[2].P1() = t.verts[0] ;

return s ;
}

```

Note that the extraction operator is overloaded specifically for **class** Triangle, whereas **operator<<**(ostream, Polygon) is used by Triangle. A similar procedure is adopted for **class** Quadrilateral:

```

// quad_v.h
// header file for Quadrilateral class

#ifndef _QUAD_V_H // prevent multiple includes
#define _QUAD_V_H

```

```
#include "pt.h"          // class Point
#include "poly_v.h"      // class Polygon

class Quadrilateral : public Polygon
{
public:
    // constructors
    Quadrilateral () ;
    Quadrilateral (Point* p_array, int n=0) ;
    // copy constructor
    Quadrilateral (const Quadrilateral& q) ;
    // destructor
    ~Quadrilateral () ;
    // overloaded operator
    Quadrilateral& operator = (const Quadrilateral& q) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s,
                                    Quadrilateral& q) ;
}; // class Quadrilateral

#endif // _QUAD_V_H
```

and the implementation file for **class** Quadrilateral:

```
// quad_v.cpp
// implementation file for Quadrilateral class

#include "quad_v.h" // header file

// constructors:
Quadrilateral::Quadrilateral ()
    : Polygon (4) {}

Quadrilateral::Quadrilateral (Point* p_array, int n)
    : Polygon (p_array, 4, n) {}

// copy constructor
Quadrilateral::Quadrilateral (const Quadrilateral& q)
    : Polygon (q) {}

// destructor
Quadrilateral::~Quadrilateral ()
{
    Polygon::~Polygon () ;
}

// overloaded operator:
// assignment operator =
```

```

Quadrilateral& Quadrilateral::operator =
    (const Quadrilateral& q)
{
    Polygon::operator = (q) ;
    return *this ;
}

// friend/overloaded extraction operator >>
istream& operator >> (istream& s, Quadrilateral& q)
{
    Point* pa = new Point[4] ;
    int number (0) ;

    s >> pa[0] >> pa[1] >> pa[2] >> pa[3]
    >> number ;

    q.verts[0] = pa[0] ; q.verts[1] = pa[1] ;
    q.verts[2] = pa[2] ; q.verts[3] = pa[3] ;
    q.number = number ;

    delete [] pa ;

    q.edges[0].P0() = q.verts[0] ; q.edges[0].P1() = q.verts[1] ;
    q.edges[1].P0() = q.verts[1] ; q.edges[1].P1() = q.verts[2] ;
    q.edges[2].P0() = q.verts[2] ; q.edges[2].P1() = q.verts[3] ;
    q.edges[3].P0() = q.verts[3] ; q.edges[3].P1() = q.verts[0] ;

    return s ;
}

```

A Circle **class** which encapsulates two data members **centre** and **radius** and derived from Shape and Plane is declared as:

```

// circ_v.h
// header file for Circle class

#ifndef _CIRC_V_H // prevent multiple includes
#define _CIRC_V_H

#include <math.h>      // atan()
#include "pt.h"         // class Point
#include "shape_v.h"    // abstract class Shape
#include "plane_v.h"    // class Plane

class Circle : public Shape, public Plane
{
protected:
    Point centre ;
    double radius ;
public:
    Circle ()

```

```

        : Shape (), Plane (), centre (), radius (0.0) {}
Circle (double a, double b, double c, double d,
        const Point& cent, double r, int n=0)
        : Shape (n), Plane (a, b, c, d),
          centre (cent), radius (r) {}
// copy constructor
Circle (const Circle& c)
        : Shape (c.number),
          Plane (c.normal[0], c.normal[1],
                  c.normal[2], c.d),
          centre (c.centre), radius (c.radius) {};
// member functions
double& Radius () { return radius; }
const double& Radius () const { return radius; }
Point& Centre () { return centre; }
const Point& Centre () const { return centre; }
// overridden member functions
Point Centroid () ;
double Perimeter () ;
double SurfaceArea () ;
double Volume () ;
// overloaded operator
Circle& operator = (const Circle& c) ;
// friends/overloaded operators
friend ostream& operator << (ostream& s,
                                const Circle& c) ;
friend istream& operator >> (istream& s, Circle& c) ;
}; // class Circle

#endif // _CIRC_V_H

```

The four-argument constructor of `Circle` defines a `Circle` object via the  $A, B, C$  and  $D$  coefficients of a plane, a centre point, a radius and an assigned number. The definition file for `class Circle` is:

```

// circ_v.cpp
// implementation file for Circle class
#include "circ_v.h" // header file

// member functions:

Point Circle::Centroid ()
{
    return centre;
}

double Circle::Perimeter ()
{
    return 2.0 * 4.0 * atan (1.0) * radius;
}

```

```

double Circle::SurfaceArea ()
{
    return 4.0 * atan (1.0) * radius * radius ;
}

double Circle::Volume ()
{
    return 0.0 ;
}

// overloaded operator:

Circle& Circle::operator = (const Circle& c)
{
    Plane::operator = (c) ;
    centre = c.centre ; radius = c.radius ;
    number = c.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Circle& c)
{
    return s << "number: " << c.number << ", centre: "
        << c.centre << ", radius: " << c.radius ;
}

// friend/overloaded extraction operator >>
istream& operator >> (istream& s, Circle& c)
{
    return s >> ws
        >> c.normal[0] >> c.normal[1] >> c.normal[2] >> c.d
        >> c.centre >> c.radius >> c.number ;
}

```

The perimeter (circumference) and area of a circle are given by  $2\pi r$  and  $\pi r^2$  respectively, with  $\pi=4\tan^{-1}1$ .

Finally, a Tetrahedra **class** is derived from Shape and uses containment to represent its four Triangle faces:

```

// tet_v.h
// header file for Tetrahedra class

#ifndef _TET_V_H // prevent multiple includes
#define _TET_V_H

#include "shape_v.h" // abstract class Shape
#include "tri_v.h" // class Triangle

class Tetrahedra : public Shape
{

```

```

private:
    void Allocate () ;
protected:
    Triangle* faces ;
public:
    // constructors
    Tetrahedra () ;
    Tetrahedra (const Triangle& t0, const Triangle& t1,
                const Triangle& t2, const Triangle& t3,
                int n=0) ;
    // copy constructor
    Tetrahedra (const Tetrahedra& t) ;
    // destructor
    ~Tetrahedra () ;
    // member functions
    Triangle& Face (int n) ;
    const Triangle& Face (int n) const ;
    Point* Vertices () ;
    // overridden member functions
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    double Volume () ;
    // overloaded operator
    Tetrahedra& operator = (const Tetrahedra& t) ;
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const Tetrahedra& t) ;
    friend istream& operator >> (istream& s, Tetrahedra& t) ;
}; // class Tetrahedra

#endif // _TET_V_H

```

The implementation file for **class** Tetrahedra is similar to that of TTT\_CON.CPP:

```

// tet_v.cpp
// implementation file for Tetrahedra class
#include "tet_v.h" // header file

// allocates memory for a Tetrahedra
void Tetrahedra::Allocate ()
{
    faces = new Triangle[4] ;
    assert (faces != NULL) ;
}

// constructors:

Tetrahedra::Tetrahedra ()
    : Shape ()
{

```

```

Allocate () ;
}

Tetrahedra::Tetrahedra (const Triangle& t0, const Triangle& t1,
                        const Triangle& t2, const Triangle& t3,
                        int n)
: Shape (n)
{
Allocate () ;
faces[0] = t0 ; faces[1] = t1 ;
faces[2] = t2 ; faces[3] = t3 ;
}

// copy constructor
Tetrahedra::Tetrahedra (const Tetrahedra& t)
{
Allocate () ;
faces[0] = t.faces[0] ; faces[1] = t.faces[1] ;
faces[2] = t.faces[2] ; faces[3] = t.faces[3] ;
number = t.number ;
}

// destructor
Tetrahedra::~Tetrahedra ()
{
Shape::~Shape () ;
delete [] faces ;
faces = NULL ;
}

// member functions:

Triangle& Tetrahedra::Face (int n)
{
assert (n >= 0 && n < 4) ;
return faces[n] ;
}

const Triangle& Tetrahedra::Face (int n) const
{
assert (n >= 0 && n < 4) ;
return faces[n] ;
}

// return a pointer to the four vertices of a Tetrahedra
Point* Tetrahedra::Vertices ()
{
Point* v_array = new Point[4] ;

// use faces[0] as base Triangle
v_array[0] = faces[0].Vertex (0) ;
}

```

```

v_array[1] = faces[0].Vertex (1) ;
v_array[2] = faces[0].Vertex (2) ;

// find fourth vertex
Point pt4 ;

for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
        if ( (pt4 = faces[1].Vertex(i))!=(faces[0].Vertex(j)) )
            break ; // found 4th. vertex
v_array[3] = pt4 ;

return v_array ;
}

// returns the centroid of a Tetrahedra
Point Tetrahedra::Centroid ()
{
    Point cent ; // centroid
    Point* verts = Vertices () ; // get 4 vertices of Tetrahedra

    cent.X() = (verts[0].X()+verts[1].X()+verts[2].X()+
                verts[3].X())/4.0 ;
    cent.Y() = (verts[0].Y()+verts[1].Y()+verts[2].Y()+
                verts[3].Y())/4.0 ;
    cent.Z() = (verts[0].Z()+verts[1].Z()+verts[2].Z()+
                verts[3].Z())/4.0 ;
    delete [] verts ; // delete memory associated with 4 vertices
    return cent ;
}

// returns the perimeter of a Tetrahedra
// uses vertices of a Tetrahedra to find the perimeter
// length since a Tetrahedra edge is common to 2 Triangles
double Tetrahedra::Perimeter ()
{
    Point* verts = Vertices () ; // get 4 vertices of Tetrahedra
    double perm = (verts[1]-verts[0]).Norm () +
                  (verts[2]-verts[0]).Norm () +
                  (verts[3]-verts[0]).Norm () +
                  (verts[3]-verts[1]).Norm () +
                  (verts[2]-verts[1]).Norm () +
                  (verts[3]-verts[2]).Norm () ;
    delete [] verts ; // delete memory associated with 4 vertices
    return perm ;
}

// returns the surface area of a Tetrahedra
double Tetrahedra::SurfaceArea ()
{
    return (faces[0].SurfaceArea () + faces[1].SurfaceArea () +

```

```

        faces[2].SurfaceArea () + faces[3].SurfaceArea ()) ;
    }

// returns the volume of a Tetrahedra
double Tetrahedra::Volume ()
{
    Point* verts = Vertices () ; // get 4 vertices of Tetrahedra
    // vol=(1/6)/a.(bxc) /
    // use vertex v0 as 'local' origin
    return fabs ( (verts[1]-verts[0]).DotProduct ( (verts[2]-
        verts[0]).CrossProduct (verts[3]-verts[0]) )
        ) / 6.0 ;
}

// overloaded operator:

// assignment operator =
Tetrahedra& Tetrahedra::operator = (const Tetrahedra& t)
{
    faces[0] = t.faces[0] ; faces[1] = t.faces[1] ;
    faces[2] = t.faces[2] ; faces[3] = t.faces[3] ;
    number = t.number ;
    return *this ;
}

// friend/overloaded insertion operator <<
ostream& operator << (ostream& s, const Tetrahedra& t)
{
    s << "number : " << t.number << endl
    << "faces : " << endl << t.faces[0] << endl
                                << t.faces[1] << endl
                                << t.faces[2] << endl
                                << t.faces[3] ;
    return s ;
}

// friend/overloaded extraction operator >>
istream& operator >> (istream& s, Tetrahedra& t)
{
    int      number (0) ;
    Point* pa = new Point[4] ;

    s >> ws >>
        >> pa[0] >> pa[1] >> pa[2] >> pa[3]
        >> number ;

    Point pa0[3] = {pa[0],pa[1],pa[2]} ;
    Point pa1[3] = {pa[0], pa[1],pa[3]} ;
    Point pa2[3] = {pa[1],pa[2],pa[3]} ;
    Point pa3[3] = {pa[2], pa[0],pa[3]} ;
}

```

```

delete [] pa ;

Triangle face0 (pa0) ; Triangle face1 (pa1) ;
Triangle face2 (pa2) ; Triangle face3 (pa3) ;

t = Tetrahedra (face0, face1, face2, face3, number) ;

return s ;
}

```

Note that `Tetrahedra` could have been derived from a more general `Polyhedra class`.

The classes presented above form the `class` hierarchy shown in Fig. 15.12. A program which tests the run-time polymorphic properties of the above classes is given in `SHAPES.CPP`:

```

// shapes.cpp
// tests abstract base class Shape and derived classes

#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()
#include "shape_v.h" // abstract class Shape
#include "circ_v.h" // class Circle
#include "quad_v.h" // class Quadrilateral
#include "tri_v.h" // class Triangle

void main ()
{
    Shape* s (NULL) ;
    Circle* c (NULL) ; Polygon* p (NULL) ;
    Quadrilateral* q (NULL) ; Triangle* t (NULL) ;

    char ch ('x') ;

    cout << "enter c, p, q or t for circle, polygon,
            quadrilateral or triangle: " ;
    cin >> ch ;

    switch (ch)
    {
        case 'c':
            c = new Circle ;
            cout << "enter centre properties: " ;
            cin >> *c ;
            s = c ;
            break ;
        case 'p':
            p = new Polygon ;
            cout << "enter polygon properties: " ;
            cin >> *p ;
            s = p ;
            break ;
        case 'q':

```

```

q = new Quadrilateral ;
cout << "enter quadrilateral properties: " ;
cin >> *q ;
s = q ;
break ;
case 't':
    t = new Triangle ;
    cout << "enter triangle properties: " ;
    cin >> *t ;
    s = t ;
    break ;
default:
    cout << "enter c, p, q or t" << endl ;
    exit (EXIT_SUCCESS) ;
}

cout << "number      : " << s->Number ()           << endl
    << "centroid   : " << s->Centroid ()        << endl
    << "perimeter: " << s->Perimeter ()       << endl
    << "area       : " << s->SurfaceArea ()     << endl
    << "volume     : " << s->Volume ()          << endl ;

delete c ; delete p ; delete q ; delete t ;
delete s ;
}

```

A user is prompted to enter the properties of a circle, polygon, quadrilateral or triangle. The cout statement which displays the characteristics of a given shape makes use of a pointer to the abstract **class** Shape, and is thus independent of any specific shape. Some user interaction is:

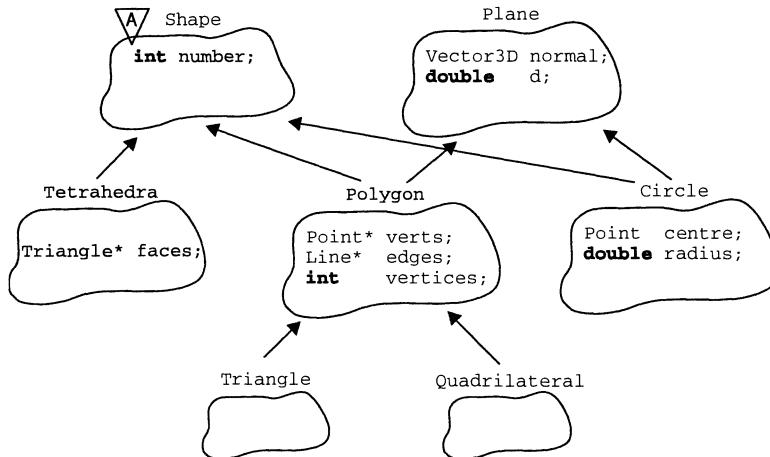
```

enter c, p, q or t for circle, polygon, quadrilateral or
triangle: c
enter circle properties: 0 -1 0 1 1 0 0 1 5
number      : 5
centroid   : (1, 0, 0)
perimeter: 6.28319
area       : 3.14159
volume     : 0

```

which illustrates various properties for a circle lying in a plane characterised by the plane coefficients  $A=0$ ,  $B=-1$ ,  $C=0$  and  $D=1$  with centre at the point  $(1, 0, 0)$ , radius 1 and number 5.

The classes presented above are far from ideal, but should have demonstrated just how powerful inheritance is in the C++ language and object-oriented programming, and how naturally the topic of computer graphics lends itself to inheritance. A possible modification and extension of the Shapes **class** hierarchy is shown in Fig. 15.13. By recognising that a polygon can have straight or curved edges and lie on either a planar or complex three-dimensional surface, **class** Polygon is now derived from Shape, with PlanarPolygon derived from Polygon. In addition to defining a polygon in terms of vertices and edges, you may need to associate a surface with the polygon. The surface could be defined either in terms of a surface constructed of objects of the Shapes **class** hierarchy or a surface defined by a given function.



**Fig. 15.12** Class hierarchy of the *Shapes* program.

A Mesh **class** hierarchy is also shown in Fig. 15.13 to enable a Shapes surface to be constructed. Mesh encapsulates a pointer to abstract base **class** Shape so that an array of similar or different Shapes objects can be used to construct a mesh. A Polyhedra **class** has been inserted into the Shapes **class** hierarchy to model a solid body consisting of polygon faces. Tetrahedra is now derived from Polyhedra. A SuperQuadrics **class** has been added to the Shapes **class** hierarchy. Quadric surfaces include spheres, ellipsoids and tori. A superquadric is a generalisation of a quadric which allows complex surfaces and solids to be defined by incorporating additional parameters in a quadric equation. For instance, the equation of a superellipse is given by:

$$\left(\frac{x}{r_x}\right)^{2/s} + \left(\frac{y}{r_y}\right)^{2/s} = 1$$

where  $s$  is real and non-zero. When  $s=1$  and  $r_x=r_y$  we have an ellipse, and when  $s=1$  and  $r_x=r_y$  we have a circle.

I hope Fig. 15.13 has demonstrated just how complex **class** hierarchies can become. Even in its simplest implementation, modifying the **class** hierarchy shown in Fig. 15.12 to that of Fig. 15.13 requires a lot of work in restructuring **class** declarations, data members and member functions. Therefore – and this is very important – when designing and implementing a **class** hierarchy ensure that you spend a significant amount of time and effort in the design stage to make your **class** hierarchy as general and accommodating to change as possible.

## 15.20 Templates and Inheritance

The use of **template** classes and inheritance is similar to that of non-**template** classes and inheritance:

```
// tmp_inh.cpp
// illustrates the use of template classes and inheritance
```

```

#include <iostream.h> // C++ I/O

class X
{
};

// template class derived from
// non-template base class
template <class T>
class A : public X
{
protected:
    T a_data ;
public:
    A ()
        : a_data (T()) {}
};

// class derived from a
// template base class
class B : public A<int>
{
protected:
    int b_data ;
public:
    B ()
        : A<int> (), b_data (0) {}
};

// template class derived from
// template base class
template <class T>
class C : public A<T>
{
protected:
    T c_data ;
public:
    C ()
        : A<T> (), c_data (T()) {}
};

void main ()
{
}

```

Let's develop a RVector **template class** which is derived from **template class** Vector. RVector allows a user to specify an index range for the elements of an RVector object rather than relying on the  $0:n-1$  rule. The **template** Vector **class** declared in the RV\_TMP.H header file below is identical to the **template** Vector **class** declared in the header file VEC\_ITER.H of Chapter 13, except that the data members of Vector<T> are now **protected** rather than **private**:

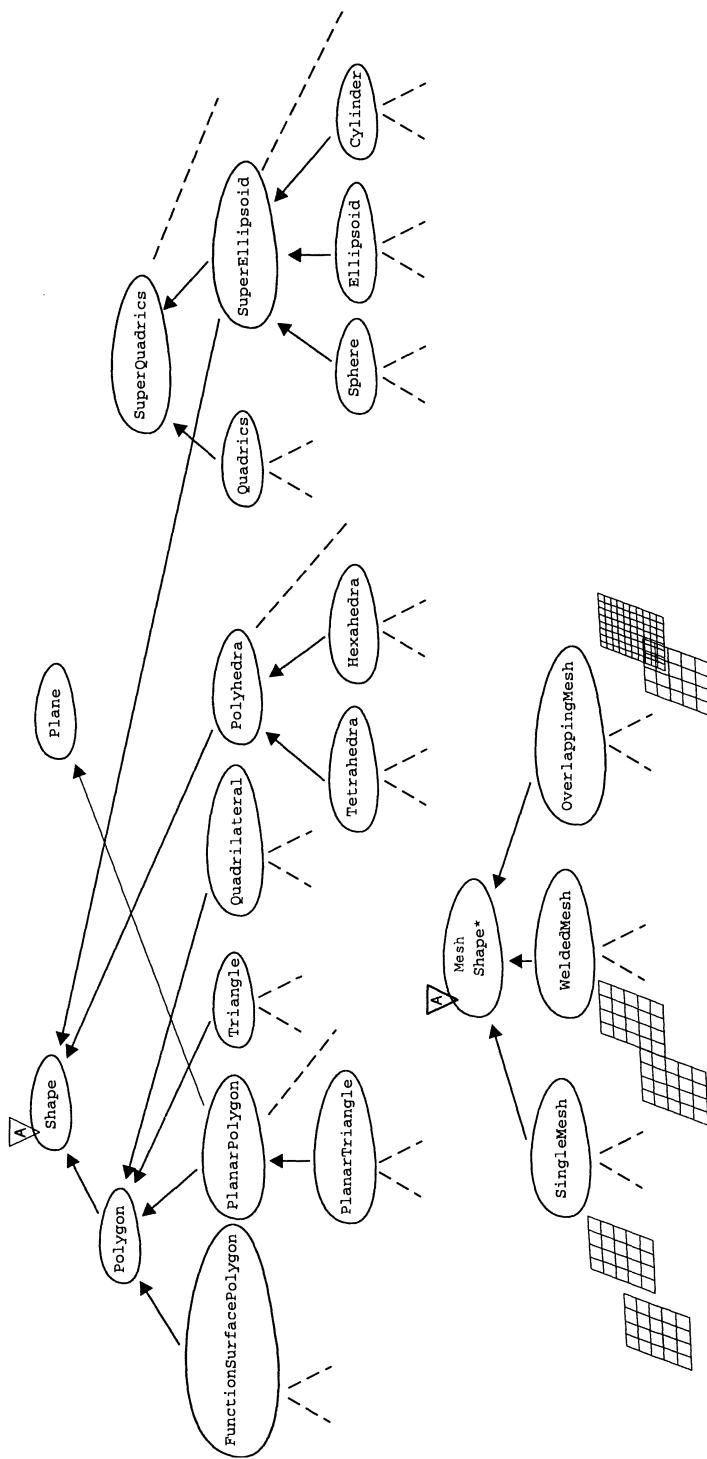


Fig. 15.13 Possible extension of the `Shapes` class hierarchy.

```
// rv_tmp.h
// template class RVector
#ifndef _RV_TMP_H // prevent multiple includes
#define _RV_TMP_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...
#include <assert.h> // assert()

enum Boolean { FALSE, TRUE };

// template class Vector
template <class T>
class Vector
{
protected:
    // data members
    T* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector ()
        : array (NULL), n_elements (0) {}
    Vector (int n, T obj) ;
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                 n_elements = 0 ; }
    // member functions
    int NumberElements () const { return n_elements ; }
    T& Value (int index) ;
    const T& Value (int index) const ;
    void New (int new_n) ;
    void ForEachElement (void (*fp)(T&), int first,
                         int last) ;
    int FirstElement (Boolean (*fp)(const T&),
                      int first, int last) const ;
    int LastElement (Boolean (*fp)(const T&),
                     int first, int last) const ;
    // overloaded operators
    Vector& operator = (const Vector& v) ;
    T& operator [] (int index) ;
    const T& operator [] (int index) const ;
    Vector operator + (const Vector& v) ;
    Vector operator - (const Vector& v) ;
    Vector operator * (const Vector& v) ;
    // friend
    friend ostream& operator << (ostream& s,
```

```

        const Vector<T>& v) ;
    }; // template class Vector
//...
// template class RVector
template <class T>
class RVector : public Vector<T>
{
private:
    int lowerbound ;
public:
    RVector ()
        : Vector<T> (), lowerbound (0) {}
    RVector (int upper, int lower, T obj)
        : Vector<T> (upper-lower+1, obj),
          lowerbound (lower) {}
    // member functions
    int Lower () const { return lowerbound ; }
    int Upper () const { return lowerbound +
                           NumberElements () - 1 ; }
    int& operator [] (int index)
        { return Vector<T>::operator [] (index - lowerbound) ; }
}; // template class RVector

#endif // _RV_TMP_H

```

A program which tests the above **template class** RVector is:

```

// rv_tst.cpp
// tests the template class RVector which
// is inherited from template class Vector
#include <iostream.h> // C++ I/O
#include "rv_tmp.h" // template class RVector

void main ()
{
    RVector<int> rv (5, 1, 0) ; // indexing from 1!

    cout << rv ;
}

```

with output:

```
[0, 0, 0, 0, 0]
```

Object rv of **template class** RVector has five elements of type **int**, whose indexes range from 1 to 5 rather than 0 to 4.

It was noted when we discussed non-**template** base and derived classes that C++ implicitly converts from a derived **class** object to a base **class** object, but not vice versa. A similar rule applies in the case of **template** base and derived classes providing both base and derived **template** classes operate on the same type or **class**:

---

```

// convert.cpp
// illustrates inheritance, template classes and conversion
#include <iostream.h> // C++ I/O

class B
{
};

class D : public B
{
};

template <class T>
class TB
{
};

template <class T>
class TD : public TB<T>
{
};

void main ()
{
    B b_obj ;
    D d_obj ;

    b_obj = d_obj ;      // o.k.: slice Derived to Base
    d_obj = b_obj ;      // error: can't convert Base to Derived

    TB<int> tb_iobj ;
    TD<int> td_iobj ;

    tb_iobj = td_iobj ; // o.k.: slice TD<int> to TB<int>
    td_obj = tb_obj ;   // error: can't convert TB<int> to TD<int>

    TD<float> td_fobj ;

    tb_iobj = td_fobj ; // error: can't convert TD<float> to TB<int>
}

```

Let's now derive a **class** `PtrVector<T>`<sup>5</sup> from `Vector<void*>` which will enable the elements of an object of **class** `PtrVector<T>` to hold the memory addresses of pointers to type `T`:

```

// vec_ptr.h
// Vector of pointers template class

```

---

<sup>5</sup> `PtrVector` is similar in principle to the `ptrdynarray` C++ standard library **template class** proposed in the draft C++ standard.

```

#ifndef _VEC_PTR_H // prevent multiple includes
#define _VEC_PTR_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...
#include <assert.h> // assert()

enum Boolean { FALSE, TRUE };

// template class Vector
template <class T>
class Vector
{
    //...
};

// template class PtrVector : public Vector<void*>
public:
    // constructor
    PtrVector (int i, T obj)
        : Vector<void*> (i, obj) {}

    // member function
    T*& Value (int index)
        {return (T*)&Vector<void*>::Value (index);}

    // overloaded operator
    T*& operator [] (int index)
        {return (T*)&Vector<void*>::operator [] (index);}

};

#endif // _VEC_PTR_H

```

Note the necessary casting in the member functions `PtrVector<T>::Value()` and `PtrVector<T>::operator[]()` from `void*` to `T*`.

A program which tests the `template class VectorPtr` is:

```

// ptrv_tst.cpp
// tests the template class PtrVector which
// is inherited from template class Vector
#include <iostream.h> // C++ I/O
#include "vec_ptr.h" // template class Vector

void main ()
{
    int i (0);
    int* i_ptr = &i;

    cout << "lvalue i      :" << &i

```

```

    << ", rvalue i      : " << i          << endl ;
cout << "lvalue i_ptr: " << &i_ptr
    << ", rvalue i_ptr: " << i_ptr   << endl ;

Vector<int*> piv (3, 0) ;
cout << "piv : " << piv << endl ;
piv[0] = i_ptr ;
cout << "piv : " << piv << endl ;

PtrVector<int*> pipv (3, 0) ;
cout << "pipv: " << pipv << endl ;
pipv[0] = &i_ptr ;
cout << "pipv: " << pipv << endl ;
}

```

The output of PTRV\_TST.CPP is:

```

lvalue i      : 0x31ef26b8, rvalue i: 0
lvalue i_ptr: 0x31ef26b4, rvalue i_ptr: 0x31ef26b8
piv : [0x00000000, 0x00000000, 0x00000000]
piv : [0x31ef26b8, 0x00000000, 0x00000000]
pipv: [0x00000000, 0x00000000, 0x00000000]
pipv: [0x31ef26b4, 0x00000000, 0x00000000]

```

As demonstrated in Chapter 13, the elements of a `Vector<T*>` object are the rvalues of pointers to type T. The elements of a `PtrVector<T*>` object are the lvalues of pointers to type T.

### 15.20.1 A `SortedVector` class

As a further illustration of the combined use of templates and inheritance, let's examine a `SortedVector` **template class** which is **publicly** derived from **template class** `Vector` and maintains a sorted array of elements:

```

// vec_sort.h
// template class SortedVector

#ifndef _VEC_SORT_H // prevent multiple includes
#define _VEC_SORT_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(),...
#include <assert.h> // assert()

enum Boolean { FALSE, TRUE };

// swaps two objects
template <class T>
void Swap (T& a, T& b)
{
    T temp ;

```

```
temp = a ;
a = b ;
b = temp ;
}

// template class Vector
template <class T>
class Vector
{
protected:
    // data members
    T* array ;
    int n_elements ;
    //...
//...
public:
    //...
    void New (int new_n) ;
    //...
    void Sort () const ;
    int Find (const T& obj) const ;
    //...
}; // template class Vector
//...
// change number of elements, maintaining old data
template <class T>
void Vector<T>::New (int new_n)
{
    assert (new_n > n_elements) ;

    T* temp_array = new T[new_n] ;
    assert (temp_array != NULL) ;

    for (int i=0; i<new_n; i++) // initialise temp. array
        *(temp_array+i) = T () ;

    for (int j=0; j<n_elements; j++) // copy over old data
        *(temp_array+j) = *(array+j) ;

    delete [] array ;
    array      = temp_array ;
    n_elements = new_n ;
}
//...
// sorts a Vector object
// performs a bubble sort
template <class T>
void Vector<T>::Sort () const
{
    for (int i=0; i<n_elements-1; i++)
        for (int j=n_elements-1; i<j; j--)
```

```

        if (array[j] < array[j-1])
            Swap (array[j], array[j-1]) ;
    }

// returns the first index of a given Vector object, else -1
// linear search
template <class T>
int Vector<T>::Find (const T& obj) const
{
    for (int i=0; i<n_elements; i++)
        if (array[i] == obj)
            return i ;
    return -1 ;
}
//...
// template class SortedVector
template <class T>
class SortedVector : public Vector<T>
{
    public:
        // constructors
        SortedVector ()
            : Vector<T> () {}
        SortedVector (int n, T obj)
            : Vector<T> (n, obj) {}
        SortedVector (const Vector<T>& v)
            : Vector<T> (v) { Sort () ; }
        // copy constructor
        SortedVector (const SortedVector& sv)
            : Vector<T> (sv) {}
        // member functions
        void Add (const T& obj) ;
        int Find (const T& obj) const ;
}; // template class SortedVector

// SortedVector:

// adds an element to SortedVector<T>
template <class T>
void SortedVector<T>::Add (const T& obj)
{
    // increase no. of elements by 1
    New (n_elements+1) ;

    int insert (0) ;
    while (obj > array[insert] && insert < n_elements-1)
        insert++ ;
    if (insert < n_elements-1)
        for (int i=n_elements-1; i>insert; i--)
            array[i] = array[i-1] ;
    array[insert] = obj ;
}

```

```

}

// returns the first index of a given Vector object, else -1
// binary search
template <class T>
int SortedVector<T>::Find (const T& obj) const
{
    int first (0), last (n_elements - 1), middle (0) ;

    while (first <= last)
    {
        middle = (first + last) / 2 ;

        if (obj == array[middle]) // found & return index
            return middle ;
        else if (obj < array[middle])
            last = middle - 1 ;
        else
            last = middle + 1 ;
    }
    return -1 ; // not found
}

#endif // _VEC_SORT_H

```

Two new member functions, *Sort()* and *Find()*, have been added to **template class** *Vector*. The *Sort()* member function uses the bubblesort algorithm to sort the elements of a *Vector*, while *Find()* performs a linear search to return the first index of a *Vector* object element which matches a supplied object.

*SortedVector* supports a default constructor, a two-argument constructor to initialise all the elements of a *SortedVector* object to a given object and a one-argument constructor which enables a *SortedVector* object to be initialised using the elements of a *Vector* object:

```

SortedVector (const Vector<T>& v)
: Vector<T> (v) { Sort () ; }

```

The constructor calls the inherited *Vector<T>::Sort()* function to ensure that upon definition of a *SortedVector* object that the object's elements are sorted.

The *SortedVector<T>::Add()* member function is slightly more tricky in that it allows an element to be added to a *SortedVector* object while simultaneously maintaining a sorted array of object elements. First, the size of a *SortedVector* object is increased to accommodate the addition of a new element. Then the point of insertion of the new element is determined, and if this insertion point is not the last element position the existing element objects are transferred to their respective new positions.

The *SortedVector<T>::Find()* member function overrides the *Vector<T>::Find()* member function so that a binary search rather than a linear search algorithm can be used when searching for a given object in a sorted list of elements. The binary search algorithm is a popular algorithm for searching a sorted list and begins by comparing the search object with the element in the middle of the list of elements. If the search object is less than or greater than the middle list element the procedure is repeated for the first or second

half sublists, respectively, of the list until the search object is either found or we establish that the search object is not present in the list. The binary search algorithm is much faster than a linear search algorithm, since the number of remaining elements to be searched in the list is approximately halved for each iteration<sup>6</sup>. For a more detailed discussion of the bubblesort and binary search algorithms in C++, refer to Budd (1994) and Horowitz *et al.* (1995).

The **template class** `SortedVector` is tested in the following program:

```
// sort_tst.cpp
// tests the template class SortedVector which is
// inherited from template class Vector.

#include "vec_sort.h" // template class SortedVector

void main ()
{
    SortedVector<int> sv (1, 0) ;
    cout << "sv: " << sv << endl ;

    sv.Add (5) ;
    cout << "sv: " << sv << endl ;
    sv.Add (10) ;
    cout << "sv: " << sv << endl ;
    sv.Add (2) ;
    cout << "sv: " << sv << endl ;
    sv.Add (-1) ;
    cout << "sv: " << sv << endl ;

    Vector<int> v (5, 0) ;
    v[0] = 3 ; v[1] = 4 ; v[2] = 2 ; v[3] = 0 ; v[4] = 1 ;
    cout << "v: " << v << endl ;

    SortedVector<int> sv1 (v) ;
    cout << "sv1: " << sv1 << endl ;

    cout << "sv1.Find (2): " << sv1.Find (2) << endl ;
}
```

with output:

```
sv: [0]
sv: [0, 5]
sv: [0, 5, 10]
sv: [0, 2, 5, 10]
sv: [-1, 0, 2, 5, 10]
v: [3, 4, 2, 0, 1]
sv1: [0, 1, 2, 3, 4]
sv1.Find (2): 2
```

---

<sup>6</sup> It can be shown that the worst-case running time of the binary search algorithm is of the order  $\log_2 n$  for  $n$  elements in the list to be searched.

The function body of `main()` first defines an object, `sv`, of **class** `SortedVector` which consists of one element of type `int`. Next, four additional `int` elements are added to `sv` using the `SortedVector<T>::Add()` member function. An object, `sv1`, of **class** `SortedVector` is then defined and initialised to the `Vector` object `v`. The output of `sv1` illustrates that directly after definition the elements of object `sv1` are sorted in increasing order. Finally, `main()` accesses the third element of `sv1` using the `SortedVector<T>::Find()` member function.

### 15.20.2 Memory, LocalMemory and GlobalMemory Classes for Windows

In Chapters 12 (WIN\_MEM.CPP) and 13 (WIN\_TMP.CPP) we examined a `GlobalMemory` **class** for allocating global memory when programming in a Windows environment. For allocating global memory we saw that the Windows Application Programming Interface (API) supports functions such as `GlobalSize()`, `GlobalAlloc()`, `GlobalReAlloc()`, `GlobalLock()`, `GlobalUnlock()` and `GlobalFree()`. The Windows API similarly supports functions for managing local memory blocks, such as `LocalSize()` and `LocalAlloc()`. Therefore it would be nice to declare a `LocalMemory` **class** with similar functionality to that of `GlobalMemory`, but for local memory.

Recognising the similarity between the Windows API functions for local and global memory and that objects of `LocalMemory` and `GlobalMemory` both require encapsulated memory handle and address data members then it seems natural to develop a base **class** `Memory` from which `LocalMemory` and `GlobalMemory` are derived:

```
// win_lgm.cpp
// illustrates local and global memory template classes
// for programming in a Windows environment
#include <iostream.h> // C++ I/O
#include <windows.h> // Windows header file

template <class T>
class Memory
{
protected:
    HANDLE hMemory ;           // handle to memory
    void FAR* mem_address ;    // address of memory
public:
    // constructor
    Memory () : hMemory (NULL), mem_address (NULL) {}
    // destructor
    ~Memory () {}
    // non-virtual member function
    void FAR* MemoryAddress () const { return mem_address ; }
    // virtual member functions
    virtual DWORD SizeOfBlock () = 0 ;
    virtual void FAR* Allocate (UINT size,
        UINT mem_flags=LMEM_MOVEABLE|LMEM_ZEROINIT) = 0 ;
    virtual void FAR* ReAllocate (UINT size,
        UINT mem_flags=LMEM_MOVEABLE|LMEM_ZEROINIT) = 0 ;
}; // template class Memory
```

```

template <class T>
class LocalMemory : public Memory<T>
{
public:
    // constructor
    LocalMemory ()
        : Memory<T> () {} ;
    // destructor
    ~LocalMemory () ;
    // overridden member functions
    DWORD     SizeOfBlock () ;
    void FAR* Allocate   (UINT size,
                          UINT mem_flags=LMEM_MOVEABLE|LMEM_ZEROINIT) ;
    void FAR* ReAllocate (UINT size,
                          UINT mem_flags=LMEM_MOVEABLE|LMEM_ZEROINIT) ;
}; // template class LocalMemory

template <class T>
class GlobalMemory : public Memory<T>
{
public:
    // constructor
    GlobalMemory ()
        : Memory<T> () {} ;
    // destructor
    ~GlobalMemory () ;
    // overridden member functions
    DWORD     SizeOfBlock () ;
    void FAR* Allocate   (UINT size,
                          UINT mem_flags=GMEM_MOVEABLE|GMEM_ZEROINIT) ;
    void FAR* ReAllocate (UINT size,
                          UINT mem_flags=GMEM_MOVEABLE|GMEM_ZEROINIT) ;
}; // template class GlobalMemory

// LocalMemory:

// destructor
template <class T>
LocalMemory<T>::~LocalMemory ()
{
    LocalUnlock (hMemory) ;    // unlock
    if (hMemory)           // free
        LocalFree (hMemory) ;
}

// return size of memory block
template <class T>
DWORD LocalMemory<T>::SizeOfBlock ()
{
    if (!hMemory)
        return 0 ;
}

```

```
    else
        return LocalSize (hMemory) ;
    }

// allocate memory
template <class T>
void FAR* LocalMemory<T>::Allocate (UINT size, UINT mem_flags)
{
    hMemory = LocalAlloc (mem_flags, size*sizeof(T)) ;

    // if unsuccessful allocation
    if (!hMemory)
    {
        MessageBox (GetActiveWindow (),
                    (LPCSTR) "insufficient memory",
                    (LPCSTR) "LocalMemory Warning",
                    MB_ICONEXCLAMATION | MB_OK) ;
        hMemory = NULL ;
        return NULL ;
    }
    mem_address = LocalLock (hMemory) ;
    return mem_address ;
}

// re-allocate memory
template <class T>
void FAR* LocalMemory<T>::ReAllocate (UINT size, UINT mem_flags)
{
    HANDLE h = LocalReAlloc (hMemory, size*sizeof(T), mem_flags) ;

    // if unsuccessful allocation
    if (!h)
    {
        MessageBox (GetActiveWindow (),
                    (LPCSTR) "insufficient memory",
                    (LPCSTR) "LocalMemory Warning",
                    MB_ICONEXCLAMATION | MB_OK) ;
        return NULL ;
    }
    else
    {
        hMemory = h ;
        mem_address = LocalLock (hMemory) ;
    }
    return mem_address ;
}

// GlobalMemory:
//...
void main ()
{
```

```

const int SIZE = 10 ;

// pointer to abstract base class Memory<T>
Memory<int>* mem ;

LocalMemory<int> l_mem ; // local memory object
GlobalMemory<int> g_mem ; // global memory object

//mem = &g_mem ;
mem = &l_mem ; // point to local memory object

// allocate memory
int* px = (int*) mem->Allocate (SIZE) ;

// set
for (int i=0; i<SIZE; i++)
    *(px+i)=i ;

// get
for (int j=0; j<SIZE; j++)
    cout << *(px+j) << " " ;
cout << endl ;
cout << "size: " << mem->SizeOfBlock () << endl ;

// re-allocate memory
px = (int*) mem->ReAllocate (2*SIZE) ;

// get
for (int k=0; k<2*SIZE; k++)
    cout << *(px+k) << " " ;
cout << endl ;
cout << "size: " << mem->SizeOfBlock () << endl ;
}

```

with output:

```

0 1 2 3 4 5 6 7 8 9
size: 32
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
size: 42

```

The memory handle and address data members, hMemory and mem\_address, have been transferred from the GlobalMemory classes presented in previous chapters to the Memory template class. The member function Memory<T> : : MemoryAddress() is declared non-virtual, since its operation is common to both derived classes. The member functions SizeOfBlock(), Allocate() and ReAllocate() of Memory<T> are, however, declared virtual because their use of the API memory functions is different for LocalMemory<T> and GlobalMemory<T>. Further, Memory<T> is an abstract base template class, since a memory object in Windows can be either local or global.

The function body of main() in the above program WIN\_LGM.CPP is similar to that presented WIN\_MEM.CPP and WIN\_TMP.CPP of Chapters 12 and 13, except that the member

functions now operate on a pointer to the abstract base `template class Memory` rather than operating directly on objects of classes `LocalMemory<T>` and `GlobalMemory<T>`.

### 15.20.3 The Matrix class Revisited

Chapter 13 presented a Matrix **template class** which was not 100% generic; refer to MTX\_TMP.H and MTMP\_TST.CPP of Chapter 13. The **class** declaration of Matrix<T> declared the function *GaussElimination()* and the overloaded **operator\*** () function, which both involved subtle floating-point type-dependent implementations. Such type dependencies can easily go unnoticed for the majority of applications of Matrix<T>. However, it was noted in Chapter 13 that this type dependency surfaced for composed types:

```
// mtmp_tst.cpp
//...
Point<double> p_init ;
Matrix < Point<double> > mpd (2, 2, p_init) ; // composed type
```

Let's now correct this type dependency problem for **template class** Matrix. The simplest solution is to observe that the *GaussElimination()* and **operator\*** () functions are floating-point routines and should consequently be removed from Matrix<T> and transferred to specialised non-**template** floating-point classes from **template class** Matrix:

```

// overloaded operator
DoubleMatrix operator * (const DoubleMatrix& m) ;
}; // class DoubleMatrix

class FloatMatrix : public Matrix<float>
{
public:
    // constructors
    FloatMatrix ()
        : Matrix<float> () {}
    FloatMatrix (int r, int c, float init_value)
        : Matrix<float> (r, c, init_value) {}
    // member function
    Vector<float> GaussElimination (Vector<float>& b,
                                      Boolean& solved) ;

    // overloaded operator
    FloatMatrix operator * (const FloatMatrix& m) ;
}; // class FloatMatrix
//...
// DoubleMatrix:

// Gauss elimination
Vector<double> DoubleMatrix::GaussElimination
    (Vector<double>& b, Boolean& solved)
{
    solved = FALSE ;

    int n = rows ; // assign number of rows to n

    Vector<double> x (n, 0.0) ; // solution Vector
    //...
}
//...
// FloatMatrix:

// Gauss elimination
Vector<float> FloatMatrix::GaussElimination (Vector<float>& b,
                                              Boolean& solved)
{
    solved = FALSE ;

    int n = rows ; // assign number of rows to n

    Vector<float> x (n, 0.0) ; // solution Vector
    //...
}
//...

```

Floating-point classes DoubleMatrix and FloatMatrix are now derived from Matrix<T> for **double** and **float** types respectively. This solution is not ideal, since the function definitions of *GaussElimination()* and *operator\**() have been repeated for

both **float** and **double** types, which conflicts greatly with the object-oriented programming philosophy.

The following program tests the DoubleMatrix, FloatMatrix and Matrix classes:

```
// mtx_tst.cpp
// tests the Point, Vector and Matrix template classes
// and the DoubleMatrix and FloatMatrix classes
#include <iostream.h> // C++ I/O

#include "pt_tmpl.h" // template class Point
#include "vec_tmpl.h" // template class Vector
#include "mtx_tmpl.h" // template class Matrix, ...

void main ()
{
    //FloatMatrix a (3, 3, 0.0) ;
    //Vector<float> b (3, 0.0), x (3, 0.0) ;
    DoubleMatrix a (3, 3, 0.0) ;
    Vector<double> b (3, 0.0), x (3, 0.0) ;

    a[0][0] = 1.0 ; a[0][1] = 2.0 ; a[0][2] = 3.0 ;
    a[1][0] = 2.0 ; a[1][1] = 3.0 ; a[1][2] = 4.0 ;
    a[2][0] = 3.0 ; a[2][1] = 4.0 ; a[2][2] = 1.0 ;
    b[0] = 14.0 ; b[1] = 20.0 ; b[2] = 14.0 ;

    cout << "DoubleMatrix a: " << endl << a ;
    cout << "Vector<double> b: " << endl << b << endl ;

    Boolean bool = FALSE ;

    x = a.GaussElimination (b, bool) ;

    if (bool)
        cout << "soln. Vector<double> x of system [a]{x}={b}: "
            << endl << x << endl ;
    else
        cout << "soln. by Gauss elimination failed" << endl ;
    cout << endl ;

    Point< Point<int> > ppi ; // composed type

    cout << "ppi: " << ppi << endl ;

    Vector<double> v_init ;
    Vector< Vector<double> > vvd (2, v_init); // composed type

    cout << "vvd: " << vvd << endl ;

    Point<double> p_init ;
```

---

```

Matrix < Point<double> > mpd (2, 2, p_init) ; // composed
// type

cout << "mpd: " << endl << mpd << endl ;
}

```

with output:

```

Matrix<double> a:
    1.00      2.00      3.00
    2.00      3.00      4.00
    3.00      4.00      1.00
Vector<double> b:
[14.00, 20.00, 14.00]
soln. Vector<double> x of system [a]{x}={b}:
[1.00, 2.00, 3.00]

ppi: ((0, 0, 0), (0, 0, 0), (0, 0, 0))
vvd: [[0.00, 0.00, 0.00], [0.00, 0.00, 0.00]]
mpd:
    (0.00, 0.00, 0.00)          (0.00, 0.00, 0.00)
    (0.00, 0.00, 0.00)          (0.00, 0.00, 0.00)

```

#### 15.20.4 Stack and Queue Classes

Let's now examine Stack and Queue **template** classes, which are both derived from the LinkedList **template class** presented in Chapter 13. The following Stack and Queue classes are based on the non-**template** Stack and Queue classes presented in Adams *et al.* (1995, pp. 972–90).

A *stack* is a special form of linked list in which an object is added to and removed from the same end of the list. The most recently added object to a stack list is the first object removed: the so-called *last in, first out* order. Since a stack is a *kind of* linked list it makes sense to derive Stack from LinkedList and add operations specific to **class** Stack.

First, let's remind ourselves of the **template class** declaration of LinkedList:

```

// 11.h
// template class LinkedList
//...
enum Boolean { FALSE, TRUE };

template <class T>
class LinkedList
{
protected:
    int num_nodes ;
    // nested Node class
    class Node
    {
public:
    T      data ;
    Node* next ;
}

```

```

// constructors
Node ()
    : data (), next (NULL) {}
Node (const T& d, Node* ptr=NULL)
    : data (d), next (ptr) {}
// destructor
~Node () { delete next ; }
} * first, * last ; // class Node
public:
enum { START, END };
// constructors
LinkedList ()
    : num_nodes (0), first (NULL), last (NULL) {}
LinkedList (const LinkedList& l) ;
// destructor
~LinkedList ()
{ delete first ; num_nodes = 0 ;
    first = last = NULL ; }
// member functions
int NumberNodes () const
{ return num_nodes ; }
Boolean Empty ()
{ return num_nodes==0 ? TRUE : FALSE ; }
int Search (const T& obj) ;
void Insert (const T& obj, int loc) ;
void Head (const T& obj) ;
void Append (const T& obj) ;
void Delete (int loc) ;
// overloaded operators
LinkedList& operator = (const LinkedList& l) ;
T& operator [] (int index) ;
Boolean operator == (const LinkedList& l) ;
Boolean operator != (const LinkedList& l) ;
// friends
friend ostream& operator << (ostream& s,
                                const LinkedList<T>& l) ;
}; // template class LinkedList
//...

```

The only difference between the above **class** declaration and the declaration of **LinkedList** presented in Chapter 13 is that the **private** data members have been changed to **protected**. The declaration of **Stack** is as follows:

```

// stack.h
// template class Stack

#ifndef _STACK_H // prevent multiple includes
#define _STACK_H

#include "ll.h" // template class LinkedList

```

```

template <class T>
class Stack : protected LinkedList<T>
{
public:
    // constructors
    Stack ()
        : LinkedList<T> () {}
    Stack (const Stack& s)
        : LinkedList<T> (s) {}

    // member functions
    Boolean Empty ()
        { return LinkedList<T>::Empty () ; }
    void Push (T obj)
        { Insert (obj, Stack<T>::START) ; }
    T Pop ()
        { assert(num_nodes>0) ; T obj=first->data ;
            Delete(Stack<T>::START) ; return obj ; }
}; // template class Stack

#endif // _STACK_H

```

Generally, there are two key operations performed on a stack: insert or *push* an object to the start of the stack, and delete or *pop* an object from the start of the stack. `Stack::Push()` and `Stack::Pop()` simply call `LinkedList::Insert()` and `LinkedList::Delete()` respectively to perform the push and pop operations.

The access specifier in the `class` declaration of `Stack` is `protected` rather than `public`, indicating that the `public` member functions of `LinkedList` are not `public` to `Stack`. This is because the majority of member functions of `LinkedList` are undefined for `Stack`. The exception is the member function `Empty()`, which is also required by `Stack` and has therefore been specifically overridden for `Stack`.

Another special form of linked list is a *queue* in which objects are added to the start of a list and removed from the end of a list. Contrary to a stack, the most recently added object to a queue list is the last object removed: the so-called *first in, first out* order. Since a queue is a kind of linked list, `class Queue` is derived from `LinkedList` with operations specific to `Queue` added. The declaration of `Queue` is:

```

// queue.h
// template class Queue

#ifndef _QUEUE_H // prevent multiple includes
#define _QUEUE_H

#include "ll.h" // template class LinkedList

template <class T>
class Queue : protected LinkedList<T>
{
public:
    // constructors
    Queue ()
        : LinkedList<T> () {}

```

```

Queue (const Queue& q)
    : LinkedList<T> (q) {}
// member functions
Boolean Empty ()
{ return LinkedList<T>::Empty () ; }
void Add (T obj)
{ Insert (obj, Queue<T>::END) ; }
T Remove ()
{ assert(num_nodes>0) ; T obj=first->data ;
  Delete(Queue<T>::START) ; return obj ; }
}; // template class Queue

#endif // _QUEUE_H

```

Instead of member functions *Push()* and *Pop()* for **class** Stack we define the functions *Add()* and *Remove()*, respectively, for **class** Queue. The member function *Add()* inserts an object to the end of a queue and *Remove()* deletes an object from the start of a queue. Like **class** Stack, the access specifier in the **class** declaration of Queue is **protected** so that the **public** member functions of LinkedList are not **public** to Queue.

A program which tests the above Stack and Queue classes is given below:

```

// s_q_tst.cpp
// tests the LinkedList template class

#include "stack.h" // template class Stack
#include "queue.h" // template class Queue

void main ()
{
// Stack:
Stack<int> s ;

s.Push (1) ; s.Push (2) ; s.Push (3) ;

while (!s.Empty ())
  cout << s.Pop () ;

cout << endl ;

// Queue:
Queue<int> q ;

q.Add (1) ; q.Add (2) ; q.Add (3) ;

while (!q.Empty ())
  cout << q.Remove () ;
}

```

with output:

---

123

### 15.20.5 VectorIterator and LinkedListIterator Classes

In Chapter 13 (V\_IT\_C.H and LLI.H) we examined iterator classes for classes `Vector` and `LinkedList`, namely `VectorIterator` and `LinkedListIterator`. It is noted that the four member functions `More()`, `Current()`, `Next()` and `ReStart()` are common to both `VectorIterator` and `LinkedListIterator`. Thus, let us now declare an abstract **class**, `Iterator`, from which `VectorIterator` and `LinkedListIterator` will be derived:

```
// iter.h
// template class Iterator

#ifndef _ITER_H // prevent multiple includes
#define _ITER_H

#include "bool.h" // enum Boolean

// abstract class Iterator
template <class T>
class Iterator
{
public:
    // virtual member functions
    virtual Boolean More () const = 0 ;
    virtual T& Current () const = 0 ;
    virtual void Next () = 0 ;
    virtual void ReStart () = 0 ;
    // virtual overloaded operators
    virtual Boolean operator ++ () = 0 ; // prefix
    virtual Boolean operator ++ (int) = 0 ; // postfix
};

#endif // _ITER_H
```

The **virtual** member functions `More()`, `Current()`, `Next()` and `ReStart()` determine whether or not more elements or nodes are within an object, return the current element or node, advance to the next element or node and reposition the current position to the start of a vector or list respectively. Virtual overloaded prefix and postfix increment operators are also declared for iterator object manipulation similar to that of C++ integral types. Note that `Iterator` has no data members.

It is now a simple matter of deriving `VectorIterator` from `Iterator`:

```
template <class T>
class Vector
{
//...
    friend class VectorIterator<T> ;
};

//...
```

```

// vi.h
// template class VectorIterator

#ifndef _VI_H // prevent multiple includes
#define _VI_H

#include "iter.h" // abstract class Iterator
#include "v.h" // template class Vector

// template class VectorIterator
template <class T>
class VectorIterator : public Iterator<T>
{
private:
    int c_elem; // current element
    const Vector<T>& vec; // current Vector
public:
    // constructor
    VectorIterator (const Vector<T>& v)
        : c_elem (0), vec (v) {}
    // member functions
    Boolean More () const
    { return c_elem < vec.n_elements ? TRUE : FALSE; }
    T& Current () const { return vec.array[c_elem]; }
    void Next () { c_elem++; }
    void ReStart () { c_elem = 0; }
    // overloaded operators
    Boolean operator ++ () // prefix
    { c_elem++; return More(); }
    Boolean operator ++ (int) // postfix
    { c_elem++; return More(); }
}; // template class VectorIterator

#endif // _VI_H

```

An object of `VectorIterator` holds a reference to the `Vector` object being manipulated and the current element index. Similarly, `LinkedListIterator` is given by:

```

template <class T>
class LinkedList
{
//...
friend class LinkedListIterator<T> ;
};

//...
// lli.h
// template class LinkedListIterator

#ifndef _LLI_ITER_H // prevent multiple includes
#define _LLI_ITER_H

```

```

#include "iter.h" // abstract class Iterator
#include "ll.h" // template class LinkedList
#include "bool.h" // enum Boolean

template <class T>
class LinkedListIterator : public Iterator<T>
{
private:
    LinkedList<T>::Node* c_node ; // current node
    const LinkedList<T>& c_list ; // current list
public:
    // constructor
    LinkedListIterator (const LinkedList<T>& l)
        : c_node (l.first), c_list (l) {}
    // member functions
    Boolean More () const
    { return c_node != NULL ? TRUE : FALSE ; }
    T& Current () const { return c_node->data ; }
    void Next () { c_node = c_node->next ; }
    void ReStart () { c_node = c_list.first ; }
    // overloaded operators
    Boolean operator ++ () // prefix
    { c_node=c_node->next ; return More() ; }
    Boolean operator ++ (int) // postfix
    { c_node=c_node->next ; return More() ; }
}; // template class LinkedListIterator

#endif // _LLI_ITER_H

```

The following program illustrates the above `VectorIterator` and `LinkedListIterator` classes:

```

// iter_tst.cpp
// tests the iterator classes VectorIterator and
// LinkedListIterator
//...
void main ()
{
    // LinkedList & LinkedListIterator
    LinkedList<int> ll ;
    ll.Append (2) ;
    ll.Append (4) ;
    ll.Append (6) ;

    cout << "LinkedListIterator lli (ll): " ;
    for (LinkedListIterator<int>lli(ll); lli.More(); ++lli)
        cout << lli.Current () << " " ;
    cout << endl ;

    // Vector & VectorIterator
    Vector<int> v (3, 0) ;

```

```

v[0] = 2 ;
v[1] = 4 ;
v[2] = 6 ;

cout << "VectorIterator vi (v)      : " ;
for (VectorIterator<int>vi(v); vi.More(); vi++)
    cout << vi.Current () << " " ;
cout << endl ;
}

```

with output:

```

LinkedListIterator lli(l1): 2 4 6
VectorIterator vi(v)      : 2 4 6

```

## 15.21 Exception Handling and Inheritance

This section examines exception handling and inheritance. To remind ourselves of the **try-catch-throw** exception handling mechanism of C++, consider the following program, which simply throws and catches an exception of a derived **class**:

```

// exp_inh.cpp
// illustrates exception handling and inheritance
#include <iostream.h> // C++ I/O

class Base
{
};

class Derived : public Base
{
};

void main ()
{
    try
    {
        Derived derived;

        // throw exception to handler
        throw derived;
    }
    // process exception
    catch (Derived)
    {
        cout << "Derived exception caught" << endl;
    }
    //catch (const Derived) // duplicate handlers
    //catch (Derived&)
}

```

---

```
// catch (const Derived&)
}
```

with output:

```
Derived exception caught
```

Remember from Chapter 14 that a **throw** expression of type T is a match for an exception handler of type T, **const** T, T& or **const** T&.

Let us now examine more closely the interaction of base and derived **class** exception handlers:

```
// bd_x_inh.cpp
// illustrates base and derived classes,
// exception handling and inheritance
#include <iostream.h> // C++ I/O

class Base
{
};

class PubDerived : public Base
{
};

void main ()
{
try
{
    PubDerived pub_derived ;

    // throw exception to handler
    throw pub_derived ;
}
// process exception
catch (Base)
{
    cout << "Base exception caught" << endl ;
}
catch (PubDerived)
{
    cout << "PubDerived exception caught" << endl ;
}
}
```

with output:

```
Base exception caught
```

The output of the above program illustrates that a **throw** expression of a derived **class** is a match for an exception handler of a base **class**, provided the base **class** is accessible to a

derived **class**. Compilers will issue a compilation warning message of the form ‘Pub-Derived handler is hidden by Base handler’ regarding the hiding of the PubDerived handler by the Base handler.

Consider now the above program, but declaring a **privately** derived **class**:

```
// b_pri_d.cpp
// illustrates base and privately derived classes,
// exception handling and inheritance
#include <iostream.h> // C++ I/O

class Base
{
};

class PriDerived : private Base
{
};

void main ()
{
try
{
    PriDerived pri_derived ;

    // throw exception to handler
    throw pri_derived ;
}
// process exception
catch (Base)
{
    cout << "Base exception caught" << endl ;
}
catch (PriDerived)
{
    cout << "PriDerived exception caught" << endl ;
}
}
```

with output:

```
PriDerived exception caught
```

which illustrates that a base **class** handler is not a match for a **privately** derived **class** exception, and similarly for a **protectedly** derived **class** exception.

Generally, a derived **class** exception is caught by a base **class** handler rather than by a handler of the derived **class**. This can result in the derived **class** exception thrown being *sliced* if the exception object is thrown by value. Alternatively, if a thrown exception object is caught by reference the thrown exception is not sliced:

```
// bd_catch.cpp
// illustrates catching a derived class exception
```

---

```
#include <iostream.h> // C++ I/O

class Base
{
public:
    virtual void Function ()
    { cout << "Base::Function() called" << endl ; }

class Derived : public Base
{
public:
    void Function ()
    { cout << "Derived::Function() called" << endl ; }

void main ()
{
try
{
    Derived derived ;

    // throw exception to handler
    throw derived ;
}
// process exception
catch (Base& b)
{
    b.Function () ;
}
}
```

with output:

```
Derived::Function() called
```

The above program illustrates that even though a derived **class** exception object is thrown and the exception handler catches objects of the base **class** the `Derived::Function()` is called because the thrown exception object is caught by reference and not by value.

### 15.21.1 Exception Specification

Chapter 14 illustrated that the *exception specification* of a function declaration enables a function to specify exactly what exceptions it is allowed to **throw**. With respect to inheritance, a function that is able to **throw** an exception of a given **class** can also **throw** an exception of any **class** **publicly** derived from that **class**:

```
// x_sp_inh.cpp
// illustrates function exception specifications and
// inheritance
#include <iostream.h> // C++ I/O
```

```
class Base
{
};

class Derived : public Base
{
};

// throw Base exception specification
void FThrowBase (int i) throw (Base)
{
    Base b;
    Derived d;

    if (i == 0)
        throw b; // throw Base
    if (i == 1)
        throw d; // throw Derived
}

void main ()
{
    try
    {
        //FThrowBase (0); // o.k.
        FThrowBase (1); // o.k.
    }
    // process exception
    catch (Base)
    {
        cout << "exception caught" << endl;
    }
}
```

with output:

```
exception caught
```

### 15.21.2 Re-Throwing an Exception

It was noted above that if a derived **class** exception is caught by a base **class** handler that this may result in the thrown object being sliced. However, if such an object is then re-thrown it is the original object that is re-thrown, not the sliced object.

## 15.22 Summary

Inheritance allows a *derived class* to inherit the features of one or more other *base classes*. Inheritance relationships are specified during the declaration of a derived **class**. The three

access specifiers **private**, **protected** and **public** give a programmer complete flexibility in controlling data member and member function accessibility during inheritance.

Generally, two alternative approaches can be adopted when developing a new **class**: containment or inheritance. Containment exploits the *has a* relationship, in which case a given **class** has objects of another **class**. On the other hand, inheritance exploits the *is a* relationship, in which case one **class** is a kind of another **class**.

Combining the features of inheritance and **virtual** functions enables a set of base and derived **class** member functions to exhibit run-time polymorphic behaviour.

## Exercises

15.1 A given **class** is declared as:

```
class Base
{
protected:
    int bdm ;
public:
    // constructors
    Base ()
        : bdm (0) {}
    Base (int i)
        : bdm (i) {}
    // copy constructor
    Base (const Base& b)
        : bdm (b.bdm) {}
    // overloaded operator
    Base& operator = (const Base& b)
    {
        bdm = b.bdm ;
        return *this ;
    }
};
```

Publicly derive a **class**, **Derived**, from **Base** which adds its own **protected** data member, **ddm**, and defines both default and one-argument constructors, a copy constructor and an overloaded **=** operator member function.

15.2 What is wrong with the following assignment expression in **main()**?

```
class Base
{
};

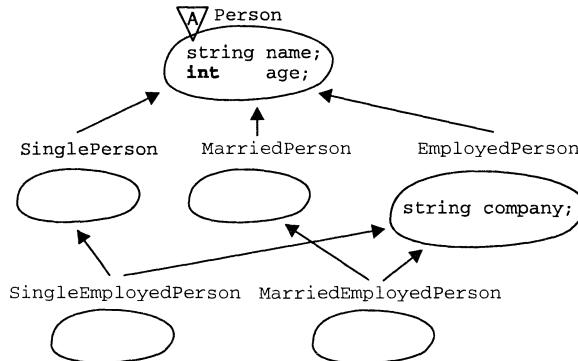
class Derived : public Base
{
};

void main ()
{
    Base b ;
```

```
Derived d ;
//...
d = b ;
}
```

15.3 Declare an abstract **class** called Person which encapsulates name and age data members.

15.4 Declare classes for the following **class** hierarchy:



15.5 The following program will not compile. Why?

```
template <class T>
class Point
{
};

void main ()
{
    Point<int>    p (1, 2) ;
    Point<double> q (3.0, 4.0) ;
    //...
    q = p ;
}
```

15.6 In Exercise 9.7 a CartCoorSys **class** was developed to represent a Cartesian coordinate system. Consider the development of classes for cylindrical, polar and spherical coordinate systems. Should containment or inheritance be used?

15.7 In the following **catch** statement, which *Message ()* member function is called?

```
class Exception
{
};

public:
    virtual string& Message () ;
    //...
};

class InsufficientMemory : public Exception
```

```
{  
public:  
    string& Message () ;  
    //...  
};  
  
void main ()  
{  
try  
{  
    InsufficientMemory memory_exp ;  
    //...  
    throw memory_exp ;  
}  
catch (Exception& exp)  
{  
    exp.Message () ;  
}  
}
```

# Run-Time Type Information and Casting

We saw in the previous chapter that similar classes can be placed in a class hierarchy, with numerous classes derived from a single base class. Working in terms of a pointer to the base class, rather than objects of the derived classes, we are able to operate on functions common to all derived classes via a single pointer object that can point to any derived class object. Such a run-time capability is essential to object-oriented programming, but unfortunately moves us away from the strongly typed compile-time mechanism so characteristic of the C++ language.

Run-time support is given through the use of virtual member functions, and C++ complements this with language-based run-time type information (RTTI) features. The RTTI approach, as the name suggests, is concerned with determining the type of an object at run-time and can be summarised in terms of two key operators: `dynamic_cast<>()` and `typeid()`. The `dynamic_cast<>()` operator allows a pointer or reference of a base class to be cast to a derived class provided the base class is polymorphic. Using the `dynamic_cast<>()` operator to perform such down-casting is, importantly, checked at run-time, whereas the non-RTTI casting approach goes unchecked at run-time. The `typeid()` operator allows the exact type of an object to be determined for both polymorphic and non-polymorphic classes.

Casting is the process of converting an object's type. This chapter will examine three cast operators (`static_cast<>()`, `reinterpret_cast<>()` and `const_cast<>()`) which replace the (T)e C-style and T(e) C++-style casts that we have seen used in previous chapters. Three different cast operators are defined because the single (T)e cast style is, in certain circumstances, not sufficiently explicit in disclosing the exact intentions behind a cast operation.



## 16.1 Recap

When we discussed **virtual** functions in the previous chapter, an abstract base **class** Shape, a base **class** Plane and associated derived classes were developed:

```
class Shape // abstract base class of all shapes
{
//...
```

```

virtual Point Centroid () = 0 ; // pure virtual function
//...
};

//...
class Plane // base class
{
};

//...
class Polygon : public Shape, public Plane
{
//...
Point Centroid () ; // overridden member function
};

//...
class Triangle : public Polygon
{
//...
};

//...
class Rectangle : public Polygon
{
//...
};

//...
class Circle : public Shape, public Plane
{
Point Centroid () ; // overridden member function
};

//...

```

A simple application of the above classes was demonstrated in SHAPES.CPP, which is of the form:

```

void main ()
{
    Shape* s (NULL) ; // pointer to abstract base class Shape
    Circle c ; Rectangle r ; Triangle t ; // derived class
                                            // objects
//...
// if Circle:
s = &c ; // point to Circle
//...
// if Rectangle:
s = &r ; // point to Rectangle
//...
// if Triangle:
s = &t ; // point to Triangle
//...
}

```

The problem with working with derived **class** objects via a pointer to a base **class**, which can point to any number of derived **class** objects, is that we have no way of determining from

the pointer alone which type of object the pointer is in fact pointing to. Run-time polymorphism, via the use of **virtual** functions, focuses our attention on the common characteristics of different objects and allows us to use these common characteristics in designing a general interface with complete disregard for the type information of each of the objects. There are occasions when this is exactly what we require. However, there are other occasions when we require the run-time polymorphic behaviour of a set of objects while at the same time maintaining the type information of each of the objects.

## 16.2 Use of **virtual** Functions

### 16.2.1 A Direct Method

One solution to the problem of recovering type information about an object from a pointer is simply to define a member function for each **class** which returns a **string** object indicating an object's **class** and then call this function when required:

```
class Shape // abstract base class of all shapes
{
//...
string WhoAmI () const { return string ("Shape") ; }
//...
};

//...
class Plane // base class
{
//...
string WhoAmI () const { return string ("Plane") ; }
};
//...
class Polygon : public Shape, public Plane
{
//...
string WhoAmI () const { return string ("Polygon") ; }
//...
};
//...
```

Alternatively, we can define the *WhoAmI()* member functions for the derived classes so that they return a specific code which indicates a **class**'s type and which is defined in the **Shape** abstract base **class**:

```
// who.cpp
// illustrates detecting the type of an object
// through the use of an abstract base class type code
// and type casting an abstract base class pointer
#include <iostream.h> // C++ I/O

// abstract base class of all shapes
```

```
class Shape
{
public:
    enum Who {POLYGON, TRIANGLE, RECTANGLE};
    virtual Who WhoAmI () const = 0 ;
    virtual void Display ()          = 0 ;
};

class Plane
{
};

class Polygon : public Shape, public Plane
{
public:
    Who WhoAmI () const { return POLYGON ; }
    void Display () { cout << "Polygon" << endl ; }
};

class Triangle : public Polygon
{
public:
    Who WhoAmI () const { return TRIANGLE ; }
    void Display () { cout << "Triangle" << endl ; }
};

class Rectangle : public Polygon
{
public:
    Who WhoAmI () const { return RECTANGLE ; }
    void Display () { cout << "Rectangle" << endl ; }
};

void main ()
{
    void Function (Shape* s) ; // prototype

    Shape* s (NULL) ;
    Triangle t ;

    s = &t ; // Shape pointer points to a Triangle object

    Function (s) ;
}

void Function (Shape* s)
{
    Shape::Who who_are_you = s->WhoAmI () ;

    if (who_are_you == Shape::POLYGON)
    {
```

```

    Polygon* p = (Polygon*)s ;
    p->Display () ;
}
else if (who_are_you == Shape::TRIANGLE)
{
    Triangle* t = (Triangle*)s ;
    t->Display () ;
}
else if (who_are_you == Shape::RECTANGLE)
{
    Rectangle* r = (Rectangle*)s ;
    r->Display () ;
}
}

```

with output:

Triangle

The type code for each derived **class** is defined using an **enum** Who, which is **publicly** declared in Shape. The **virtual** function *WhoAmI()* is defined for each derived **class** and simply returns an integer corresponding to the **class** of an object which the function operates on.

*Function()* is passed a pointer, s, to abstract base **class** Shape which could be pointing to any type. The exact type that s is pointing to is determined from the *WhoAmI()* function, and by using a block of **if-else- if** statements the appropriate *Display()* function is called. On considering the first **if** statement, we observe that the pointer, s, to Shape is *down-cast*<sup>1</sup> to **class** Polygon:

```

//...
Polygon* p = (Polygon*)s ;

```

It was discussed in Chapter 15 that it is natural to cast from a derived **class** object to a base **class** object, or *slice* a derived **class** object. However, the reverse process actually adds information to a base **class** object. The compiler knows nothing of such type casts and can therefore offer no support, warning or error messages to a programmer.

### 16.2.2 The Double-Dispatch Method

An alternative solution to the problem of recovering the type information of an object at run-time is a technique called *double-dispatch*. The technique used above for determining run-time type information involved defining an overridden *WhoAmI()* **virtual** function which returns a unique type code for each derived **class**, explicitly testing the type code returned by *WhoAmI()* and then performing the appropriate operation. The double-dispatch approach is still based on the use of **virtual** functions, but now involves a *Dispatch()* function which is declared a pure **virtual** function in the abstract base **class** Shape and overridden for each derived **class**. The single argument to *Dispatch()* is a reference to a

---

<sup>1</sup> Type casting a base **class** object to a derived **class** object is referred to as *down-casting* because we are going down the **class** inheritance diagram rather than up; see Fig. 15.1.

**class** called Accept. The *Dispatch()* function for each derived **class** simply calls the appropriate accept function, which is to be defined in a **class** derived from **class** Accept. The abstract **class** Accept declares a pure **virtual** function for each **class** derived from Shape, such as *AcceptPolygon(const Polygon& p)*, and has a single function argument which is a reference to the **class** or type of object for which the function is to provide an operation:

```
// dbl_dis.cpp
// illustrates double dispatch
#include <iostream.h> // C++ I/O

class Accept ;

class Polygon ; class Triangle ; class Rectangle ;

// abstract base class of all shapes
class Shape
{
public:
    virtual void Dispatch (Accept& a) const = 0 ;
};

class Accept
{
public:
    virtual void AcceptPolygon (const Polygon& p) = 0 ;
    virtual void AcceptTriangle (const Triangle& t) = 0 ;
    virtual void AcceptRectangle (const Rectangle& r) = 0 ;
};

class Plane
{
};

class Polygon : public Shape, public Plane
{
public:
    void Dispatch (Accept& a) const
        { a.AcceptPolygon (*this) ; }
};

class Triangle : public Polygon
{
public:
    void Dispatch (Accept& a) const
        { a.AcceptTriangle (*this) ; }
};

class Rectangle : public Polygon
{
public:
```

```

void Dispatch (Accept& a) const
{ a.AcceptRectangle (*this) ; }

class DisplayShape : public Accept
{
public:
    void AcceptPolygon (const Polygon& p)
    { cout << "Polygon" << endl ; }
    void AcceptTriangle (const Triangle& t)
    { cout << "Triangle" << endl ; }
    void AcceptRectangle (const Rectangle& r)
    { cout << "Rectangle" << endl ; }
};

void main ()
{
    Shape* s (NULL) ;
    Triangle t ;

    s = &t ; // Shape pointer points to a Triangle object

    DisplayShape dis_shape ;

    s->Dispatch (dis_shape) ;
}

```

with output:

Triangle

Since we are currently interested in simply displaying the **class** name of an object, we derive a **class** called **DisplayShape** from **Accept** and override the required accept function(s) in **DisplayShape**. A **DisplayShape** object is then passed to *Dispatch()*, indicating that we want to display an object pointed to by the **Shape** abstract base **class** pointer **s**.

The double-dispatch approach completely bypasses the need to determine the type of a given object at run-time in order to call the appropriate function – it figures out the details for you. The price to pay is that there are now two calls to **virtual** functions instead of the one **virtual** function call used in the more direct method presented in WHO.CPP. The first **virtual** function call is to the *Dispatch()* function, while the second call is to the appropriate accept function. Once the *Dispatch()* function is overridden for each derived **class** we can add new functions to the existing classes via the **Accept** **class** and classes derived from **Accept** without modification to the classes derived from **Shape**.

## 16.3 The RTTI Approach

Before C++ offered direct support for run-time type information, a programmer had to rely solely on the use of **virtual** functions to resolve the issue of an object's type at run-time. The C++ RTTI approach focuses around two operators called **dynamic\_cast<>()** and

`typeid()` and several support classes. Let's first examine the `dynamic_cast<>()` operator, which is probably the most important feature of RTTI.

### 16.3.1 The `dynamic_cast<>()` Operator

The `dynamic_cast<>()` operator can be used to cast to either a pointer or a reference type. Let's first examine the cast to a pointer version:

```
dynamic_cast<T*>(p)
```

This will convert and return the operand `p` to type `T*` if, and only if, `*p` is of type `T` or of a **class** or type derived from `T`; else the value of the returned pointer is 0. Note that `p` must be a pointer or an expression which resolves to a pointer. To help illustrate `dynamic_cast<T*>()` let's examine a program:

```
// dyn_cstp.cpp
// illustrates the dynamic_cast<T*>() operator
#include <iostream.h> // C++ I/O

// non-polymorphic class
class X
{
};

// polymorphic class
class Base
{
public:
    virtual void Function () {} ;
};

// derived from Base
class Derived : public Base
{
};

void main ()
{
    Derived d ;
    Base*    bp = &d ;
    X*       xp (NULL) ;

    // o.k.: downcast from Base to Derived-unchecked by compiler
    Derived* dp0 = (Derived*)bp ;

    // o.k.: downcast from Base to Derived-checked by compiler
    // at run-time
    Derived* dp1 = dynamic_cast<Derived*>(bp) ;

    // error: X is a non-polymorphic class
    Derived* dp2 = dynamic_cast<Derived*>(xp) ;
```

```
// o.k.: operand expression
Derived* dp3 = dynamic_cast<Derived*>(bp++) ;

// o.k.: cast from Base to void
void* vp = dynamic_cast<void*>(bp) ;
}
```

The following static down-cast from Base to Derived:

```
Derived* dp0 = (Derived*)bp ;
```

is legal but unchecked at compilation. However, the same cast operation using the `dynamic_cast<T*>()` operator:

```
Derived* dp1 = dynamic_cast<Derived*>(bp) ;
```

is checked at run-time.

Type T in the `dynamic_cast<T*>(p)` operator can be of type **void** provided operand p is a pointer:

```
void* vp = dynamic_cast<void*>(bp) ;
```

The above program illustrates that a cast from a base **class** to a derived **class** can only be performed if the base **class** is a polymorphic **class**, i.e. a **class** with one or more **virtual** functions. RTTI only supports polymorphic types for which objects can be manipulated via a base **class**. Thus, there is no RTTI support for integral types such as **int** and **double**.

Let's now examine the use of the `dynamic_cast<>()` operator for casting to a reference type:

```
dynamic_cast<T&>(r)
```

The reference version of `dynamic_cast<>()` is similar to the pointer version except that if a cast to a reference type is unsuccessful a `Bad_cast` exception is thrown. The exception-handling **class** `Bad_cast` is declared in the `TYPEINFO.H` header file, which is discussed later. For example:

```
// dyn_cstr.cpp
// illustrates the dynamic_cast<T&>()
// operator
#include <iostream.h> // C++ I/O
#include <typeinfo.h> // RTTI

// non-polymorphic class
class X
{
};

// polymorphic class
class Base
{
```

```

public:
    virtual void Function () {} ;

}

// derived from Base
class Derived : public Base
{
};

void main ()
{
try
{
    Derived d ;
    Base& br = d ;

    Derived& dr = dynamic_cast<Derived&>(br) ;
}
catch (Bad_cast)
{
    cout << "dynamic_cast<T&>() failed" << endl ;
}
}
}

```

Using the conventional C-style cast syntax it is illegal to cast from a **virtual** base **class** to a derived **class**. The **dynamic\_cast<T\*>()** operator, however, allows such a cast to be performed:

```

// v_b_c.cpp
// illustrates the use of the dynamic_cast<T*>() operator
// with virtual base classes
#include <iostream.h> // C++ I/O

// polymorphic class
class Base
{
public:
    virtual void Function () {} ;
};

// derived from virtual base class, Base
class Derived : virtual public Base
{
};

void main ()
{
    Derived d ;
    Base* bp = &d ;

    // error: can't cast from virtual base class to derived class
}

```

---

```
Derived* dp0 = (Derived*)bp ;

// o.k.: checked at run-time
Derived* dp1 = dynamic_cast<Derived*>(bp) ;
}
```

### 16.3.2 The typeid() Operator

The `dynamic_cast<>()` operator informs us of whether or not an object pointed to is of a given **class** or any **class** derived from that **class**, but does not allow us to determine the exact type of an object. The `typeid()`<sup>2</sup> operator, however, does allow the exact type of an object to be determined, and is of the following general syntax:

```
typeid (type_name)
typeid (expression)
```

The `typeid()` operator returns a **const** reference to an object of **class** `type_info`; see below. The type of the object returned by `typeid()` represents the type of the `type_name` or `expression` operand. Unlike `dynamic_cast<>()`, the `typeid()` operator can have an integral non-polymorphic data type operand. If the operand is non-polymorphic `typeid()` returns an object of the static type, whereas if the operand is polymorphic `typeid()` returns the dynamic type of the object pointed to. If the operand to `typeid()` is a dereferenced NULL pointer a `Bad_typeid` exception is thrown. The following program illustrates `typeid()`:

```
// typeid.cpp
// illustrates the use of the typeid() operator
#include <iostream.h> // C++ I/O
#include <typeinfo.h> // RTTI

// polymorphic class
class Base
{
public:
    virtual void Function () {} ;
};

// derived from Base
class Derived : public Base
{
};

void main ()
{
    Derived d ;
    Base* bp = &d ;

    if (typeid (*bp) == typeid (Derived))
```

---

<sup>2</sup> Type identity

```

        cout << "Base pointer pointing to Derived object"
        << endl ;
}

```

with output:

```
Base pointer pointing to Derived object
```

Note the inclusion of the TYPEINFO.H header file, which is necessary if the object returned by typeid() is to be used.

### 16.3.3 The TYPEINFO.H Header File and the type\_info, Bad\_cast and Bad\_typeid Classes

The TYPEINFO.H header file includes **class** declarations for the run-time type information classes type\_info, Bad\_cast and Bad\_typeid. A typical extract from the TYPEINFO.H header file is<sup>3</sup>:

```

class type_info
{
private:
    // private copy constructor & assignment operator
    type_info (const type_info& rhs) ;
    type_info& operator = (const type_info& rhs) ;
public:
    // virtual destructor
    virtual ~type_info () ;
    // member functions
    bool before (const type_info& rhs) const ;
    const char* name () const ;
    // overloaded operators
    bool operator == (const type_info& rhs) const ;
    bool operator != (const type_info& rhs) const ;
};

//...
class Bad_cast
{};

class Bad_typeid
{};
//...

```

The copy constructor and overloaded assignment operator of **class** type\_info are declared **private** so that type\_info objects cannot be defined and copied. type\_info supports the two comparison operators == and !=. The before() member function compares the lexical order of different types and returns either logically true or false, e.g.:

```
bool i = typeid (int).before (typeid (double)) ; // false
```

---

<sup>3</sup> The exact definition of the type\_info polymorphic **class** is implementation dependent.

The `name()` member function returns the type name of the operand to the `typeid()` operator, e.g.:

```
cout << typeid (*p).name () ;
```

If a cast to a reference type is performed using the `dynamic_cast<T&>()` operator and is unsuccessful then a `Bad_cast` exception is thrown. The empty `Bad_cast` exception-handling **class** is declared in the header file `TYPEINFO.H`.

If the `typeid()` operator is unsuccessful in identifying the type of its operand it throws a `Bad_typeid` exception. As with `Bad_cast`, the empty `Bad_typeid` exception-handling **class** is declared in the header file `TYPEINFO.H`.

The following program helps illustrate the `type_info` **class**:

```
// typeinfo.cpp
// illustrates the type_info class
#include <iostream.h> // C++ I/O
#include <typeinfo.h> // RTTI

// polymorphic class
class Base
{
public:
    virtual void Function () {} ;
};

// derived from Base
class Derived : public Base
{
};

void main ()
{
try
{
    Derived d ;
    Base* bp = &d ;

    cout << "typeid(Derived).name(): "
        << typeid (Derived).name () << endl ;
    cout << "typeid(int).name(): "
        << typeid (int).name () << endl ;

    typeid (Base).before (typeid (Derived))
    ? cout << "Base before Derived" << endl
    : cout << "Base after Derived" << endl ;

    typeid (int).before (typeid (double))
    ? cout << "int before double" << endl
    : cout << "int after double" << endl ;

    // throw Bad_typeid exception
```

```
    bp = NULL ;
    cout << typeid (*bp).name () << endl ;
}
catch (Bad_typeid)
{
    cout << "Bad_typeid exception caught" << endl ;
}
}
```

with output:

```
typeid(Derived).name(): Derived
typeid(int).name() : int
Base before Derived
int after double
Bad_typeid exception caught
```

## 16.4 Casting

The C-style syntax of casting that we have seen to date is of the form `(T) e` and casts an expression `e` to type `T`. The C++-style of casting which we have seen in previous chapters uses the constructor notation `T(e)` as an alternative to `(T) e`. However, the `dynamic_cast<>()` operator introduces a new style of performing conversions from one type to another. In addition to the `dynamic_cast<>()` operator previously discussed, C++ also supports three additional cast operators, `static_cast<>()`, `reinterpret_cast<>()` and `const_cast<>()`, each having its own specific application. These three cast operators help separate and clarify the different forms of conversion performed by the traditional `(T) e` cast notation. This is necessary because the simple `(T) e` cast notation can have several different interpretations, making it difficult for a programmer to establish from a cast expression alone what the original programmer had in mind.

### 16.4.1 The `static_cast<>()` Operator

The syntax of the `static_cast<>()` operator is:

```
static_cast<T>(e)
```

where type `T` can be either an integral or user-defined type, pointer, reference or `enum`. No run-time check is performed when using the `static_cast<>()` operator; only static type checking.

Integral types, such as `int` and `double`, can be cast to `enum` types using `static_cast<>()`. Both pointer and reference types must be complete at the time of compilation for `static_cast<>()` to perform the conversion from one type to another. To cast a pointer of a `class X` to a pointer of a `class Y`, `X` must be a non-`virtual` base `class` of `Y` and an unambiguous conversion must exist. A given object can be cast to an object of another `class` provided that the appropriate constructor or conversion function exists.

One of the serious faults of the C-style cast notation,  $(T)e$ , is that it allows a programmer to cast-away the **const** attribute attached to an object. The `static_cast<>()` operator rectifies this fault by respecting **constness**.

The following program illustrates some of the above-mentioned properties of `static_cast<>()`:

```
// stat_cst.cpp
// illustrates the static_cast<>() operator

enum Boolean { FALSE, TRUE };

class Base
{
};

// derived from Base
class Derived : public Base
{
public:
    // conversion function - Derived to int
    operator int () { return int () ; } ;
};

// derived from virtual base class, Base
class VDerived : virtual public Base
{
};

// class forward-declarations - incomplete classes
class X ;
class Y ;

void main ()
{
    Derived d ;
    Base* bp = &d ;

    // cast Base* to Derived* using (T)e notation
    Derived* dp0 = (Derived*)bp ;

    // cast Base* to Derived* using static_cast<>() operator
    Derived* dp1 = static_cast<Derived*>(bp) ;

    int i (0) ;
    // cast int to enum
    Boolean boolean = static_cast<Boolean>(i) ;

    // cast from Derived to int using Derived conversion
    // function
    int i_obj0 = (Derived)d ;
    int i_obj1 = static_cast<Derived>(d) ;
```

```

// cast using undefined types X and Y
X* xp ;
Y* yp0 = (Y*)xp ; // o.k.
Y* yp1 = static_cast<Y*>(xp) ; // error

// cast from virtual base class
VDerived* vdp0 = (VDerived*)bp ; // error
VDerived* vdp1 = static_cast<VDerived*>(bp) ; // error

// cast an object to a reference type
Base b ;
Derived& dp4 = (Derived&)b ; // o.k.
Derived& dp5 = static_cast<Derived&>(b) ; // o.k.

// cast-away const
const Base* cbp ;
Derived* dp2 = (Derived*) cbp ; // o.k.
Derived* dp3 = static_cast<Derived*>(cbp) ; // error
}

```

### 16.4.2 The reinterpret\_cast<>() Operator

The `reinterpret_cast<T>(e)` operator allows a programmer to perform casts which are generally unsafe and implementation-dependent, with `reinterpret_cast<>()` returning a reinterpretation of its operand. An integral type can be converted to a pointer type (e.g. `int` to `char*`) and vice versa using `reinterpret_cast<>()`. Unlike `static_cast<>()` it is possible with `reinterpret_cast<>()` to perform both pointer and reference type conversions with types that are not complete at the time of compilation. As with `static_cast<>()`, `reinterpret_cast<>()` does not cast-away the `const` attribute of an object. A pointer of a given type to a function can be cast to a function pointer of another type using `reinterpret_cast<>()`. The program below illustrates a few properties of `reinterpret_cast<>()`:

```

// rein_cst.cpp
// illustrates the reinterpret_cast<>() operator
#include <iostream.h> // C++ I/O

class Base
{
};

// derived from Base
class Derived : public Base
{
};

// class forward-declarations - incomplete classes
class X ;
class Y ;

void main ()

```

```

{
int* ip ;

// cast int* to char* using (T)e notation
char* cp0 = (char*)ip ;

// cast int* to char* using reinterpret_cast<>() operator
char* cp1 = reinterpret_cast<char*>(ip) ;

Derived* dp0 = (Derived*)ip ;
Derived* dp1 = reinterpret_cast<Derived*>(ip) ;

// cast using undefined types X and Y
X* xp ;
Y* yp0 = (Y*)xp ; // o.k.
Y* yp1 = reinterpret_cast<Y*>(xp) ; // o.k.

// cast-away const
const Base* cbp ;
Derived* dp2 = (Derived*) cbp ; // o.k.
Derived* dp3 = reinterpret_cast<Derived*>(cbp) ; // error

void Function (int) ;
typedef void (*pfv) () ;

// cast pointer to function of one type to another type
pfv p_function0 = (pfv)Function ; // o.k.
pfv p_function1 = reinterpret_cast<pfv>(Function) ; // o.k.

p_function0 () ;
p_function1 () ;
}

void Function (int i)
{
cout << "Function () called" << endl ;
}

```

### 16.4.3 The `const_cast<>()` Operator

The `const_cast<T>(e)` operator allows access to an object with the `const` or `volatile` attribute attached. Type T must be the same type as operand e except for `const` and `volatile` modifiers, and the result returned by `const_cast<>()` is the same as e but of type T. The `const_cast<>()` operator also respects the `constness` of an object's `const` attribute. The following program helps illustrate `const_cast<>()`:

```

// cnst_cst.cpp
// illustrates the const_cast<>() operator
#include <iostream.h> // C++ I/O

class Base

```

```

{
};

// derived from Base
class Derived : public Base
{
};

void Function (char* cp)
{
}

void main ()
{
// cast-away const
const Base* cbp ;
Derived* dp2 = (Derived*) cbp ; // o.k.
Derived* dp3 = const_cast<Derived*>(cbp) ; // error

const char* ccp = "string" ;
Function (ccp) ; // error: can't cast const char* to char*
Function ((char*)ccp) ; // o.k.
Function (const_cast<char*>(ccp)) ; // o.k.

volatile char* vc ;
Function ((char*)vc) ;
Function (const_cast<char*>(vc)) ;

const volatile char* cvcp ;
Function ((char*)cvcp) ;
Function (const_cast<char*>(cvcp)) ;
}

```

## 16.5 Summary

The C++ run-time type information mechanism provides support for resolving an object's type at run-time. The `dynamic_cast<>()` operator can have one of two forms: pointer or reference. `dynamic_cast<T*>()` returns a pointer to an object of a derived `class` given a pointer to a base `class` if, and only if, the type of the object pointed to matches that of the derived `class`; else 0 is returned. The `dynamic_cast<T&>()` operator performs a similar operation for reference types, except that if a cast to a reference is unsuccessful a `Bad_cast` exception is thrown. The `typeid()` operator allows the exact type of an object to be extracted at run-time. If the `typeid()` operator is unsuccessful in identifying the type of its operand it throws a `Bad_typeid` exception. The `typeid()` operator returns a reference to an object of the polymorphic `class type_info`, which supports equality and inequality comparisons and `before()` and `name()` member functions.

C++ also supports three operators for performing various types of casting: `static_cast<>()`, `reinterpret_cast<>()` and `const_cast<>()`. These three

cast operators act as a replacement for the C-style `(T) e cast` notation, each performing its own specific operation.

Regarding both run-time type information and casting, C++'s long-term philosophy is to eliminate both and rely exclusively on the static type-checking mechanism because it is inherently less error-prone and generally results in better program style. At present it is not always possible to develop programs without reference to RTTI and casting. If this is the case, try to adopt the `dynamic_cast<>()` and `typeid()` operator approaches rather than `virtual` functions for extracting run-time type information and the `static_cast<>()`, `reinterpret<>()` and `const_cast<>()` style of casting rather than the traditional `(T) e cast` style.

## Exercises

- 16.1 Use the `dynamic_cast<T*>()` operator to cast a pointer to `Base` to a pointer to `Derived`:

```
class Base
{
public:
    virtual void Function () ;
};

class Derived : public Base
{
};
```

- 16.2 For the `Base` and `Derived` classes of Exercise 16.1 use the `dynamic_cast<T&>()` operator to cast a reference to `Base` to a reference to `Derived`:

- 16.3 Use the `typeid()` operator to determine exactly which `class` derived from `Base` the `bp` pointer is pointing to:

```
void main ()
{
    Base* bp ;
    //...
}
```

- 16.4 Use the `(T) e` style cast to cast the `Base class` pointer, `bp`, to a `Derived class` pointer:

```
class Base
{
};

class Derived : public Base
{
};

void main ()
{
    Base* bp ;
```

```
//...
}
```

16.5 Rather than using the `(T)e` style cast of Exercise 16.4 use the `static_cast<>()` operator.

16.6 Use the `reinterpret_cast<>()` operator to cast the following pointer to a function, `pFv`, to point to the `Sqrt()` function:

```
double Sqrt(double) ;

void main ()
{
    typedef void (*pFv) () ;
//...
}
```

16.7 The following program results in a compilation error because it is not possible to convert from a `const char*` to a `char*`. Use the `const_cast<>()` operator to remove this compilation error and hence determine the length of string:

```
// returns the length of a string
int StrLength (char* string)
{
//...
}

void main ()
{
    const char* string = "MEMORY_ERROR" ;
//...
cout << "string length: " << StrLength (string) ;
}
```

# **Input and Output, Files and Streams**

*from atoms to bits*  
Negroponte (1995)

*The vast majority of computer programs involve some degree of input or output or both. This chapter examines the C++ approach to the input and output of objects.*

*To begin with, the notion of a stream is discussed. The standard input and output streams are the keyboard and the display screen, respectively. The C++ language offers a programmer an extensive input and output class library, with high-level classes such as istream and ostream for the manipulation of input and output.*

*Ever since the simplest of programs in Chapter 3 we have seen the repeated use of the input and output extraction and insertion operators >> and <<. The use of >> and << for both integral and user-defined types is examined more closely in the present chapter. The formatting of input and output is essential for achieving good program-user interaction. In C++, input and output can be formatted either by the use of formatting functions or manipulators which can be both non-parametrised or parametrised and specifically defined for user-defined classes.*

*The reading from and writing to disk of objects of integral and user-defined types is covered, along with the associated C++ file stream classes and their operations. User-defined file stream classes are developed which are derived from the C++ file stream classes. This is then followed by spending some time discussing the development of a slightly larger class called World, which reads and writes a set of geometric objects from disk in a purely free-format manner.*

*C++ supports the redirection of standard input and output to alternative devices. The main() function is revisited from the point of view of command-line arguments. For console-based applications, C++ provides a library class called constream with several corresponding console stream manipulators.*

*Although not object-oriented, the latter half of the chapter presents an overview of the C function-based stream mechanism because it is so well supported and still widely used in C/C++ programming. The chapter concludes by developing a device-independent bitmap class, DIBitmap, for Windows. The class is capable of reading from and writing to file bitmaps of 1, 4, 8 or 24 bits per pixel.*



## 17.1 Why Wait Until Now?

Until now we have made good use of the standard input and output stream objects `cin` and `cout`, respectively, with little explanation of the input and output **class** libraries from which they are defined. Also, the topics of writing and reading objects to and from a disk file have somehow been bypassed except for a brief discussion in Chapter 4. This is because input and output are not central to the object-oriented and **class** concepts of the C++ language, and to appreciate the C++ input and output mechanism requires a reasonably good understanding of *the object* and inheritance. Furthermore, the C++ stream classes form a comprehensive **class** library, so don't worry if it takes you a great deal of time and effort before you feel comfortable with the library. I certainly recommend referring to your compiler's **class** library reference manual for detailed information regarding the data members and member functions of each of the stream classes.

## 17.2 Streams and Stream Classes

A frequently used term when discussing input and output is the *stream*. A stream is an abstraction which refers to a common interface to the various input and output devices of a computer. There are two forms of stream: text and binary. Text streams are used with ASCII characters, whereas binary streams can be used with any type of data. The synonyms *extracting* or *getting* are generally used when referring to the input of data from a device and *inserting* or *putting* when referring to the output of data to a device.

The C++ stream classes form a **class** hierarchy shown in Fig. 17.1. It is seen that the stream **class** hierarchy is divided into two main areas: classes derived from `streambuf` and classes derived from `ios`. Note that access from `ios`-based classes to `streambuf`-based classes is obtained via a pointer declared in `ios`. The following sections briefly describe the `ios` and `streambuf` base classes.

### 17.2.1 The `ios` Base **class**

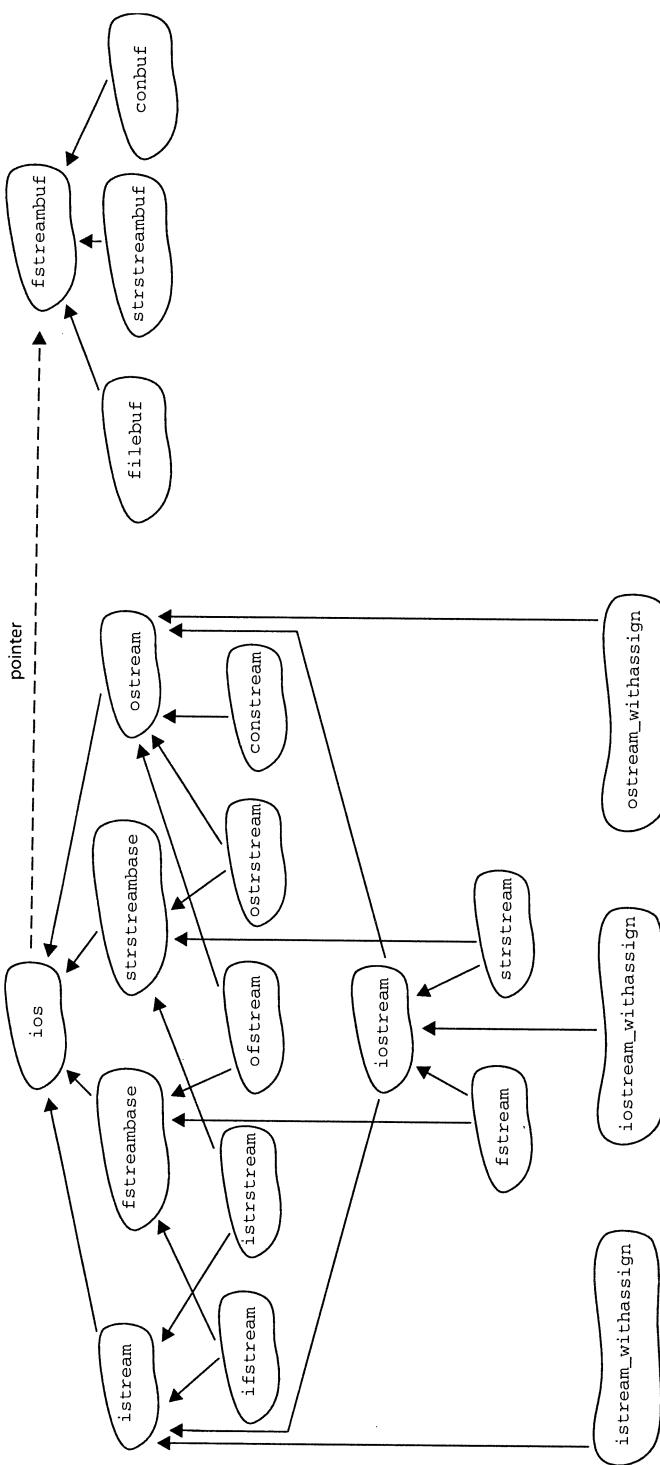
The `ios` **class** provides methods common to both formatted input and output with error checking. The classes `istream`, `fstreambase`, `strstreambase` and `ostream` are all derived from `ios`.

### 17.2.2 The `streambuf` Base **class**

The `streambuf` **class** provides a buffer interface between data and storage areas, such as memory and physical devices, and supports low-level methods for handling streams and data buffering when negligible formatting is required. The buffering classes `filebuf`, `strstreambuf` and `conbuf` are all derived from `streambuf`.

### 17.2.3 Stream Classes Derived from `ios` and `streambuf`

Figure 17.1 illustrates that the derivation of classes from `ios` and `streambuf` is complicated, so the following sections briefly describe the most frequently used classes, `istream`, `ostream`, `iostream`, `ifstream`, `ofstream`, `fstream`, which are derived from the `ios` base **class**.



**Fig.17.1** C++ stream **class** hierarchy illustrating base classes `ios` and `streambuf` and their derived classes.

### The **istream** class

Class **istream** allows you to define an input stream and supports methods for both formatted and unformatted input. The extraction operator, `>>`, is overloaded for all C++ integral data types, thus making high-level input operations possible. Classes **ifstream**, **istrstream**, **iostream** and **istream\_withassign** are all derived from **istream**.

It is worth noting that **class istream** is derived from **class ios** using **virtual** inheritance to avoid multiple **class** member declarations when **class iostream** is derived from both **istream** and **ostream** using multiple inheritance:

```
class istream : virtual public ios // ISTREAM.H
{
//...
};
```

### The **ostream** class

Similarly, the **ostream** **class** allows a user to define an output stream and supports methods for both formatted and unformatted output. The insertion operator, `<<`, is overloaded for all C++ integral data types. Classes **ofstream**, **ostrstream**, **constream**, **iostream** and **ostream\_withassign** are all derived from **ostream**.

As with **istream**, **class ostream** is **virtually** derived from **class ios** to avoid multiple declarations when declaring **iostream**:

```
class ostream : virtual public ios // ISTREAM.H
{
//...
};
```

### The **iostream** class

The **iostream** **class** is derived from **istream** and **ostream**, and as a result caters for both input and output of a stream. Classes **fstream** and **strstream** are both derived from **iostream**.

Since **class iostream** is derived from both **istream** and **ostream**, **iostream** is declared using multiple inheritance:

```
class iostream : public istream,
                public ostream // ISTREAM.H
{
//...
};
```

### The **ifstream** class

The **ifstream** **class** is derived from **istream** and **fstreambase**, and therefore allows file input of a stream. The **class** declaration of **ifstream** can be found in **FSTREAM.H** and is of the form:

```
class ifstream : public fstreambase,
                 public istream // FSTREAM.H
```

---

```
{
//...
};
```

### The ofstream class

The ofstream **class** is derived from ostream and fstreambase, and therefore allows file output of a stream:

```
class ofstream : public fstreambase,
                public ostream // FSTREAM.H
{
//...
};
```

### The fstream class

The fstream **class** is derived from iostream and fstreambase and therefore caters for both file input and output of a stream:

```
class fstream : public fstreambase,
                public iostream // FSTREAM.H
{
//...
};
```

## 17.2.4 Header Files

Table 17.1 associates each stream **class** with its appropriate header file.

CONSTREA.H provides support for console output only, IOSTREAM.H memory buffers, FSTREAM.H files and STRSTREA.H for strings.

I recommend examining the stream **class** declarations of the four header files listed in Table 17.1 in more detail. Since the manipulation of stream objects in C++ relies heavily on standard C++ stream libraries, and because the libraries are presently being standardised, I also recommend that you consult the ANSI/ISO draft standard, Standard (1996). Alternatively, a very good book which covers the C++ draft standard library in detail is *The Draft Standard C++ Library* (Plauger, 1995).

## 17.2.5 Predefined Streams

C++ defines four predefined *open* streams in the IOSTREAM.H header file:

**Table 17.1** C++ stream header files and associated stream classes.

Header file	Stream classes
CONSTREA.H	conbuf, constream
IOSTREAM.H	ios, iostream, iostream_withassign, istream, istream_withassign, ostream, ostream_withassign, streambuf
FSTREAM.H	filebuf, fstream, fstreambase, ifstream, ofstream
STRSTREA.H	istrstream, oistrstream, strstream, strstreambase, strstreambuf

```
extern istream_withassign cin ;
extern ostream_withassign cout ;
extern ostream_withassign cerr ;
extern ostream_withassign clog ;
```

Each of the stream objects `cin`, `cout`, `cerr` and `clog` is defined as an object of one of the `_withassign` classes. As we have seen throughout previous chapters, the `cin` and `cout` stream objects are associated with standard input and standard output respectively. The `cerr` and `clog` stream objects are generally used for error handling in which an object is sent to the standard output device. The only difference between `cerr` and `clog` is that `cerr` outputs an object immediately whereas `clog` buffers its output.

## 17.3 Standard Stream Output

Consider the following program, which outputs various objects of C++ integral types using the `cout` stream object:

```
// cout.cpp
// illustrates the cout stream object and insertion operator <<
#include <iostream.h> // C++ O/P

void main ()
{
    char string[] = "hi there!" ;
    int i (1) ;
    double d (1.25) ;

    cout << string << endl ;
    cout << i << endl ;
    cout << d << endl ;

    cout << "string: " << string << "; i: " << i
        << "; d: " << d << endl ;

    cerr << d << endl ;
    clog << i << endl ;

    // void* data type
    cout << &d << endl ;
}
```

with output to the display screen:

```
hi there!
1
1.25
string: hi there!; i: 0; d:1.25
1.25
1
```

---

0x2404

As we have seen on numerous occasions in previous chapters, stream output is achieved by a combination of the `cout` stream object and the insertion, `<<`, operator. Operator `<<` is the left shift binary operator, which has been overloaded for all C++ integral data types and can be overloaded for user-defined types. The left-hand operand of `<<` is an object of **class ostream** while the right-hand operand is an object for which stream output has been defined.

The `<<` operator is left-associative and returns a reference to the `ostream` object which invoked the operator. Returning a reference allows stream objects to be cascaded, as illustrated above in `COUT.CPP`.

By default, the standard output stream is directed to the console. However, we shall see later that output streams can be redirected to alternative devices, such as disk files and printers.

Program `COUT.CPP` also illustrates the use of output stream objects `cerr` and `clog`. Note that the `void*` overloaded insertion operator is called for displaying the memory address, in hexadecimal, of object `d`.

## 17.4 Standard Stream Input

To illustrate basic stream input of integral data types consider the program:

```
// cin.cpp
// illustrates the cin stream object and extraction operator >>
#include <iostream.h> // C++ I/O

void main ()
{
    char* string (new char(80)) ;
    int i (0) ;
    double d (0.0) ;

    cout << "enter a string: " ;
    cin >> string ;

    // skip 80 char. in input stream; stop if newline encountered
    //cin.ignore (80, '\n') ;

    cout << "enter an int and a double: " ;
    cin >> i >> d ;

    cout << "string: " << string << ";" i: " << i
                                << ";" d: " << d << endl ;
}
```

with some user interaction:

```
enter a string: hi there!
enter an int and a double: 1 1.25
string: hi; i: 1; d: 1.25
```

As with stream output, stream input is a combination of the `cin` stream object and the extraction, `>>`, operator. The left-hand operand of `>>` is an object of **class** `istream`, while the right-hand operand is an object for which the stream input has been defined. The `>>` operator is left-associative and returns a reference to the `istream` object which invoked the operator, thus allowing input stream objects to be cascaded.

By default, the standard input stream is from the keyboard. As with output streams, an input stream can be redirected so that input arrives from a device other than the keyboard.

By default the `>>` operator extracts input from the `istream` and then transfers the input to the right-hand object until a white space character<sup>1</sup> (such as a space, tab, vertical tab, carriage return, new line or form feed) is encountered. This is why the string object displayed in CIN.CPP consists only of the first two letters 'hi' of the total 'hi there!' string entered, since a space exists between 'hi' and 'there!'. Any entered characters will remain in the `istream` and will thus be read by subsequent input streams. In CIN.CPP this is not what was intended, and hence the reason for the statement:

```
cin.ignore (80, '\n') ;
```

Function `ignore()` is a member function of **class** `istream` and has the following signature:

```
istream& ignore (int n=1, int delim=EOF) ; // Iostream.h
```

It skips over `n` characters in the input stream and stops if `delim` is encountered before the `n` characters. The call to `istream::ignore()` in CIN.CPP assumes that at most 80 characters will be entered on a single line and the delimiter is a new line.

Try commenting out the call to `istream::ignore()`. The output will now be:

```
enter a string: hi there!  
enter an int and a double: string: hi; i: 0; d: 0
```

A user is not given the opportunity to enter integer and floating-point numbers. After the string 'hi there!' is entered, followed by a carriage return, the second line shown above is immediately displayed.

In addition, note that by default the `>>` operator skips any leading white space. As an illustration, execute CIN.CPP once more, but this time press carriage return several times before entering the string 'hi there!' and press the spacebar several times before entering integer and floating-point numbers:

```
enter a string:  
  
hi there!  
enter an int and a double: 1 1.25  
string: hi; i: 1; d: 1.25
```

White space can be controlled by the `ios::skipws` flag, which skips white space on input. The `skipws` flag can be set using the `ios::setf()` member function and unset with the

---

<sup>1</sup> To test whether a given character is a white space character use the `isspace()` (`int isspace(int c) ;`) C++ library function, which returns non-zero if `c` is a white space character.

`ios::unsetf()` member function. The `ws` manipulator can also be used to skip leading white space.

The member functions of classes `istream` and `ios`, `ios` flags and manipulators will be discussed in more detail later in the chapter.

## 17.5 Overloading the Insertion and Extraction Operators

In Chapters 10 and 11 we examined the overloading of the C++ insertion, `<<`, and extraction, `>>`, operators for a given **class**. Both the insertion and extraction operators are overloaded for C++'s integral data types in the `IOSTREAM.H` header file, so that the appropriate input or output function is called automatically:

```
class istream : virtual public ios // IOSTREAM.H
{
//...
public:
    istream& operator >> (char&) ;
    istream& operator >> (signed char&) ;
//...
    istream& operator >> (int&) ;
    istream& operator >> (float&) ;
    istream& operator >> (double&) ;
//...
};

//...
class ostream : virtual public ios // IOSTREAM.H
{
//...
public:
    ostream& operator << (char&) ;
    ostream& operator << (int&) ;
    ostream& operator << (float&) ;
    ostream& operator << (double&) ;
//...
};

//...
```

The C++ integral data types supported are `char`, `char*` (null terminated string), `short`, `int`, `long`, `float`, `double`, `long double` and the pointer to `void: void*`.

Similarly, `<<` and `>>` can be overloaded for a user-defined **class**. As an illustration let's examine overloading `<<` and `>>` for a basic `Point` **class**:

```
// usr_type.cpp
// illustrates overloading the insertion and extraction
// operators
// for a user-defined class
#include <iostream.h> // C++ I/O

class Point
```

```

{
private:
    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // overloaded operators/friends
    friend ostream& operator << (ostream& s,
                                    const Point& p) ;
    friend istream& operator >> (istream& s, Point& p) ;
}; // class Point

ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
}

istream& operator >> (istream& s, Point& p)
{
    return s >> p.x >> p.y >> p.z ;
}

void main ()
{
    Point p, q (4.0, 5.0, 6.0) ;

    cout << "enter a Point object p: " ;
    cin >> p ;
    cout << "Point objects p and q: " << p << " " << q ;
}

```

with some user interaction:

```

enter a Point object p: 1.0 2.0 3.0
Point objects p and q: (1, 2, 3) (4, 5, 6)

```

Provided the << and >> operators are appropriately overloaded for **class** Point, the use of **cin** and >> and **cout** and << for objects of **class** Point is identical to that for integral data types.

On examining **Point::operator<<()** we observe that it is declared a **friend** of **class** Point, which allows **class** ostream access to Point's **private** data members. The **operator<<()** member function returns a reference to ostream, which is necessary to enable cascading of multiple objects. Since << is a binary operator and **operator<<()** is a **friend** function it has two arguments. The first is a reference to an ostream object and is the left-hand operand of <<. The second is a **const** reference to a Point object which is the right-hand operand of <<. The single statement of **operator<<()** outputs the x, y and z data members of a Point object, each separated by a comma, and with all three data members enclosed within a pair of parentheses.

The declaration and definition of **operator>>()** for **class Point** is similar to **operator<<()** except that it uses **class istream** instead of **ostream** and a non-**const** **Point class** object as second argument.

It is tempting when defining **operator<<()** to use the **cout** output stream object instead of the argument output stream object **s**:

```
ostream& operator << (ostream& s, const Point& p)
{
    cout << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
    return s ;
}
```

This works fine provided you only intend to use **Point::operator<<()** for standard stream output. The whole idea behind passing a reference to an **ostream** object as first argument is that it allows **Point::operator<<()** to operate on any **ostream** object and not just **cout**.

Similarly, it is equally tempting to place user prompts within the definition of **operator>>()** which are hard-wired to **cout**:

```
istream& operator >> (istream& s, Point& p)
{
    cout << "enter a Point object: x y z: " ;
    return s >> p.x >> p.y >> p.z ;
}
```

You may have noticed from the program **USR\_TYPE.CPP** that there is an incompatibility between the overloaded insertion and extraction operators for **class Point**. The extraction operator expects the **x, y** and **z** data members of **class Point** to be in the format **1 2 3**, whereas the insertion operator outputs the data members in the format **(1, 2, 3)**. For the standard input and output streams this may not be a problem, but in the case of file streams, which we shall examine later, this incompatibility in format will result in problems when writing and reading **Point** objects to and from a disk file. Therefore, let's modify the overloaded extraction operator of **class Point** presented in **USR\_TYPE.CPP**:

```
// in_out.cpp
// further illustrates overloading the insertion
// and extraction operators for class Point
#include <iostream.h> // C++ I/O

class Point
{
}; // class Point
//...
istream& operator >> (istream& s, Point& p)
{
    char left ('('), right (')'), comma (',') ;
    return s >> left
        >> p.x >> comma >> p.y >> comma >> p.z
        >> right ;
}
```

```
void main ()
{
    Point p ;

    cout << "enter a Point object p: " ;
    cin  >> p ;
    cout << "Point object p: " << p ;
}
```

With some user interaction:

```
enter a Point object p: (1,2,3)
Point object p: (1, 2, 3)
```

The overloaded extraction operator of **class** Point now defines three temporary objects, left, right and comma, to enable the input format to consist of the x, y and z data members each separated by a comma and enclosed within a pair of parentheses.

## 17.6 Formatted Input and Output

C++ allows a programmer complete control over the formatting of input and output. In general there are two ways in which formatting can be specified by a programmer: manipulators and member functions of **class** ios. Let's first examine the manipulator approach.

### 17.6.1 Formatting via Manipulators

Throughout previous chapters we have frequently seen the use of *manipulators*, such as endl, to alter the format of a stream. Manipulators are function-like operators and take a stream reference as argument and return a reference to the same stream and can therefore be used in cascading several insertions or extractions. For instance, the declaration of the endl manipulator can be found in the IOSTREAM.H header file and is of the form:

```
ostream& endl (ostream& s) ; // IOSTREAM.H
```

Table 17.2 lists the stream manipulators available in C++.

Note that the non-parametrised manipulators (dec, endl, ends, flush, hex, oct and ws) are declared in the IOSTREAM.H header file, whereas the other parametrised manipulators (*resetiosflags()*, *setbase()*, *setfill()*, *setiosflags()*, *setprecision()* and *setw()*) are declared in IOMANIP.H.

The following program illustrates typical applications of several manipulators:

```
// manip.cpp
// illustrates the C++ stream manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // C++ manipulators

void main ()
{
```

**Table 17.2** C++ stream manipulators.

<i>Manipulator</i>	<i>Action</i>
<code>dec</code>	Decimal conversion base
<code>endl</code>	Insert new line and flush an <code>ostream</code>
<code>ends</code>	Insert NULL terminator in string
<code>flush</code>	Flush an <code>ostream</code>
<code>hex</code>	Hexadecimal conversion base
<code>oct</code>	Octal conversion base
<code>ws</code>	Skip leading white space
<code>resetiosflags (long f)</code>	Turn off format flags specified by <code>f</code>
<code>setbase (int n)</code>	Set conversion base format flag to <code>n</code>
<code>setfill (int c)</code>	Set fill character to <code>c</code>
<code>setiosflags (long f)</code>	Turn on format flags specified by <code>f</code>
<code>setprecision (int p)</code>	Set floating-point precision to <code>p</code>
<code>setw (int n)</code>	Set field width to <code>n</code>

```

int      i (1), j (11) ;
double   d1 (1.0), d125 (1.25), d12345 (1.2345) ;

cout << i << endl ;
cout << d1 << " " << d125 << " " << d12345 << endl ;

cout << setprecision (3)
    << d1 << " " << d125 << " " << d12345 << endl ;

cout << setprecision (4)
    << setw (7) << d1
    << setw (7) << d125
    << setw (7) << d12345
    << endl ;

cout << setfill ('?')
    << setw (7) << d1
    << setw (7) << d125
    << setw (7) << d12345
    << endl ;

cout << dec << j << " "
    << hex << j << " "
    << oct << j << " "
    << endl ;

cout << setbase (16)
    << j
    << endl ;

cout << dec << j << endl ; // reset conversion base

cout << flush ;
}

```

with output:

```
1
1 1.25 1.2345
1 1.25 1.23
    1   1.25  1.234
??????1????1.25??1.234
11 b 13
b
11
```

In addition to the C++ manipulators it is possible to define your own manipulators. The user-defined manipulators can be either non-parametrised or parametrised. Let's first examine the slightly easier case of non-parametrised manipulators by considering the following program, which illustrates two user-defined manipulators `PointPrompt` and `PointFormat` for the input and output, respectively, of objects of user-defined **class** `Point`:

```
// my_manip.cpp
// illustrates user-defined non-parametrised stream manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // C++ manipulators

class Point
{
private:
    double x, y, z;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    // overloaded operators/friends
    friend ostream& operator << (ostream& s,
                                    const Point& p) ;
    friend istream& operator >> (istream& s, Point& p) ;
}; // class Point

ostream& operator << (ostream& s, const Point& p)
{
    return s << "(" << p.x << ", " << p.y << ", " << p.z << ")" ;
}

istream& operator >> (istream& s, Point& p)
{
    return s >> p.x >> p.y >> p.z ;
}

// user-defined output manipulator
ostream& PointFormat (ostream& s)
{
```

```

return s << setiosflags (ios::fixed)
    << setprecision (2)
    << setw (2) ;
}

// user-defined input manipulator
istream& PointPrompt (istream& s)
{
    cout << "enter a Point object; x y z: " ;
    return s ;
}

void main ()
{
    Point p, q (4.0, 5.0, 6.0) ;

    cin >> PointPrompt >> p ;

    cout << PointFormat << p << PointFormat << q << endl ;
}

```

with some user interaction:

```

enter a Point object; x y z: 1 2 3
(1.00, 2.00, 3.00) (4.00, 5.00, 6.00)

```

The `PointPrompt` manipulator simply prompts a user to enter the `x`, `y` and `z` data members of a `Point` object and is defined as:

```

istream& PointPrompt (istream& s)
{
    cout << "enter a Point object; x y z: " ;
    return s ;
}

```

The form of the manipulator is characteristic of a user-defined non-parametrised manipulator function in that it has a single argument which is a reference to the stream object that it is operating on, and that the manipulator function returns a reference to the same stream object.

The definition of the output manipulator `PointFormat` is similar in outline to `PointPrompt`, but illustrates that user-defined manipulators are useful for compressing several other manipulators into a single manipulator.

It is also possible in C++ to define parametrised manipulators. As an example let's revisit the `endl` manipulator mentioned above so that a variable number of new lines can be inserted into a stream instead of a single new line. The key problem in parametrising the manipulator is the limitation of the overloaded insertion, `<<`, operator syntax. The overloaded insertion operator is a binary operator with an `ostream` object as the left-hand operand and an object to be output as the right-hand operand:

```

ostream& operator << (ostream& s, const Point& p) ;

```

with a typical use:

```
Point p ;
//...
cout << p << endl ;
```

Hence there is no way of attaching an additional argument to **operator<<** () which specifies the number of new lines to be inserted. One solution is to declare an additional **class**, say X, which encapsulates both an additional argument, data, a data member and a pointer to a function, pf, data member:

```
// p_manip.cpp
// illustrates user-defined parametrised stream
// manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // C++ manipulators

class X
{
private:
    int data ;
    ostream& (*pf) (ostream&, int) ;
public:
    X (ostream& (*pf_arg) (ostream&, int), int data_arg)
        : pf (pf_arg), data (data_arg) {}
    friend ostream& operator << (ostream& s, const X& x)
        { return (*x(pf)) (s, x.data) ; }
}; // class X

ostream& insert_lines (ostream& s, int n)
{
    for (int i=0; i<n; i++)
        s << endl ;
    return s ;
}

X endl (int n)
{
    return X (insert_lines, n) ;
}

class Point
{
//...
}; // class Point
//...
void main ()
{
    Point p, q (1.0, 1.0, 1.0), r (2.0, 2.0, 2.0) ;

    cout << p << endl ;           // non-parametrised
    cout << q << endl (3) << r ; // parametrised
}
```

with output:

```
(0, 0, 0)
(1, 1, 1)
```

```
(2, 2, 2)
```

The overloaded insertion operator for **class X** calls a function specified in the definition of an object of **class X**:

```
friend ostream& operator << (ostream& s, const X& x)
{ return (*x.pf)(s, x.data) ; }
```

To define a parametrised version of the endl manipulator it is a simple matter of now overloading endl so that it is passed a variable argument which indicates the number of new lines to be inserted into the stream and returns an object of **class X**, which in turn calls (via the function pointer) a hidden function, insert\_lines, which actually inserts the specified number of new lines into the stream:

```
X endl (int n)
{
return X (insert_lines, n) ;
}

ostream& insert_lines (ostream& s, int n)
{
for (int i=0; i<n; i++)
    s << endl ;
return s ;
}
```

Because endl was overloaded it is still possible to use the original endl manipulator declared in IOSTREAM.H.

An alternative approach to user-defined parametrised manipulators is to make use of the **template imanip, omanip and smanip** classes declared in IOMANIP.H:

```
template <class T>
class imanip
{
private:
    T data ;
    istream& (*pf)(istream&, T) ;
public:
    imanip (istream& (*pf_arg)(istream&, T), T data_arg)
        : pf(pf_arg), data(data_arg) {}
    friend istream& operator >> (istream& s, imanip<T>& m)
        { return (*m.pf)(s, m.data) ; }
};

template <class T>
```

```

class omanip
{
private:
    T data ;
    ostream& (*pf)(ostream&, T) ;
public:
    omanip (ostream& (*pf_arg)(ostream&, T), T data_arg)
        : pf (pf_arg), data (data_arg) {}
    friend ostream& operator << (ostream& s, omanip<T>& m)
        { return (*m(pf))(s, m.data) ; }
};

template <class T>
class smanip
{
private:
    T data ;
    ios& (*pf)(ios&, T) ;
public:
    smanip (ios& (*pf_arg)(ios&, T), T data_arg)
        : pf (pf_arg), data (data_arg) {}
    friend istream& operator >> (istream& s, smanip<T>& m)
        { (*m(pf))(s, m.data) ; return s ; }
    friend ostream& operator << (ostream& s, smanip<T>& m)
        { (*m(pf))(s, m.data) ; return s ; }
};

```

On comparing the **class** declarations of omanip and X in P\_MANIP.CPP we observe that the declarations are of identical format except that X has been replaced by omanip and **int** replaced by the parametrised type T. The imanip, omanip and smanip **template** classes declared in IOMANIP.H can be used to create user-defined parametrised input, output, and input and output stream manipulators respectively. Their operation is identical to that outlined for **class** X in P\_MANIP.CPP in that a parametrised manipulator calls (via a pointer to a function) a user-defined function which actually does the work. The following program again parametrises the endl manipulator, as in P\_MANIP.CPP, but using omanip instead of X:

```

// p_manip1.cpp
// an alternative approach to user-defined
// parametrised stream manipulators
#include <iostream.h> // C++ I/O
#include <iomanip.h> // C++ manipulators

ostream& insert_lines (ostream& s, int n)
{
    for (int i=0; i<n; i++)
        s << endl ;
    return s ;
}

omanip<int> endl (int n)
{

```

```

return omanip<int> (insert_lines, n) ;
}

class Point
{
//...
}; // class Point
//...
void main ()
{
    Point p, q (1.0, 1.0, 1.0), r (2.0, 2.0, 2.0) ;

    cout << p << endl ;           // non-parametrised
    cout << q << endl (3) << r ; // parametrised
}

```

with output identical to that generated by P\_MANIP.CPP.

### 17.6.2 Formatting via Member Functions of **class ios**

The previous subsection illustrated the use of an **ios** format flag, **ios**::**fixed**, with the C++ **setiosflags()** manipulator in the definition of the user-defined manipulator **PointFormat**. The **ios** format flags determine how both input and output are to be formatted and are generally specified as an enumerated **long** integer in **Iostream.h**:

```

enum
{
    skipws      = 0x0001, // skip whitespace
    left         = 0x0002, // left-justified
    right        = 0x0004, // right-justified
    internal     = 0x0008, // padding
    dec          = 0x0010, // decimal conversion
    oct          = 0x0020, // octal conversion
    hex          = 0x0040, // hexadecimal conversion
    showbase     = 0x0080, // show base
    showpoint    = 0x0100, // show decimal point
    uppercase    = 0x0200, // use uppercase for e and x
    showpos      = 0x0400, // show positive sign
    scientific   = 0x0800, // use scientific notation
    fixed        = 0x1000, // use floating-point notation
    unitbuf      = 0x2000, // flush O/P streams
    stdio        = 0x4000 // C I/O compatibility
};

```

When a format flag is set, the respective formatting feature is turned on; otherwise the formatting feature is turned off and default formatting is performed.

Let's describe each of the above **ios** format flags briefly. The **skipws** flag skips leading white space for stream input and is set by default. The **left** and **right** flags left- and right-justify stream output respectively. The **internal** flag pads the field of a numeric number after the sign or base indicator of the number. The flags **dec**, **hex** and **oct** are the

decimal, hexadecimal and octal conversion bases respectively. The `showbase` flag ensures that the base of a numeric number is displayed, whereas `showpoint` displays the decimal point for floating-point numbers. When the `uppercase` flag is set, the lowercase letters `e` and `x` for scientific notation and hexadecimal numbers will be displayed in uppercase. The `showpos` flag causes a positive sign to prefix positive numeric numbers. The `scientific` flag causes a numeric number to be displayed using scientific notation, whereas the `fixed` flag causes floating-point numbers to be displayed to six decimal places with a decimal point. When the `unitbuf` flag is set output streams are flushed following insertion. Finally, the `stdio` flag typically provides compatibility with the C language input and output stream system; refer to your compiler's documentation for the exact implementation details of `stdio`.

The `ios` format flags are generally set, unset and read using the `setf()`, `unsetf()` and `flags()` member functions of **class** `ios`. The `ios::setf()` member function comes in two forms:

```
long setf (long f) ;
long setf (long set_bits, long field) ;
```

The first version is the most frequently used and sets the format flag(s) specified by the argument `f` and returns the previous settings of the format flags. The second form of `setf()` clears the flags specified in `field` and sets them to those specified in `set_bits`. The following three constants are declared in `IOSTREAM.H` to be used with `field`:

```
static const long basefield ;           // dec | oct | hex
static const long adjustfield ;         // left | right | 'internal
static const long floatfield ;          // scientific | fixed
```

The declaration of `ios::unsetf()` is:

```
long unsetf (long f) ;
```

and clears the flag(s) specified by `f` and returns the previous settings.

There are two versions of the `ios::flags()` member function:

```
long flags () ;
long flags (long f) ;
```

The first version simply returns the current format flag settings, while the second version sets the flags specified by `f` and returns the previous settings. The call `flags(0)` sets the default values of the formatting flags.

The following program helps to illustrate the `ios` format flags and the `setf()`, `unsetf()` and `flags()` member functions:

```
// ios_f_mf.cpp
// illustrates the format flags and the
// setf(), unsetf() and flags() member functions of
// class ios
#include <iostream.h> // C++ I/O

// display the state of the fifteen ios format flags
void DisplayFlags (long f)
{
```

```

for (long i=0x4000; i; i=i>>1)
    i & f ? cout << "1" : cout << "0" ;
    cout << endl ;
}

void main ()
{
long f = cout.flags () ;

cout << "default format flags: " ;
DisplayFlags (f) ;

cout.setf (ios::showpos | ios::fixed) ;
f = cout.flags () ;
DisplayFlags (f) ;
cout << 10.01 << endl ;

cout.unsetf (ios::showpos) ;
f = cout.flags () ;
DisplayFlags (f) ;

// set the default format flags
cout.flags (0) ;
f = cout.flags () ;
DisplayFlags (f) ;
}

```

with output:

```

default format flags: 01000000000001
01101000000001
+10.010000
01100000000001
00000000000000

```

The program output illustrates that in the present case the default settings for the 15 `ios` format flags is that `skipws` and `unitbuf` are turned on while the other 13 flags are turned off. Note that the use of the bitwise inclusive operator `|` allows two or more format flags to be specified in a single function call.

Formatting can also be specified via the `fill()`, `precision()` and `width()` member functions of **class** `ios`. There are two versions of `ios::fill()`:

```

char fill () ;
char fill (char c) ;

```

The first version simply returns the current fill character, which is a space by default. The second version of `ios::fill()` sets the current fill character to `c` and returns the previous fill character.

The declaration of `ios::precision()` is:

```
int precision (int n) ;
```

and sets the floating-point precision to *n*, returning the previous precision setting. The default number of decimal places is six.

The `ios::width()` member function takes one of two forms:

```
int width () ;
int width (int n) ;
```

You have probably guessed that the first version of `ios::width()` simply returns the current field width setting, whereas the second version sets the field width to *n* and returns the previous field width.

The operations of the member functions `ios::fill()`, `ios::precision()` and `ios::width()` are similar to the `setfill()`, `setprecision()` and `setw()` manipulators, respectively, discussed in the previous subsection.

An interesting application of `ios::width()` is the prevention of overflowing when entering the characters of a fixed-sized string:

```
// width.cpp
// illustrates the ios::width() member function
#include <iostream.h> // C++ I/O

void main ()
{
    char string[10] ;

    // prevent overflow
    cin.width (sizeof (string)) ;

    cin  >> string ;
    cout << string << endl ;
}
```

with some user interaction:

```
abcdefghijklmnp
abcdefghijkl
```

## 17.7 Working with Disk Files

Let's now examine file handling. The C++ stream **class** library contains the three classes `ifstream`, `ofstream` and `fstream`, and associated methods for creating files and handling file input and output. The stream **class** hierarchy shown in Fig. 17.1 illustrates that the input file stream **class** `ifstream` is derived from both `istream` and `fstreambase`, and thus inherits the file stream operations of `fstreambase` and the extraction operations of `istream`, whereas the output file stream **class** `ofstream` is derived from both `ostream` and `fstreambase`, and as a result inherits the file stream operations of `fstreambase` and the insertion operations of `ostream`. The `fstream` **class** is similarly derived from `fstreambase`, but also from `iostream`, and consequently allows for both input and output.

All three classes ifstream, ofstream and fstream are declared in the header file FSTREAM.H, which, incidentally, includes the IOSTREAM.H header file so that you do not need to #include IOSTREAM.H explicitly if header file FSTREAM.H is included.

### 17.7.1 File Input Using class ifstream

To see file handling in action, let us first examine a program called IFSTR.CPP, which reads a file and then proceeds to display the contents of the file to the screen. The program makes use of the ifstream class, so first let's take a look at the class declaration of ifstream and its four constructors:

```
class ifstream : public fstreambase,
                  public istream // FSTREAM.H
{
public:
    // constructors
    ifstream () ;
    ifstream (const char* name, int mode=ios::in,
               int access=filebuf::openprot) ;
    ifstream (int fd) ;
    ifstream (int fd, char* buffer, int buffer_length) ;
    ~ifstream () ;
    filebuf* rdbuf () ;
    void open (const char* name, int mode=ios::in,
               int access=filebuf::openprot) ;
};
```

The no-argument constructor defines an ifstream object which is not attached to a file. The first of the three-argument constructors defines an ifstream object, opens the file of filename name in protected mode and then attaches the object to the file. Note that, by default, the file is not created if it does not exist. The mode and access arguments will be discussed with reference to the *open()* member function later, but note at this point that the default value for mode is ios::in, which indicates input. The one-argument constructor of class ifstream attaches an ifstream object to an open file descriptor fd, while the second of the three-argument constructors of ifstream attaches an ifstream object to an open file descriptor fd using the buffer, buffer, of length buffer\_length.

Member function *rdbuf()* returns a pointer to the filebuf used. The *open()* member function will be examined in more detail later in the present chapter for classes ifstream, ofstream and fstream.

Here is the program IFSTR.CPP:

```
// ifstr.cpp
// illustrates the ifstream stream class for handling file
input
#include <fstream.h> // C++ file I/O

void main ()
{
    char file_name[] = "IFSTR.CPP" ;

    // define an ifstream object
```

```

ifstream in_file (file_name) ;

if (!in_file)
    cerr << "sorry-can't open file \""
        << file_name << "\"" << endl ;

char ch ;

while (in_file)
{
    in_file.get (ch) ; // get a character
    cout << ch ;
}
}

```

which displays itself on the screen.

An object `in_file` is defined of **class** `ifstream` as follows:

```
ifstream in_file (file_name) ;
```

in which `file_name` is an uppercase string. The filename passed to the constructor of `ifstream` can also include a path name; for instance:

```
ifstream in_file ("C:\\BC5\\INCLUDE\\FSTREAM.H") ;
```

Note the use of the double backslash characters (`\\"`). If a path name is not explicitly given it is assumed that the file resides in the current directory. If the file does not exist or cannot be opened the associated stream object is set to zero, which is useful for confirming that the requested file has indeed been opened before you begin to operate on it:

```

if (!in_file)
    cerr << "sorry-can't open file \""
        << file_name << "\"" << endl ;
//...

```

Assuming that the requested file is successfully opened, IFSTR.CPP proceeds to read the contents of the file character by character using the `istream::get()` member function (which `ifstream` inherits from `istream`) and immediately outputs each character to the screen. There are a few versions of `istream::get()`:

```

// extract single character
int      get () ;
istream& get (char& ch) ;
istream& get (signed char& ch) ;
istream& get (unsigned char& ch) ;
// extract character in to a streambuf
istream& get (streambuf& str_buffer, char delim='\\n') ;

// extract single character in to buffer
istream& get (char*          buffer, int len, char
              delim='\\n') ;

```

---

```
istream& get (signed char* buffer, int len, char
               delim='\n') ;
istream& get (unsigned char* buffer, int len, char
               delim='\n') ;
```

The no-argument version extracts a character or end-of-file. The three one-argument versions extract a character from the input stream, place it into ch and return a reference to an istream object. The two-argument version extracts a character into the supplied strembuf object until the delimiter delim is encountered. The three three-argument versions extract a character and place it in buffer until the delimiter delim is encountered (the default delimiter is a new line character) or until (len-1) bytes or characters have been read, or until end-of-file is encountered. The array pointed to by buffer is NULL terminated.

Rather than using the istream::get() member function, which extracts a single character, the istream::getline() member function could have been used, which is similar to get() but extracts characters and places them in an array pointed to by buffer until the delimiter delim is encountered or until (len-1) bytes or characters have been read or until end-of-file is encountered:

```
istream& getline (char* buffer, int len, char
                  delim='\n') ;
istream& getline (signed char* buffer, int len, char
                  delim='\n') ;
istream& getline (unsigned char* buffer, int len, char
                  delim='\n') ;
```

The getline() member function is useful for reading a string that contains white space. Note that istream::getline() extracts the delimiter, whereas istream::get() does not. The following program modifies IFSTR.CPP for istream::getline():

```
// ifstr1.cpp
// illustrates the ifstream stream class for handling file
// input and the istream::getline() member function
#include <fstream.h> // C++ file I/O

void main ()
{
    char file_name[] = "IFSTR1.CPP" ;

    // define an ifstream object
    ifstream in_file (file_name) ;

    if (!in_file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    const int BUF_SIZ (80) ;
    char     buffer[BUF_SIZ] ;

    while (in_file)
    {
```

```

    // get a line of characters
    in_file.getline (buffer, BUF_SIZ-1) ;
    cout << buffer << endl ;
}
}

```

Let's consider another application of the `istream::getline()` member function to read the number of lines of a given file requested by a user:

```

// no_lines.cpp
// illustrates the use of the istream::getline() member
// function to determine the number of lines of a file
#include <fstream.h> // C++ file I/O
#include <cctype.h> // toupper()

void main ()
{
    char file_name[50] ;

    cout << "enter a filename: " ;
    cin >> file_name ;

    // convert file_name entered to uppercase
    for (int i=0; i<50; i++)
        file_name[i] = toupper (file_name[i]) ;

    // define an ifstream object
    ifstream in_file (file_name) ;

    if (!in_file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    const int BUF_SIZ (120) ;
    char     buffer[BUF_SIZ] ;
    int      line_count (0) ; // line count

    while (in_file)
    {
        in_file.getline (buffer, BUF_SIZ-1) ;
        line_count++ ; // increment line count
    }
    cout << "no. of lines of \"" << file_name << "\": "
        << line_count << endl ;
}

```

with some user interaction:

```

enter a filename: no_lines.cpp
no. of lines of "NO_LINES.CPP": 35

```

The above program simply increments a counter, `line_count`, after each line of the input file is extracted by `getline()`. `NO_LINES.CPP` also makes use of the C++ library function `toupper()`<sup>2</sup>:

```
int toupper (int ch) ; // CTYPE.H
```

which translates character `ch` to uppercase if it is not already. Function `toupper()` is used in `NO_LINES.CPP` to convert the filename entered by a user to uppercase before defining the `ifstream` object `in_file`.

Note that the number of lines returned by the above program is only approximate, since several new line characters are usually floating around at the end of a file.

Before discussing `class ostream` there are two additional member functions of `class istream` which deserve a mention: `peek()` and `putback()`, with declarations:

```
int peek () ;
istream& putback (char ch) ;
```

`istream::peek()` returns the next character in an input stream without actually extracting the character from the stream, while `istream::putback()` places a character back into the input stream.

### 17.7.2 File Output Using `class ofstream`

The `ofstream` `class` is designed for handling file output, and has the following declaration:

```
class ofstream : public fstreambase,
                  public ostream // FSTREAM.H
{
public:
    // constructors
    ofstream () ;
    ofstream (const char* name, int mode=ios::out,
              int access=filebuf::openprot) ;
    ofstream (int fd) ;
    ofstream (int fd, char* buffer, int buffer_length) ;
    ~ofstream () ;
    filebuf* rdbuf () ;
    void open (const char* name, int mode=ios::out,
               int access=filebuf::openprot) ;
};
```

It is of a similar form to `ifstream`. The default value for `mode` is `ios::out`, indicating output. If a given file pointed to by `name` does not exist then, by default, it is created. The following program illustrates output to a file called `OUT.TXT` using an `ofstream` object:

```
// ofstr.cpp
```

---

2 The complement to `toupper()` is the function `int tolower(int ch)`, which translates character `ch` to lowercase if it is not already. The `strlwr()` function converts the uppercase letters in a string `s` to lowercase and is declared in `STRING.H`. The equivalent global functions for arguments of `class string` are `string to_upper(const string& s)` and `string to_lower(const string& s)`.

```

// illustrates the ofstream stream class for handling file
// output
#include <fstream.h> // C++ file I/O
#include <string.h> // strlen()

void main ()
{
    char file_name[] = "OUT.TXT" ;

    // define an ofstream object
    ofstream out_file (file_name) ;

    if (!out_file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    char string[] = "I am a string" ;
    int i (0) ;
    double d (1.25) ;

    // <<
    for (int j=0; j<strlen (string); j++)
        out_file << *(string+j) ;
    out_file << endl ;
    out_file << i << endl ;
    out_file << d << endl ;

    char* an_string (new char[20]) ;
    an_string = "I am another string" ;

    // ostream::put()
    while (*an_string)
        out_file.put (*an_string++) ;
    out_file.put ('\n') ;
}

```

After execution of the above program the contents of file OUT.TXT should read:

```

I am a string
0
1.25
I am another string

```

OFSTR.CPP illustrates two forms of writing to a disk file. The first uses the << insertion operator, while the second approach uses the ostream::put() member function, which **class** ofstream inherits from **class** ostream:

```

ostream& put (char ch) ;
ostream& put (signed char ch) ;
ostream& put (unsigned char ch) ;

```

The member function `put()` inserts a character `ch` into the output stream and returns a reference to an `ostream` object.

### 17.7.3 File Input and Output Using class `fstream`

The stream `class fstream` is derived from both `fstreambase` and `iostream`, and correspondingly caters for both file input and output:

```
class fstream : public fstreambase,
                public iostream // FSTREAM.H
{
public:
    // constructors
    fstream () ;
    fstream (const char* name, int mode,
              int access=filebuf::openprot) ;
    fstream (int fd) ;
    fstream (int fd, char* buffer, int buffer_length) ;
    ~fstream () ;
    filebuf* rdbuf () ;
    void open (const char* name, int mode,
               int access=filebuf::openprot) ;
};
```

Note that there is no default value for `mode`.

The following program illustrates simple file input and output using an `fstream` object:

```
// fstr.cpp
// illustrates the fstream stream class
// for handling file input and output
#include <fstream.h> // C++ file I/O

void main ()
{
    char file_name[] = "IN_OUT.TXT" ;

    // define an ofstream object
    fstream io_file (file_name, ios::in | ios::out) ;

    if (!io_file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    // read from file and echo to screen
    char ch ;

    while (io_file)
    {
        io_file.get (ch) ; // get a character
        cout << ch ;
    }
}
```

```
// write to file
char* string (new char[46]) ;
string = "I am probably a different string than the one read" ;
while (*string)
    io_file.put (*string++) ;
io_file.put ('\n') ;
}
```

The program uses an object, `io_file`, of **class** `fstream` to first read the contents of file `IN_OUT.TXT` and then, using the same object, append an additional string to `IN_OUT.TXT`.

#### 17.7.4 Stream Opening mode and access

In the previous subsection we saw the use of the `ios::in` and `ios::out` stream operation and opening modes for the second argument of the `fstream` constructor:

```
fstream io_file (file_name, ios::in | ios::out) ;
```

The stream opening modes determine how a file is to be opened and are generally specified as an enumerated `int` declared in **class** `ios` in the `IOSTREAM.H` header file:

```
enum open_mode
{
    in      = 0x01, // open for reading (default for ifstream)
    out     = 0x02, // open for writing (default for ofstream)
    ate     = 0x04, // seek to end-of-file
    app     = 0x08, // append to end-of-file
    trunc   = 0x10, // truncate file to zero if exists
    nocreate = 0x20, // open() fails if file does not exist
    noreplace = 0x40, // open() fails if file exists
    binary   = 0x80 // binary mode
};
```

The flags `in` and `out` indicate that a file is capable of input (open for reading) and output (open for writing) respectively. We saw in the above **class** declarations that `ios::in` and `ios::out` are the default opening modes for the three-argument constructors and `open()` member functions of `ifstream` and `ofstream`, respectively. The definition of object `io_file` of **class** `fstream` in `FSTR.CPP` specified both `ios::in` and `ios::out`, since `fstream` is capable of both input and output and has no default mode value.

The `ios::ate` flag results in a seek to the end-of-file, while `ios::app` appends output to any existing file contents. The `ios::trunc` deletes the existing contents of a file (if any), thus truncating the file length to zero.

The `ios::nocreate` and `ios::noreplace` flags cause the `open()` member function to fail if a file does not exist and to fail if a file does exist, respectively.

Finally, the `ios::binary` flag ensures that a file is opened in binary mode rather than text mode, text mode being the default. For instance, when a file is opened in text mode the carriage return/line feed sequence is converted to the new line '`\n`' character and vice versa for file output. Such conversions are not performed when in binary mode.

The `access` attribute specifies the access permission of a file. For Microsoft MS-DOS and Windows, Table 17.3 lists the five available attributes.

**Table 17.3** MS-DOS and Windows access permission values.

<i>access attribute</i>	<i>Permission</i>
0	Read and write file (default)
1	Read only file
2	Hidden file
4	System file
8	Archive bit set

### 17.7.5 Stream Status Flags

The base stream **class** `ios` declares an enumerated type called `io_state` with appropriate members that indicate the state of a stream:

```
enum io_state
{
    goodbit = 0x00, // no bit set- no errors
    eofbit = 0x01, // end-of-file
    failbit = 0x02, // last operation failed
    badbit = 0x04, // invalid operation attempted
    hardfail = 0x80 // unrecoverable error
};
```

When handling files it is important to know if any error(s) have occurred and if so to know specifically which error(s) so that appropriate action can be taken. For instance, the `istream::get()` member function fails if it encounters an end-of-file before a character is extracted and then proceeds to set the `ios::failbit` flag to inform the stream of failure.

Class `ios` supports a number of member functions to help diagnose the stream status flags and stream errors:

```
int rdstate () ; // IOSTREAM.H
int eof () ;
int fail () ;
int bad () ;
int good () ;
void clear (int state=0) ;
```

`rdstate()` returns the current state of a stream. The member function `eof()` returns non-zero if at the end of a file, `fail()` returns non-zero if an operation failed and `bad()` returns non-zero if an error occurred. The `good()` member function returns non-zero if no stream status flags have been set (i.e. no stream errors have occurred) and `clear()` sets the state of a stream to a given value.

The following program illustrates the `ios` stream status member functions:

```
// ios_stat.cpp
// illustrates the ios stream status flags
// and associated member functions
#include <fstream.h> // C++ file I/O
#include <iomanip.h> // setw(), ...
```

```
void main ()
```

```

{
// file stream object
ifstream file ;

// get file to open
char file_name[50] ;
cout << "enter a filename: " ;
cin >> file_name ;

// open
file.open (file_name, ios::in | ios::nocreate) ;

// test
if (!file)
    cerr << "file \" " << file_name
        << "\" not opened " << endl ;
else
    cout << "file \" " << file_name
        << "\" opened " << endl ;

cout << setw (15) << "file.rdstate():"
    << setw (2) << file.rdstate () << endl ;
cout << setw (15) << "file.good() :"
    << setw (2) << file.good () << endl ;
cout << setw (15) << "file.eof() :"
    << setw (2) << file.eof () << endl ;
cout << setw (15) << "file.fail() :"
    << setw (2) << file.fail () << endl ;
cout << setw (15) << "file.bad() :"
    << setw (2) << file.bad () << endl ;

// close
file.close () ;
}

```

with some user interaction for a non-existent file:

```

enter a filename: xxx.xxx
file "xxx.xxx" not opened
file.rdstate(): 4
file.good() : 0
file.eof() : 0
file.fail() : 4
file.bad() : 4

```

Incidentally, the `ios::eof()` member function is useful for detecting when the end of a file has been reached. The following program illustrates `eof()` via modification of IFSTR.CPP:

```

// eof.cpp
// illustrates the ios::eof() member function
// for detecting the end of a file

```

```
#include <fstream.h> // C++ file I/O

void main ()
{
    char file_name[] = "EOF.CPP" ;

    // define an ifstream object
    ifstream in_file (file_name) ;

    if (!in_file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    char ch ;

    while (!in_file.eof ())
    {
        in_file.get (ch) ; // get a character
        cout << ch ;
    }
}
```

### 17.7.6 The `open()` and `close()` Member Functions

Each of the three classes discussed above, `ifstream`, `ofstream` and `fstream`, defines an `open()` member function. Incidentally, an `open()` member function is defined for **class** `fstreambase`, from which `ifstream`, `ofstream` and `fstream` are derived. A `close()` member function is also defined by `fstreambase`.

The `open()` member function for the three classes `ifstream`, `ofstream` and `fstream` has the following declaration:

```
void open (const char* name, int mode, int access) ;
```

The argument `name` is the name of the file to be opened, whereas `mode` and `access` are the opening mode and access attribute, respectively, discussed above. If `open()` fails the value of the object which invoked the member function is set to zero.

The declaration of `fstreambase::close()` is:

```
void close () ;
```

which closes a file associated with a given stream.

Since classes `ifstream`, `ofstream` and `fstream` allow file stream objects to be defined which are not attached to a file at the time of definition, clearly the `open()` and `close()` member functions are useful for associating or disassociating, respectively, a file stream object with or from a file after object definition:

```
fstream file ;
//...
file.open /*...*/
//...
file.close () ;
```

As an illustration of the *open()* and *close()* functions, consider the following program:

```
// op_cl.cpp
// illustrates the open() and close() stream functions
#include <fstream.h> // C++ file I/O

void main ()
{
    // file stream object
    ifstream file ;

    // get file to open
    char file_name[50] ;
    cout << "enter a filename you wish to open and display: " ;
    cin  >> file_name ;

    // open
    file.open (file_name, ios::in | ios::nocreate) ;

    // test
    if (!file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    // display
    char ch ;

    while (file)
    {
        file.get (ch) ; // get a character
        cout << ch ;
    }

    // close
    file.close () ;
}
```

What happens if an attempt is made to open a file that is already open and similarly close a file that has previously been closed? Try duplicating the call to *open()* in the above program:

```
//...
// open
file.open (file_name, ios::in | ios::nocreate) ;
file.open (file_name, ios::in | ios::nocreate) ;
```

If the above program is now recompiled and executed, then upon entering a valid filename the ‘sorry can’t open file’ message will be displayed. A valid stream object is made invalid by further calls to *open()*. The *close()* function is not as fussy, in that a closed file remains closed following repeated *close()* calls.

You may be thinking that you will always know whether a file is open or not, or that it is simply a matter of referring to the program code to determine whether a file is open. In a large and complex program with numerous function calls it is not always a straightforward matter to establish whether a file has been previously opened or whether an open file has been inadvertently closed.

The **filebuf** **class**, which is derived from **streambuf**, declares a member function called *is\_open()*, which returns a non-zero integer value if a file is open:

```
int is_open () ; // FSTREAM.H
```

To call *is\_open()* via an **ifstream** object we have to note that **class** **fstreambase** declares a **private** data member of **class** **filebuf**, and that **class** **ifstream**, which is derived from **fstreambase**, declares the *rdbuf()* member function, which returns a pointer to **filebuf**:

```
int filebuf::is_open () {/*...*/}
filebuf* ifstream::rdbuf () {/*...*/}
```

The use of *is\_open()* is demonstrated in the following program:

```
// op_or_cl.cpp
// illustrates the is_open() function for
// determining whether a file is open or closed
#include <fstream.h> // C++ file I/O

void main ()
{
    // file stream object
    ifstream file ;

    // get file to open
    char file_name[50] ;
    cout << "enter a filename you wish to open and display: " ;
    cin >> file_name ;

    // open
    file.open (file_name, ios::in | ios::nocreate) ;

    // test
    if (!file)
        cerr << "sorry-can't open file \""
            << file_name << "\" " << endl ;

    if (file.rdbuf()->is_open())
        cout << "file is open" << endl ;

    // close
    file.close () ;

    if (!file.rdbuf()->is_open())
        cout << "file is closed" << endl ;
```

```
}
```

with some user interaction:

```
enter a filename you wish to open and display: op_or_cl.cpp
file is open
file is closed
```

### 17.7.7 The `read()` and `write()` Member Functions

The `istream::read()` member function extracts n characters or bytes from an input stream and places them in an array pointed to by buffer:

```
istream& read (char* buffer, int n) ;
istream& read (signed char* buffer, int n) ;
istream& read (unsigned char* buffer, int n) ;
```

If an end-of-file is reached before the specified number of characters have been read then `read()` stops reading. In such cases the `istream::gcount()` function is useful for determining the number of characters extracted from the last operation:

```
int gcount () ;
```

The `ostream::write()` member function inserts n characters or bytes from the array pointed to by buffer and places them into an output stream:

```
ostream& write (const char* buffer, int n) ;
ostream& write (const signed char* buffer, int n) ;
ostream& write (const unsigned char* buffer, int n) ;
```

The following program illustrates `read()`, `write()` and `gcount()`:

```
// r&w.cpp
// illustrates the read(), write() and gcount() member
functions
#include <fstream.h> // C++ file I/O

void main ()
{
    const int BUF_SIZ = 1e03 ;
    char in_file_name[] = "R&W.CPP" ;
    char out_file_name[] = "R&W.TXT" ;
    char buffer[BUF_SIZ] ;

    // read & display
    ifstream in_file (in_file_name) ;

    if (!in_file)
        cerr << "sorry-can't open file \""
            << in_file_name << "\"" " " << endl ;
```

```

in_file.read (buffer, sizeof buffer) ;

cout << in_file.gcount () << endl ;
for (int i=0; i<in_file.gcount(); i++)
    cout << buffer[i] ;

// write
ofstream out_file (out_file_name) ;

if (!out_file)
    cerr << "sorry-can't open file \""
        << out_file_name << "\"" << endl ;

out_file.write (buffer, in_file.gcount ()) ;

// close
out_file.close () ;
in_file.close () ;
}

```

This program uses the *read()* member function to fill an array of characters, *buffer*, of fixed size (1000). Since an end-of-file could occur before 1000 characters are read the *gcount()* member function is used to determine the actual number of characters read. The contents of *buffer* are then written to an output file *out\_file*.

The entire contents of the input file will be read provided the length of the input file is less than or equal to the size of *buffer*. In general this cannot be guaranteed, and an alternative mechanism would have to be devised to read the contents of an arbitrary sized file. In the next subsection we shall revisit this problem.

### 17.7.8 Random File Access

The file access that we have seen so far has been purely sequential, in that we have either read or written a file from beginning to end. However, there are occasions on which it is required to access an arbitrary position of a file. Random file access is generally described in terms of a *file pointer* (not to be confused with C++ pointers). A file pointer can be viewed as a finger which points to a given position or the *current position* of a file between the beginning and the end of a file. The current position is the point at which file access next commences. Since file access can be generally categorised in terms of input and output, the current position can more specifically be referred to as a *current get position* and a *current put position*. The terms *current get* and *put* positions will help introduce the associated member functions of classes *istream* and *ostream* for performing random or non-sequential file access.

Let's first examine the two random file access functions, *seekg()* and *tellg()*, of **class istream**:

```

istream& seekg (streampos pos) ;
istream& seekg (streamoff offset, seek_dir dir) ;

streampos tellg () ;

```

The one-argument version of `seekg()` moves the current `get` position to an absolute position, `pos`, in the input stream. The argument `pos` is of type `streampos`, which is simply a **long** **typedef** defined in `IOSTREAM.H`:

```
typedef long streampos ;
```

The two-argument version of `seekg()` moves `offset` bytes relative to the current `get` position in the direction specified by `dir`. As with `streampos`, `streamoff` is similarly defined in `IOSTREAM.H` as:

```
typedef long streamoff ;
```

The type of `dir` is an enumerated type `seek_dir` and is declared a member of **class** `ios`:

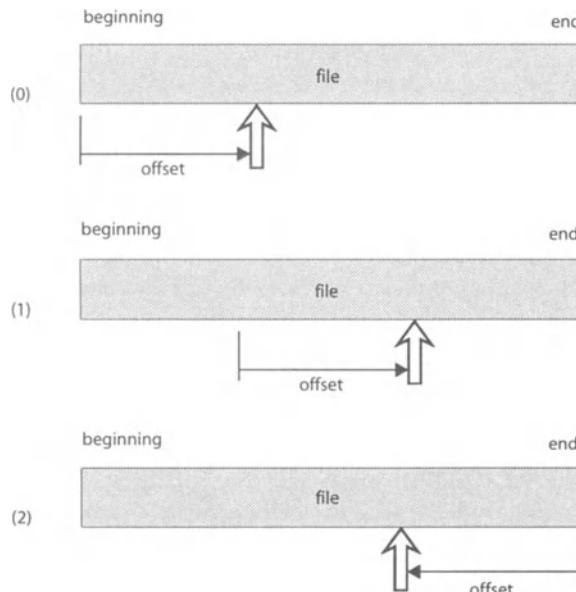
```
enum seek_dir
{
    beg, // beginning
    cur, // current position
    end // end
};
```

illustrating that there are three possibilities: an offset from the beginning (0); an offset from the current position (1); and an offset from the end (2): refer to Fig. 17.2.

The `istream::tellg()` member function returns the current position of the input stream and returns a value of `-1` if an error occurs.

The following program helps illustrate the `seekg()` and `tellg()` member functions:

```
// cgp.cpp
```



**Fig. 17.2** The three possibilities for defining a relative offset in a given direction from (0) the beginning; (1) the current position; and (2) the end of a stream.

```

// illustrates the seekg() and tellg() member functions
// of class istream for extracting the current get position
#include <fstream.h> // C++ file I/O

void main ()
{
    char file_name[] = "CGP.CPP" ;

    ifstream in_file (file_name) ;

    if (!in_file)
        cerr << "sorry-can't open file \""
            << file_name << "\"" << endl ;

    // determine file length
    streampos start = in_file.seekg (0, ios::beg).tellg () ;
    streampos end = in_file.seekg (0, ios::end).tellg () ;

    // get an offset
    streamoff offset (0) ;

    cout << "enter an offset(+) from the beginning of file \""
        << file_name << "\"" << endl
        << "which has a file length of "
        << (end-start) << " characters: " ;
    cin >> offset ;

    // go to offset
    in_file.seekg (offset, ios::beg) ;

    // display file to screen
    char ch ;

    while (in_file)
    {
        in_file.get (ch) ;
        cout << ch ;
    }
    // close
    in_file.close () ;
}

```

with some user interaction:

```

enter an offset(+) from the beginning of file "CGP.CPP"
which has a file length of 910 characters: 840
while (in_file)
{
    in_file.get (ch) ;
    cout << ch ;
}

```

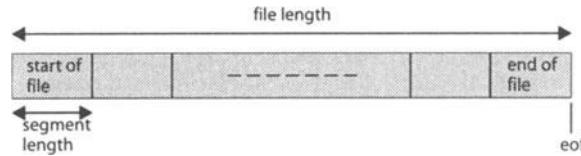


Fig. 17.3 Fragmentation of a file.

}

Similarly, **class ostream** has two member functions, *seekp()* and *tellp()*:

```
ostream& seekp (streampos pos) ;
ostream& seekp (streamoff offset, seek_dir dir) ;

streampos tellp () ;
```

The one-argument version of *seekp()* moves the current *put* position to an absolute position, *pos*, in the output stream. The two-argument version of *seekp()* moves *offset* bytes relative to the current *put* position in the direction specified by *dir*. The *tellp()* member function returns the current position of the output stream and returns a value of -1 if an error occurs.

When we examined program R&W.CPP it was mentioned that we would readdress the problem of reading an arbitrary sized file using a fixed sized buffer and the *istream::read()* member function. One solution is to view the entire input file as a series of adjacent segments, each of size equal to the buffer size; see Fig. 17.3. We can use the *istream::seekg()* member function to determine the length of the input file and therefore determine the number of segments to read, and then simply use the *read()* and *write()* member functions to transfer the input file contents to the output file:

```
// r&w1.cpp
// further illustrates the read(), write() and
// gcount() member functions
#include <fstream.h> // C++ file I/O
#include <math.h> // ceil()

void main ()
{
    ifstream in_file ("R&W1.CPP") ;
    ofstream out_file ("R&W1.TXT") ;

    // determine file length
    streampos start = in_file.seekg (0, ios::beg).tellg () ;
    streampos end   = in_file.seekg (0, ios::end).tellg () ;
    double file_length = end - start ;

    const int BUF_SIZ (100) ;
    char buffer[BUF_SIZ] ;

    // return to start of in-file
```

```

in_file.seekg (0, ios::beg) ;

// for each segment of the in-file
for (int i=0; i<ceil(file_length/BUF_SIZE); i++)
{
    in_file.read (buffer, BUF_SIZE) ; // fill buffer
    int siz = in_file.gcount () ;
    if (siz < BUF_SIZE) // if chars read < buffer size
        out_file.write (buffer, siz) ;
    else // a full buffer
        out_file.write (buffer, BUF_SIZE) ;
}
// close
out_file.close () ;
in_file.close () ;
}

```

Care is required when reading the last segment, since the actual number of characters read could be less than the buffer size, BUF\_SIZE, if an end-of-file occurred before BUF\_SIZE characters were read. Rather than flushing the buffer after each call to `write()` to ensure that no characters remain in the buffer from the last segment, we can alternatively use the `istream::gcount()` member function to return the exact number of characters extracted from the last `read()` operation. Note the use of the `ceil()`<sup>3</sup> C++ library function in R&W1.CPP, which rounds up a floating-point argument to the smallest integer not less than the argument value:

```
double ceil (double x) ; // MATH.H
```

### 17.7.9 Objects and Files

We saw earlier with reference to USR\_TYPE.CPP the use of the overloaded insertion and extraction operators for the user-defined `class Point` with the standard stream objects `cout` and `cin`. Let us now modify USR\_TYPE.CPP to read a number of `NumberedPoint` objects from the disk file POINTS.TXT and display them to the screen:

```

// ob_file.cpp
// illustrates the use of user-defined objects of
// class NumberedPoint and an input file stream
#include <fstream.h> // C++ file I/O
#include "pt_tmph.h" // template class Point
#include "npt_tmph.h" // template class NumberedPoint
#include "vec_tmph.h" // template class Vector

void main ()
{
    char file_name[] = "POINTS.TXT" ;
    int n_points (0) ;

```

---

<sup>3</sup> The complement to `ceil()` is the function `floor()`, which rounds down.

```

ifstream in_file ;

in_file.open (file_name) ;

in_file >> n_points ;

Vector<NumberedPoint<double> > points (n_points, 0.0) ;

for (int i=0; i<n_points; i++)
    in_file >> points[i] ;

cout << "points: " << endl << points ;

in_file.close () ;
}

```

with output:

```

points:
[(1.00, 2.00, 3.00, 0), (4.00, 5.00, 6.00, 1), (7.00, 8.00,
9.00, 2)]

```

The data file POINTS.TXT, from which the three NumberedPoint objects are read, is:

```

3
1.0 2.0 3.0 0
4.0 5.0 6.0 1
7.0 8.0 9.0 2

```

OB\_FILE.CPP defines an ifstream object called `in_file` from which the file POINTS.TXT is opened and the number of NumberedPoint objects, `n_points`, in the file read. The **template class** NumberedPoint is derived from **template class** Point; they are declared in NPT\_TMP.H and PT\_TMP.H respectively:

```

template <class T> // PT_TMP.H
class Point
{
protected:
    // protected data members
    T x, y, z ;
public:
    //...
};

//...
template <class T> // NPT_TMP.H
class NumberedPoint : public Point<T>
{
protected:
    // protected data member
    int number ;
public:

```

```

// constructors
NumberedPoint ()
    : Point<T> (), number (0) {}
NumberedPoint (T x_arg, T y_arg, T z_arg=0.0, int n=0)
    : Point<T> (x_arg, y_arg, z_arg), number (n) {}
//...
} ;

```

NumberedPoint associates an integer number with a Point object. The **template** classes Point, NumberedPoint and Vector are described in more detail in Chapters 13 and 15.

OB\_FILE.CPP then proceeds to read each NumberedPoint object in the POINTS.TXT file into the points object of **class** Vector< NumberedPoint<b>double> :

```

Vector< NumberedPoint<b>double> points (n_points, 0.0) ;

for (int i=0; i<n_points; i++)
    in_file >> points[i] ;

```

which illustrates that the use of the ifstream object in\_file, overloaded extraction operator and object points of a user-defined **class** is identical in syntax to the use of the cin predefined stream object with objects of integral data types.

The above program, OB\_FILE.CPP, read a series of NumberedPoint objects from a file called POINTS.TXT by first reading the number of objects stored in the file, then defining an object of **class** Vector to store the NumberedPoint objects to be read, and finally reading each NumberedPoint object using a **for**-loop. The operation of reading the number of objects stored in the file may seem unnecessary, and could be a difficult task to specify exactly if the number of objects is very large. Therefore, let us now modify OB\_FILE.CPP so that it determines the number of objects stored in a file rather than requiring a user to specify the number of objects explicitly. The file POINTS.TXT is replaced by POINTS1.TXT:

```

1.0 2.0 3.0 0
4.0 5.0 6.0 1
7.0 8.0 9.0 2

```

and OB\_FILE.CPP is replaced by OB\_FILE1.CPP:

```

// ob_file1.cpp
// further illustrates the use of user-defined objects of
// class NumberedPoint and an input file stream
#include <fstream.h> // C++ file I/O
#include "pt_tmp.h" // template class Point
#include "npt_tmp.h" // template class NumberedPoint
#include "ll.h" // template class LinkedList

void main ()
{
    char file_name[] = "POINTS1.TXT" ;

    ifstream in_file ;

```

```

    in_file.open (file_name) ;

LinkedList <NumberedPoint<double> > points ;

while (!in_file.eof ())
{
    NumberedPoint<double> pt ;
    in_file >> pt ;
    if (!in_file.eof ())
        points.Append (pt) ;
}

cout << points.NumberNodes () << " points: " << endl
    << points ;

in_file.close () ;
}

```

with output:

```

3 points:
(1.00, 2.00, 3.00, 0) (4.00, 5.00, 6.00, 1) ( 7.00, 8.00,
9.00, 2)

```

The NumberedPoint objects read from the file are stored in a linked list object, `points`, of the **template class** `LinkedList` (see Chapter 13) because a linked list is more responsive to the addition and deletion of elements or nodes than an array-based `Vector class` object. Since the number of objects to be read is no longer explicitly specified, a `while`-loop is used instead of a `for`-loop. The `while`-loop will continue to read objects until the end-of-file is reached. Each object read is first placed in the temporary object `pt` and then appended to the linked list object `points` via the `LinkedList<T>::Append()` member function. Note the necessity of having to perform a second test for end-of-file within the body of the `while`-loop so as not to append the last object read twice to the linked list.

You may be wondering why the `LinkedList<T>::Append()` member function was used in preference to an overloaded extraction operator of **template class** `LinkedList`:

```

//...
while (!in_file.eof ())
{
//...
    if (!in_file.eof ())
        in_file >> points[i] ;
}

```

The extraction operator is not overloaded for **template class** `LinkedList` because a node object can be added to a linked list at an arbitrary location, inserted at the head or appended to the tail of the linked list. Thus, it is simpler to use the `Insert()`, `Head()` or `Append()` member functions rather than imposing conditions on an overloaded extraction operator for **template class** `LinkedList`.

## 17.8 ReadFile, WriteFile and ReadAndWriteFile Classes

This section discusses the development of three classes called `ReadFile`, `WriteFile` and `ReadAndWriteFile` for the manipulation of input, output and input and output file streams respectively. The key reason for examining these classes is to demonstrate that user-defined classes can be derived from C++ library classes. Although it sounds obvious that classes can be derived from C++ library classes, it is a property that is frequently neglected. The C++ library classes have taken professional programmers years to develop and as a result are extremely powerful and robust; this is particularly applicable to the stream classes because they have a complex hierarchy.

The `ReadFile`, `WriteFile` and `ReadAndWriteFile` classes are derived from the C++ library file stream classes `ifstream`, `ofstream` and `fstream`, and therefore inherit the features of these classes, but in addition are derived from a **class** called `PathAndFile`, which encapsulates a filename. The filename will in fact incorporate the drive, directory path, filename and extension (assuming we are using MS-DOS) if each is specified by a user.

The `ReadFile` **class** is, in part, derived from `ifstream`, so before we examine `ReadFile` let's remind ourselves of the **class** declaration of `ifstream`:

```
class ifstream : public fstreambase, public istream // FSTREAM.H
{
public:
    // constructors
    ifstream () ;
    ifstream (const char* name, int mode=ios::in,
              int access=filebuf::openprot) ;
    ifstream (int fd) ;
    ifstream (int fd, char* buffer, int buffer_length) ;
    ~ifstream () ;
    filebuf* rdbuf () ;
    void open (const char* name, int mode=ios::in,
               int access=filebuf::openprot) ;
};
```

In the declaration of **class** `ReadFile` we shall focus our attention on replicating the four constructors and the `open()` and `close()` member functions of `ifstream`. The first three-argument constructor and `open()` member function of `ifstream` each specify a filename argument, `name`, of type `char*`. However, when defining a file stream object it is nice to be able to work with filenames which are of the C++ library **class** `string` (`CSTRING.H`) rather than `char*`. Therefore, `ReadFile` will declare a constructor and an `Open()` member function with a filename argument of **class** `string`. When the C++ library member functions `open()` and `close()` were discussed previously it was noted that `open()` can be called to open a file which is already open, and similarly for `close()`. Therefore, the `ReadFile` `Open()` and `Close()` member functions will detect whether a file is open or not and act accordingly.

In declaring the `ReadFile` **class** we could encapsulate either a **private** or **protected** data member of **class** `string`, which attaches a path and filename to a stream object. Alternatively, we can declare another **class** called `PathAndFile` which encapsulates a path and file object because the path and filename of a file are common features to all three classes, `ReadFile`, `WriteFile` and `ReadAndWriteFile`. It is then a simple matter for each stream **class** to derive from `PathAndFile`. The following header file `RW_FILE.H` includes the **class** declarations of `PathAndFile` and the user-defined stream classes:

```
// rw_file.h
// ReadFile, WriteFile and ReadAndWriteFile class
declarations

#ifndef _RW_FILE_H // prevent multiple includes
#define _RW_FILE_H

#include <fstream.h> // C++ file I/O
#include <cstring.h> // C++ class string
#include <ctype.h> // isalnum()

class PathAndFile
{
protected:
    string pathfilename ;
public:
    // constructors
    PathAndFile () ;
    PathAndFile (const string& s) ;
    // member functions
    void SetPathAndFileName (const string& s) ;
    string PathAndFileName () const ;
    string FileName () ;
    string FileExtension () ;
    string Path () ;
    char Drive () ;
}; // class PathAndFile

class ReadFile : public ifstream, public PathAndFile
{
public:
    // constructors
    ReadFile () ;
    ReadFile (const char* name, int mode=ios::in,
              int access=filebuf::openprot) ;
    ReadFile (int fd) ;
    ReadFile (int fd, char* buffer, int buffer_length) ;
    ReadFile (const string& s, int mode=ios::in,
              int access=filebuf::openprot) ;
    // member functions
    int Open (const string& s, int mode=ios::in,
              int access=filebuf::openprot) ;
    int Open (int mode=ios::in,
              int access=filebuf::openprot) ;
    void Close () ;
}; // class ReadFile

class WriteFile : public ofstream, public PathAndFile
{
public:
    // constructors
```

```

        WriteFile () ;
        WriteFile (const char* name, int mode=ios::out,
                   int access=filebuf::openprot) ;
        WriteFile (int fd) ;
        WriteFile (int fd, char* buffer, int buffer_length) ;
        WriteFile (const string& s, int mode=ios::out,
                   int access=filebuf::openprot) ;
    // member functions
    int Open (const string& s, int mode=ios::out,
               int access=filebuf::openprot) ;
    int Open (int mode=ios::out,
               int access=filebuf::openprot) ;
    void Close () ;
}; // class WriteFile

class ReadAndWriteFile : public fstream, public PathAndFile
{
public:
    // constructors
    ReadAndWriteFile () ;
    ReadAndWriteFile (const char* name, int mode,
                      int access=filebuf::openprot) ;
    ReadAndWriteFile (int fd) ;
    ReadAndWriteFile (int fd, char* buffer,
                      int buffer_length) ;
    ReadAndWriteFile (const string& s, int mode,
                      int access=filebuf::openprot) ;
    // member functions
    int Open (const string& s, int mode,
               int access=filebuf::openprot) ;
    int Open (int mode, int access=filebuf::openprot) ;
    void Close () ;
}; // class ReadAndWriteFile

#endif // _RW_FILE_H

```

PathAndFile declares one **protected** data member, two constructors and six member functions. The pathfilename data member is of **class** string and holds the path and filename of a given file. The no-argument constructor initialises pathfilename to a zero length string, whereas the one-argument constructor simply sets pathfilename to the constructor argument. Before discussing the member functions of PathAndFile, let's list the key elements of RW\_FILE.CPP, which is the corresponding implementation file of RW\_FILE.H:

```

// rw_file.cpp
// implementation file for classes
// ReadFile, WriteFile and ReadAndWriteFile
#include "rw_file.h"

// PathAndFile:

```

```
PathAndFile::PathAndFile ()
: pathfilename ()
{
}

PathAndFile::PathAndFile (const string& s)
: pathfilename (s)
{
}

// sets the path and filename
void PathAndFile::SetPathAndFileName (const string& s)
{
    pathfilename = s ;
}

// returns the path and filename
string PathAndFile::PathAndFileName () const
{
    return pathfilename ;
}

// extracts the filename from a path and filename string
string PathAndFile::FileName ()
{
    string filename ;
    for (int i=pathfilename.rfind('\\')+1;
          i<pathfilename.length(); i++)
        filename.append (pathfilename[i]) ;
    return filename ;
}

// extracts the filename extension from a path and filename
// string
string PathAndFile::FileExtension ()
{
    string file_ext ;
    for (int i=pathfilename.rfind('.')+1;
          i<pathfilename.length(); i++)
        file_ext.append (pathfilename[i]) ;
    return file_ext ;
}

// extracts the path from a path and filename string
string PathAndFile::Path ()
{
    string path ;
    if (pathfilename.rfind ('\\') != NPOS) // if path
        for (int i=0; i<pathfilename.rfind('\\'); i++)
            path.append (pathfilename[i]) ;
    return path ;
}
```

```
}
```

```
// extracts the drive from a path and filename string
char PathAndFile::Drive ()
{
    char drive ('\0') ;
    if (pathfilename.find (':') != NPOS) // if drive
        drive = pathfilename[0] ;
    return drive ;
}
```

```
// ReadFile:
```

```
ReadFile::ReadFile ()
: ifstream (), PathAndFile ()
{ }
```

```
ReadFile::ReadFile (const char* name, int mode, int access)
: ifstream (name, mode, access), PathAndFile (name)
{ }
```

```
ReadFile::ReadFile (int fd)
: ifstream (fd)
{ }
```

```
ReadFile::ReadFile (int fd, char* buffer, int buffer_length)
: ifstream (fd, buffer, buffer_length)
{ }
```

```
ReadFile::ReadFile (const string& s, int mode, int access)
: ifstream (s.c_str (), mode, access), PathAndFile (s)
{ }
```

```
// opens a specified file
// returns 1 if opened successfully and
// -1 if the file is already open
int ReadFile::Open (const string& s, int mode, int access)
{
    pathfilename = s ; // set path and filename
    if (this->rdbuf()->is_open()) // already open
        return -1 ;
    else // open
    {
        ifstream::open (pathfilename.c_str (), mode, access) ;
        return 1 ;
    }
```

```

}

// opens a file when filename has previously been defined
// returns 1 if opened successfully, 0 if a filename has
// not been set and -1 if the file is already open
int ReadFile::Open (int mode, int access)
{
    if (pathfilename.length () == 0)
        return 0 ;
    else if (this->rdbuf()->is_open())
        return -1 ;
    else // open
    {
        ifstream::open (pathfilename.c_str (), mode, access) ;
        return 1 ;
    }
}

// close an open file
void ReadFile::Close ()
{
    if (this->rdbuf()->is_open())
        ifstream::close () ;
}

// WriteFile:
//...
// ReadAndWriteFile:
//...

```

The `SetPathAndFileName()` member function allows a path and filename to be set to a specified string and, as we shall see later, is useful for defining an object of **class** `ReadFile` without simultaneously associating a filename with the object at the time of object definition. The `PathAndFileName()` access member function returns the `pathfilename` string data member. `FileName()` returns the filename (including extension) of `pathfilename`:

```

string PathAndFile::FileName ()
{
    string filename ;
    for (int i=pathfilename.rfind('\\')+1;
          i<pathfilename.length(); i++)
        filename.append (pathfilename[i]) ;
    return filename ;
}

```

The temporary object `filename` will hold the filename. The **for**-loop uses the `string::rfind()` member function to find the *last* occurrence of the backslash character '`\`' in the `pathfilename` string and initialise the control variable `i` of the **for**-loop to this position. The **for**-loop then proceeds to append (using `string::append()`) each character of `pathfilename` to the `filename` string until the end of the string is reached (using

`string::length()`). The declarations of member functions `string::rfind()`, `string::append()` and `string::length()` are:

```
size_t rfind (const string& s) ;
size_t rfind (const string& s, size_t pos) ;

string& append (const string& s) ;
string& append (const string& s, size_t n) ;
string& append (const char* cp, size_t n) ;

unsigned length () const ;
```

where `size_t` is an `unsigned` `typedef` and is declared in `STDIO.H`.

`rfind()` finds the last occurrence of string `s` in the target string that is not beyond position `pos` in the target string if specified. If the string `s` is not found then `rfind()` returns `NPOS`:

```
const size_t NPOS = size_t (-1) ; // CSTRING.H
```

`NPOS` is equivalent to the largest value of `typedef` `size_t` (i.e. 65 535 and not -1!). `append()` appends the string `s` to the target string or the first `n` characters of `s` to the target string if specified. The string `s` can be specified either as a `string` object or as a pointer to an array of characters. The number of characters in a `string` object is returned by `length()`. It is worth mentioning that indexing the characters of a `string` object starts at zero and that a `string` object can hold NULL characters.

The member functions `FileExtension()` and `Path()` are similar in principle to `FileName()` except that `FileExtension()` uses `rfind()` to locate the last occurrence of the '.' character instead of '\', and `Path()` confirms that a path is specified before performing a search using `rfind()`. The `Drive()` member function is slightly different in that it returns a single character rather than a `string` object to indicate the drive and uses the `string::find()` member function to confirm that a drive is specified before performing the assignment:

```
char PathAndFile::Drive ()
{
    char drive ('\0') ;
    if (pathfilename.find (':') != NPOS) // if drive
        drive = pathfilename[0] ;
    return drive ;
}
```

The `string::find()` member function has the following declaration:

```
size_t find (const string& s) ; // CSTRING.H
```

and finds the *first* occurrence of string `s` in the target string. Note that there are additional overloaded versions of `string::find()`.

With `PathAndFile` declared we can now declare `class ReadFile` using multiple inheritance:

```
class ReadFile : public ifstream, public PathAndFile
{
```

```
//...
};
```

The first four constructors of `ReadFile` replicate those of `ifstream`, whereas the fifth constructor allows a `stream` object to be defined using a `string` filename argument:

```
ReadFile::ReadFile (const string& s, int mode, int access)
    : ifstream (s.c_str (), mode, access), PathAndFile (s)
{
}
```

Note the use of the `string::c_str()` member function:

```
const char* c_str () const; // CSTRING.H
```

which returns a pointer to a NULL-terminated character array which contains the same characters as the `string` object operated on by the member function.

Similarly, the three-argument `ReadFile::Open()` member function allows a `stream` object to be opened in terms of a `string` filename:

```
int ReadFile::Open (const string& s, int mode, int access)
{
    if (this->rdbuf()->is_open ()) // already open
        return -1 ;
    else // open
    {
        pathfilename = s; // set path and filename
        ifstream::open (pathfilename.c_str (), mode, access) ;
        return 1 ;
    }
}
```

The `ReadFile::Open()` member function could equally have been named `open()` to overload `ifstream::open()`, but was named `Open()` purely on the grounds of notational convention. `Open()` first confirms, using `filebuf::is_open()`, that the function is not called to open an already open file. If the associated file is already open a value of `-1` is returned. If the file is being opened for the first time the `pathfilename` data member is set, the file is opened by simply calling `ifstream::open()` and a value of `1` is returned to indicate a successful file open call.

Alternatively, the two-argument `ReadFile::Open()` member function allows a file to be opened for which the path and filename have been previously set:

```
int ReadFile::Open (int mode, int access)
{
    if (pathfilename.length () == 0)
        return 0 ;
    else if (this->rdbuf()->is_open ())
        return -1 ;
    else // open
    {
        ifstream::open (pathfilename.c_str (), mode, access) ;
```

```
    return 1 ;  
}  
}
```

The member function first confirms that a path and filename have indeed been defined, and if not a value of 0 is returned. Next, the function tests if the specified file is already open and if so a value of -1 is returned. Finally, if a path and filename have been set and the file is not already open, the file is opened and a value of 1 is returned.

The `ReadFile::Close()` member function will close a file only if it is open:

```
void ReadFile::Close ()
{
    if (this->rdbuf()->is_open())
        ifstream::close () ;
}
```

The `WriteFile` and `ReadAndWriteFile` **class** declarations are of a similar form to that of `ReadFile`. Use of the `ReadFile` **class** presented above is illustrated in the following program:

```

// rw_tst.cpp
// tests the ReadFile, WriteFile and ReadAndWriteFile classes
#include <iostream.h> // C++ I/O
#include <iomanip.h> // setw()
#include <cstring.h> // C++ class string
#include "rw_file.h" // classes ReadFile, ...

void main ()
{
    string filename ("c:\\bc5\\include\\fstream.h") ;

    //ReadFile in_file ;
    //in_file.Open (filename) ;

    ReadFile in_file (filename) ;

    cout << setw (21) << "path and filename: "
        << in_file.PathAndFileName () << endl ;
    cout << setw (21) << "filename: "
        << in_file.FileName () << endl ;
    cout << setw (21) << "filename extension: "
        << in_file.FileExtension () << endl ;
    cout << setw (21) << "path: " << in_file.Path () << endl ;
    cout << setw (21) << "drive: " << in_file.Drive () << endl ;

    in_file.close () ;

    // open or not open?
    ReadFile in_file1 ; // define object-no file associated
    if (in_file1.Open () == 0) // try and open
        cout << "filename not set" << endl ;
}

```

---

```

in_file1.SetPathAndFileName("obj_in.dat") ; // set filename
if (in_file1.Open ()) // now open
    cout << "file opened" << endl ;
if (in_file1.Open () == -1) // try to open an open file
    cout << "file already open" << endl ;
in_file1.Close () ;
}

```

with output:

```

path and filename : c:\bc5\include\fstream.h
filename : fstream.h
filename extension: h
        path: c:\bc5\include
        drive: c
filename not set
file opened
file already open

```

## 17.9 Reading a Collection of Objects from a Disk File

Continuing our examination of reading from and writing to a disk, let's now address the problem of reading a disk file which contains a collection of user-defined geometrical objects.

Frequently when performing geometric and solid modelling it is desirable to be able to read objects from an *object data file* rather than defining them explicitly in either a header or implementation file. The type of objects that we shall consider are those discussed in the Shapes program of Chapter 15; see SHAPE\_V.H and SHAPES.CPP and Fig. 15.12 for an overview of the **class** hierarchy of the Shapes program. Excluding the abstract base **class** Shape and base **class** Plane, the actual shapes or objects considered are a circle, sphere, polygon, quadrilateral, triangle and tetrahedron.

Shown below is the disk file OBJ\_IN.DAT, which represents a typical object data file:

```

CIRCLE
{
[0,-1,0,1,(1,0,0),1,1] // normal=(0,-1,0),c=(1,0,0),
r=1 & no.=1
}
// CIRCLE { [0,-1,0,1,(1,0,0),1,5] }
SPHERE
{
[(5,5,5),1,2]
}
Triangle // 2 Triangle objects
{
[(1,1,0),(4,2,0), (3, 4,2),     3]
[(1,1,1),(2,2,1),(3,3,3),4]
}
QUADRILATERAL

```

```
{
[(0,0,0),(1,0,0),(1,1,0),(0,1,0),5] /*...*/,6]
}
//TRIANGLE { /*.../,7} POLYGON { /*.../,8}
POLYGON
{
// a Polygon object
[5,(1,0,2),(3,0,1),(4,0,3),(3,0,6),(0.5,0,5),7] // 5 vertices
}
TRIANGLE { /*.../,8} POLYGON { /*.../,9}
TETRAHEDRA { [(1,1,0),(3,2,0),(4,5,2),(2,4,0),10] }
```

The start of an object is indicated by a *keyword* or *object name descriptor*, which for the sake of convenience, as we shall see later, is in fact equivalent to the object's **class** name. The keyword can be specified entirely in lowercase, entirely in uppercase or a mixture of both lowercase and uppercase characters. The data of an object (such as vertices or number) are enclosed by two square brackets( [ and ] ), which are similarly enclosed by the braces ( { and } ), which are similar to the C++ syntax. Encompassing an individual object between square brackets allows multiple objects to be defined between a single pair of braces as illustrated above for the first of the two triangle objects.

The object description language is fully free-format and allows C++-style comments ( // . . . ) to be placed anywhere within a data file. The above object data file illustrates that any number of objects can be placed in any order, at any position within a line or with multiple objects on a single line. Similarly, comments can be positioned at the start of a line, after a keyword, within the braces or directly following the square brackets of an object's data. Also, the free-format nature of the object description language applies within the square brackets as illustrated above for the first of the two triangle objects.

Note that the C-style comments ( /\* . . . \*/ ) are not part of the object description language, but are used above to illustrate the variety of ways in which objects can be defined.

A first solution in the implementation of a **class** which reads a collection of objects from a file would be simply to extend the **ReadFile** **class** presented in the previous section to read Shape objects from a file. However, such an implementation would inappropriately specialise **ReadFile** because there is no relationship between reading a file and a list of arbitrary objects. Therefore, the following implementation will introduce a new **class** called **World** which encapsulates a list of *world* objects. In addition, the choice of a **World** **class** will become more evident when we discuss the development of a raytracing program in Chapter 20. Thus, the **PathAndFile**, **ReadFile**, **WriteFile** and **ReadAndWriteFile** **class** declarations and definitions remain the same as those in **RW\_FILE.H** and **RW\_FILE.CPP**.

The following lists the declaration of **class** **World**:

```
// world.h
// a World class

#ifndef _WORLD_H // prevent multiple includes
#define _WORLD_H

#include <typeinfo.h> // RTTI
#include <cstring.h> // C++ class string

#include "rw_file.h" // classes ReadFile, ...
```

```

#include "bool.h"           // enum Boolean
#include "ll.h"             // template class LinkedList

#include "shape_v.h"         // class Shape
#include "plane_v.h"         // class Plane
#include "poly_v.h"          // class Polygon
#include "tri_v.h"           // class Triangle
#include "quad_v.h"          // class Quadrilateral
#include "circ_v.h"           // class Circle
#include "sphere_v.h"         // class Sphere
#include "tet_v.h"            // class Tetrahedra

class World
{
private:
    // nested class
    class SortKeyword
    {
public:
    // data members
    string kw ;
    size_t pos ;
    // constructors
    SortKeyword ()
        : kw (), pos (0) {}
    SortKeyword (const string& s,
                 const size_t& p)
        : kw (s), pos (p) {}
    // overloaded relational operators
    Boolean operator < (const SortKeyword& rhs)
    { return pos < rhs.pos ? B_TRUE :
      B_FALSE ; }
    Boolean operator == (const SortKeyword& rhs)
    { return (pos == rhs.pos &&
              kw == rhs.kw) ?
      B_TRUE : B_FALSE ; }
    Boolean operator != (const SortKeyword& rhs)
    { return (pos != rhs.pos &&
              kw != rhs.kw) ?
      B_TRUE : B_FALSE ; }
}; // nested class SortKeyword

// data members
ReadFile obj_infile ; // obj in file
WriteFile obj_outfile ; // obj out file
LinkedList<Shape*> l_obj ; // list of world objects
// member function
LinkedList<SortKeyword>
    KeywordPositions (const string& str,
                      LinkedList<string>& l_kw) ;
Boolean CheckBracesNoNesting

```

```

                                (const streampos& fp) ;
Shape*           ReadParticularObject
                                (const string& obj_type) ;
LinkedList<Shape*> ReadListOfObjects
                                (const string& obj_type,
                                 const streampos& fp_arg) ;

public:
// constructor
World (const string& o_infile,
       const string& o_outfile) ;
// member functions
LinkedList<string> ReadKeywords
                                (const string& kw_fname="OBJ_KW.DAT") ;
void ReadObjects (const string& kw_fname="OBJ_KW.DAT") ;
void WriteObjects () ;
LinkedList<Shape*> ObjectList () const { return l_obj ; }
}; // class World

#endif // _WORLD_H

```

It is worth noting that in addition to the header files included in RW\_FILE.H the above file WORLD.H also includes the C++ RTTI header file TYPEINFO.H, CTYPE.H and the Shape and LinkedList associated header files<sup>4</sup>.

The two-argument constructor of **class** World sets the path and filenames of the in and out object data files without opening the files:

```

World::World (const string& o_infile, const string& o_outfile)
{
    obj_infile.SetPathAndFileName (o_infile) ;
    obj_outfile.SetPathAndFileName (o_outfile) ;
}

```

The in and out object data files will be opened and closed, as required, by World member functions.

Eight member functions are defined for **class** World. *KeywordPositions()* is a **private** member function and returns a linked list of keywords (with their respective positions via the nested **class** SortKeyword) found in a given string. *CheckBracesNoNesting()* returns logical-true if an open brace is followed by a close brace from a specified file pointer position, else logical-false is returned. Thus, as the name suggests, *CheckBracesNoNesting()* does not allow the nesting of braces. *ReadKeywords()* returns a linked list of keywords listed in a specified file. The default filename is OBJ\_KW.DAT, which in the present case, is:

```

Triangle
QUADRILATERAL
CIRCLE
SPHERE

```

---

<sup>4</sup> If you are programming in a Windows environment and your compiler's version of the C++ **string** library header file, CSTRING.H, includes the WINDOWS.H header file you may experience a name clash between the Polygon **class** declared in POLY\_V.H and the Windows *Polygon()* API function declared in WINDOWS.H.

---

TETRAHEDRA

POLYGON

Listing the type of objects considered in a separate data file removes them from a .H or .CPP file and therefore makes it easier for a user to alter the object type list or specify an alternative list. The listed object types can be specified entirely in lowercase, entirely in uppercase or a mixture of both lowercase and uppercase characters. *ReadParticularObject()* returns a pointer to a given object via the Shape abstract base **class**. *ReadListOfObjects()* returns a linked list of objects between the next pair of open and close braces from a specified file pointer position. *ReadObjects()* is the most important member function to be defined for **class** World and associates a linked list of all the objects read from a file to the World::l\_obj data member. *WriteObjects()* writes a linked list of objects to a file and *ObjectList()* returns the l\_obj data member. World also declares a **private** nested **class** called SortKeyword which will be discussed later in connection with *ReadObjects()*.

So, let's now examine the eight member functions of World discussed above in more detail. The member functions quickly become very involved, and I am not suggesting that you attempt to understand every detail of their operation. As always, try to focus on the overall design of the functions and their integration with the World and Shape classes. Further, the following functions will be far from ideal, so try to look for areas which could be improved or simplified. Here we go:

```
// world.cpp
// implementation file for class World
#include "world.h"

// private:

// returns a linked list of the
// keywords (with their positions) in a string
// accounts for '//' comments in the string
LinkedList<World::SortKeyword>
World::KeywordPositions (const string& str,
                        LinkedList<string>& l_kw)
{
    LinkedList<SortKeyword> l_skw ;

    // if string is worth examining
    if (str.length() > 0)
    {
        // find position of '//' in string-if present
        Boolean comment_found (B_FALSE) ;
        size_t comment_pos (0) ;
        for (int i=0;i<str.length()-1; i++)
            if (str[i] == '/' && str[i+1] == '/')
            {
                comment_found = B_TRUE ;
                comment_pos = i ;
                break ;
            }
        // find positions of keywords in string-if any
```

```

for (int j=0; j<l_kw.NumberNodes(); j++)
{
// find first occurrence of keyword
size_t kw_pos = str.find (l_kw[j]) ;
if (kw_pos != NPOS &&
comment_found && kw_pos < comment_pos)
{
SortKeyword skw (l_kw[j], kw_pos) ;
l_skw.Append (skw) ;
}
if (kw_pos != NPOS && !comment_found)
{
SortKeyword skw (l_kw[j], kw_pos) ;
l_skw.Append (skw) ;
}
// find other occurrences-if any
while (kw_pos != NPOS)
{
// commence search from end of last kw. found
kw_pos =
str.find (l_kw[j],
kw_pos+l_kw[j].length()-1) ;
if (kw_pos != NPOS &&
comment_found && kw_pos < comment_pos)
{
SortKeyword skw (l_kw[j], kw_pos) ;
l_skw.Append (skw) ;
}
if (kw_pos != NPOS && !comment_found)
{
SortKeyword skw (l_kw[j], kw_pos) ;
l_skw.Append (skw) ;
}
}
}
}
}

// sort and return
l_skw.Sort () ;
return l_skw ;
} // World::KeywordPositions()

// returns logical-true if the sequence {...} is found
// from the specified file pointer position (fp),
// else logical-false is returned
Boolean World::CheckBracesNoNesting (const streampos& fp)
{
Boolean bool (B_FALSE) ;

// first get current file position
streampos orig_fp = obj_infile.tellg () ;

```

```
// go to specified file position
obj_infile.seekg (fp, ios::beg) ;

char ch ;
int number (0) ;
char bra[2] = { 'x', 'x' } ; // ensure != '{' or '}'

while (!obj_infile.eof ()) // till eof
{
    obj_infile.get (ch) ;
    // test for open and close braces
    if (ch == '{' || ch == '}')
    {
        number++ ;
        bra[number] = ch ;
    }
    if (number == 2) // found first two braces
        break ;
}
if (bra[1] == '{' && bra[2] == '}') ? bool = B_TRUE
                                         : bool = B_FALSE ;

// return file position to original value
obj_infile.seekg (orig_fp, ios::beg) ;

return bool ;
} // World::CheckBracesNoNesting()

// returns a pointer to a particular object type
Shape* World::ReadParticularObject (const string& obj_type)
{
Shape* obj (NULL) ;

if (obj_type == "CIRCLE")
{
    Circle* c (new Circle) ;
    obj_infile >> *c ;
    obj = c ;
}
if (obj_type == "SPHERE")
{
    Sphere* s (new Sphere) ;
    obj_infile >> *s ;
    obj = s ;
}
if (obj_type == "POLYGON")
{
    Polygon* p (new Polygon) ;
    obj_infile >> *p ;
    obj = p ;
}
```

```

if (obj_type == "TRIANGLE")
{
    Triangle* t (new Triangle) ;
    obj_infile >> *t ;
    obj = t ;
}
if (obj_type == "QUADRILATERAL")
{
    Quadrilateral* q (new Quadrilateral) ;
    obj_infile >> *q ;
    obj = q ;
}
if (obj_type == "TETRAHEDRA")
{
    Tetrahedra* t (new Tetrahedra) ;
    obj_infile >> *t ;
    obj = t ;
}
return obj ;
} // World::ReadParticularObject()

// returns a linked list of objects read
// of a particular object type between braces { and }
LinkedList<Shape*> World::ReadListOfObjects
    (const string& obj_type,
     const streampos& fp_arg)
{
    // first get current file position
    streampos orig_fp = obj_infile.tellg () ;

    // go to specified file position
    obj_infile.seekg (fp_arg, ios::beg) ;

    // pointer to abstract base class Shape
    Shape* obj (NULL) ;
    LinkedList<Shape*> l_obj ; // linked list of objects

    streampos fp = fp_arg ; // file pointer to start of line
    string str, r_str ; // original and reduced strings

    // read file
    while (r_str.find ('}') == NPOS)
    {
        r_str.remove (0) ; // flush r_str

        // a line at a time
        ::getline (obj_infile, str, '\n') ;

        // get current file pointer
        streampos cp = obj_infile.tellg () ;
}

```

```
// convert string to uppercase-if not already
str = to_upper (str) ;

// check for comments and develop reduced string
for (int i=0; i<str.length(); i++)
{
    // /**' comment: quit reading characters
    if (i<str.length()-1 &&
        str[i] == '/' && str[i+1] == '/')
        break; // leave for-loop
    else // read '[' and ']' characters
    {
        if (str[i] == '[' || str[i] == ']' ||
            str[i] == ')')
        {
            r_str.append (str[i]) ;
            if (str[i] == ')') // upto first ')'
                break; // leave for-loop
        }
    }
}

// line is worth examining further
if (r_str.length () > 0)
{
    // find first '[' in line-if any
    size_t r_pos = r_str.find ('[') ;
    if (r_pos != NPOS) // '[' found
    {
        // find first '[' in original line
        size_t o_pos = str.find ('[') ;
        if (o_pos != NPOS)
        {
            streampos fp_pos = fp + o_pos ;
            obj_infile.seekg (fp_pos, ios::beg) ;

            // read object and append to list
            obj = ReadParticularObject
                (obj_type) ;
            l_obj.Append (obj) ;
        }
    }

    // find other '-'if any
    int number_of (0) ;
    for (int i=0; i<r_str.length(); i++)
        if (r_str[i] == '-')
            number_of++ ;

    // if more than one '[' per line
    if (number_of >= 2)
```

```

{
    LinkedList<int> pos ;

    // find position/s in line of '['
    for (int i=0; i<str.length(); i++)
        if (str[i] == '[')
            pos.Append (i) ;

    // already read first object above,
    // so start from second
    for (int j=1; j<pos.NumberNodes(); j++)
    {
        streampos fp_pos = fp + pos[j] ;
        obj_infile.seekg (fp_pos, ios::beg) ;

        // read object and append to list
        obj = ReadParticularObject
            (obj_type) ;
        l_obj.Append (obj) ;
    }
}
// if r_str.length()>0
obj_infile.seekg (cp, ios::beg) ;// re-pos'n file pointer
fp = obj_infile.tellg () ;           // update file pointer
} // while ()

// return file position to original value
obj_infile.seekg (orig_fp, ios::beg) ;

return l_obj ;
} // World::ReadListOfObjects()

// public:

// set object in and out data files but do not
// open files!
World::World (const string& o_infile,
              const string& o_outfile)
{
    obj_infile.SetPathAndFileName (o_infile) ;
    obj_outfile.SetPathAndFileName (o_outfile) ;
}

// returns a linked list of the object keywords in the
// specified file
LinkedList<string> World::ReadKeywords (const string& kw_fname)
{
    // open keyword file
    ifstream kw_file (kw_fname.c_str ()) ;
    if (!kw_file)
    {

```

```
    cerr << "sorry-can't open file \" " << kw_fname
        << "\n" << endl ;
    exit (EXIT_FAILURE) ;
}

// place keywords in linked list
LinkedList<string> keywords ;
while (!kw_file.eof ())
{
    string str ;
    kw_file >> str ;           // read string
    str = to_upper (str) ;     // convert to uppercase
    if (!kw_file.eof ())
        keywords.Append (str) ;
}
// close and return
kw_file.close () ;
return keywords ;
} // World::ReadKeywords()

// reads a series of objects from a file
void World::ReadObjects (const string& kw_fname)
{
    streampos fp (0) ;          // file pointer to start of line
    size_t    line_count (0) ;   // line count
    string    str ;             // line string

    // open in object data file
    obj_infile.Open () ;

    // get keywords
    LinkedList<string> keywords = ReadKeywords (kw_fname) ;

    // read file
    while (!obj_infile.eof ())
    {
        // a line at a time
        ::getline (obj_infile, str, '\n') ;

        // convert string to uppercase-if not already
        str = to_upper (str) ;

        // get keywords (with their respective positions) in line
        LinkedList<SortKeyword> l_skw =
            KeywordPositions (str, keywords) ;

        line_count++ ;

        // line is worth examining further
        if (l_skw.NumberNodes() > 0)
        {
```

```

// for each keyword in line string
for (int i=0; i<l_skw.NumberNodes(); i++)
{
    // adjust 'exact' fp position
    // and record current fp
    streampos fp_pos = fp + l_skw[i].pos ;
    streampos cp      = obj_infile.tellg () ;
    // if braces o.k.
    if (CheckBracesNoNesting (fp_pos))
    {
        obj_infile.seekg (cp, ios::beg) ;
        // read objects and append to list
        LinkedList<Shape*> l_pobj =
            ReadListOfObjects (l_skw[i].kw,
                               fp_pos) ;
        for (int j=0;
              j<l_pobj.NumberNodes(); j++)
            l_obj.Append (l_pobj[j]) ;
    }
    else // simply quit!
    {
        cout << "invalid braces on line: "
            << line_count << endl ;
        exit (EXIT_FAILURE) ;
    }
}
fp = obj_infile.tellg () ; // update file pointer
} // while (!eof())

// close in object data file
obj_infile.close () ;
} // World::ReadObjects()

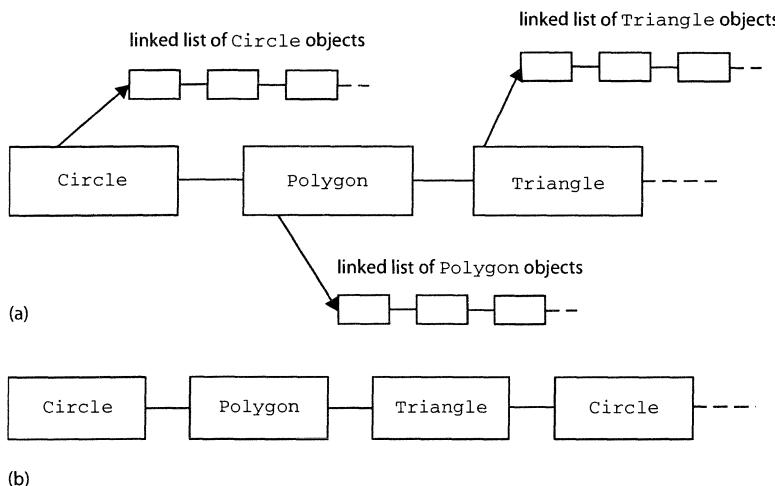
// simply write all objects to the object out file
void World::WriteObjects ()
{
    // open out object data file
    obj_outfile.Open () ;

    if (l_obj.NumberNodes() == 0) // objects not yet read
        this->ReadObjects () ;

    // write objects to out-file
    for (int i=0; i<l_obj.NumberNodes(); i++)
    {
        obj_outfile << typeid (*l_obj[i]).name () << endl ;
        obj_outfile << *l_obj[i] << endl ;
    }

    // close out object data file
}

```



**Fig. 17.4** Two alternative approaches to storing a linked list of objects. (a) The nodes of a *central* linked list point to linked lists of objects of a given type or **class**. (b) Each node of a *single* linked list is a unique object.

```
obj_outfile.close () ;  
}
```

Let's begin by taking a look at the *ReadObjects()* member function. The first question to ask when designing a function to read a collection of objects is just how the set of objects is to be stored. The storage mechanism has to be dynamic due to the unknown nature, at the time of compilation, of the object data file. The two most popular dynamic storage mechanisms are clearly vectors and linked lists. A linked list approach is chosen owing to its increased flexibility for the addition and deletion of nodes or elements when compared to a vectorial approach. Two approaches were considered for storing a linked list of objects; see Fig. 17.4. The approach in Fig. 17.4(a) is made possible by the fact that **LinkedList<T>** is a **template class** which accommodates composed types. The approach shown in Fig. 17.4(b) is the adopted approach because it allows a simpler implementation than that of Fig. 17.4(a) and because the RTTI features of C++ allow the type or **class** of an object in a single linked list to be determined at run-time. Since we are dealing with objects of a **class** in the **Shape** inheritance hierarchy, the type or **class** of the linked list of objects is a pointer to **Shape**.

*ReadObjects()* first opens the in-object data file and then calls *ReadKeywords()* to obtain a linked list of the keywords which will be searched for in the object data file. The object data file is then read line by line till the end-of-file is reached using the global *getline()* function. Following a call to *getline()* the file pointer position is incremented to the start of the next line. The version of *getline()* used is particularly useful because it places the string of characters read into an object of the C++ library **class** **string**, and will continue to read characters until the new-line delimiter is encountered rather than specifying a fixed number of characters to be read. For the present application it is more advantageous to read the object data file line by line rather than a character at a time, for reasons which will become evident. Working with a **string** object rather than a C++ array of characters greatly simplifies the implementation. The line **string** read from the object data file is then converted to uppercase (if not already) in order to simplify future string comparisons. For the line **string** just read the *KeywordPositions()* member function is then called to obtain a linked list of keywords found in the **string** (if any) with their associated positions in the **string**. An association between

a keyword and its position in a string is achieved by declaring an additional **private** nested **class** called **World**: : **SortKeyword**, which encapsulates both string, kw, and position, pos, data members. Three overloaded relational operators are defined for **World**: : **SortKeyword** to assist in the sorting of a linked list of **SortKeyword** objects. The < operator compares two **SortKeyword** objects in terms of position only, which guarantees that the keywords in a line string containing multiple keywords will be read in the order in which they appear from left to right.

For each keyword found in a line string the exact file position of the keyword is first determined and then passed to the member function *CheckBracesNoNesting()* to confirm that the non-nested syntax of the open and close braces is correct. If the braces pass the test then the *ReadListOfObjects()* member function is called to return a linked list of all the objects read from the object data file between the braces. The linked list of objects read between braces is then appended to the total linked list of objects previously read. *ReadObjects()* concludes by closing the previously opened in-object data file.

The **World**: : *ReadKeywords()* member function returns a linked list of the keywords read from a specified file, which is OBJ\_KW.DAT by default. The keywords read from the keyword data file are converted to uppercase if not already.

**World**: : *KeywordPositions()* is passed a string and a linked list of keywords, for which it is to search the string and return a linked list of **SortKeyword** objects. Comments may exist in the passed string, and if the comment characters (//) are found then the string position of the first forward slash character is recorded. *KeywordPositions()* then proceeds to search for each keyword in the string using the **string**: : *find()* member function, being careful to ensure that commented-out keywords are not appended to the linked list of keywords returned. Note the second form of the **string**: : *find()* member function used in the **while**-loop of *KeywordPositions()*:

```
//...
while (kw_pos != NPOS)
{
    // commence search from end of last kw. found
    kw_pos = str.find (l_kw[j], kw_pos+l_kw[j].length()-1) ;
    //...
}
```

which commences searching for a keyword from the end of a previously found keyword.

The **World**: : *CheckBracesNoNesting()* member function returns logical-true if, from a specified file position, first an open brace is encountered followed by a close brace with other characters possibly appearing before the open brace and between the two braces. Any other permutation of braces will result in *CheckBracesNoNesting()* returning a value of logical-false.

**World**: : *ReadListOfObjects()* returns a linked list of objects of a particular type from a specified file pointer position. *ReadListOfObjects()* checks for the comment characters within the line string and works with two **string** objects: **str**, which is the original line string read, and **r\_str**, which is a *reduced* string containing only the characters [, ] and ]. The reduced string object is then examined to determine the file pointer positions of the start of an object's data indicated by the character [. If an object is found then its data are read by calling the *ReadParticularObject()* member function and appended to a linked list of total objects read of a particular type or **class**. Locating objects by the character [ allows multiple objects of a particular type or **class** to be defined between a single pair of { and } braces.

Finally (in connection with the reading of a list of objects) the *ReadParticularObject()* member function creates a new object of a specified type or **class**, reads the object from the current file pointer position, assigns the object to a pointer to Shape and returns the Shape pointer.

The *World::WriteObjects()* member function first opens the out-object data file (OBJ\_OUT.DAT by default), confirms that the *World::l\_obj* linked list of objects is not empty (an example of anticipation) and then proceeds to write each object encapsulated in the *World::l\_obj* linked list data member to a disk file. The linked list of objects is written to a file with the aid of the overloaded insertion operator and the *Print()* overridden member function. Note the use of the *typeid()* operator and the *Type\_info::name()* member function to extract, at run-time, the **class** name of each object via the pointer to abstract base **class** *Shape*. Finally, *WriteObjects()* closes the previously opened out-object data file.

The access member function *World::ObjectList()* simply returns the *l\_obj* data member to allow access to the linked list of *World* objects.

The following program reads a collection of objects from a specified in-object data file and then writes the objects read to an out-object data file:

```
// read_obj.cpp
// tests the World class by reading a set of objects from a
// disk file
#include "world.h" // class World

void main ()
{
    World world ("obj_in.dat", "obj_out.dat") ;

    // read world objects
    world.ReadObjects () ;

    // write world objects
    world.WriteObjects () ;
}
```

Following execution of READ\_OBJ.EXE the OBJ\_OUT.DAT disk file will be of the form:

```
Circle
number: 1, centre: (1, 0, 0), radius: 1
Sphere
number: 2, centre: (5, 5, 5), radius: 1
Triangle
number: 3
vertices: (1, 1, 0), (4, 2, 0), (3, 4, 2)
edges: [(1, 1, 0), (4, 2, 0)], [(4, 2, 0), (3, 4, 2)],
      [(3, 4, 2), (1, 1, 0)]
//...
Tetrahedra
number: 10
faces:
number: 0
vertices: (1, 1, 0), (3, 2, 0), (4, 5, 2)
//...
```

When we discussed the Shape inheritance hierarchy in Chapter 15 the insertion operator was overloaded for classes derived from Shape to assist in the stream output of objects. For the present chapter the approach taken is to declare a pure **virtual** function for abstract **class** Shape called *Print()* and then to override this function for each **class** in the Shape inheritance hierarchy:

```
// shape_v.h
// header file for Shape class
//...

class Shape
{
public:
//...
virtual void Print (ostream& s=cout) = 0 ;
friend ostream& operator << (ostream& s, Shape& shp)
{ shp.Print (s) ; return s ; }
}; // class Shape

// poly_v.h
// header file for Polygon class
//...
class Polygon : public Shape, public Plane
{
public:
//...
void Print (ostream& s) ;
}; // class Polygon
//...
```

Note the use of the predefined **ostream** object **cout** as a default argument for **s** in the member function declaration of **Shape::Print()**. The overloaded insertion operator for abstract base **class** Shape redirects any object of a **class** derived from Shape to a specified output stream by means of **Shape::operator<<()** invoking the correct overridden **virtual Print()** function. **READ\_OBJ.CPP** called the **World::WriteObjects()** member function and illustrated an application of the overloaded **<<** operator for writing a linked list of objects (pointers to Shape) to a disk file pointed to by the **WriteFile** object **out\_file**:

```
void World::WriteObjects()
{
//...
for (int i=0; i<l_obj.NumberNodes(); i++)
{
//...
out_file << *l_obj[i] ;
//...
}
//...
}
```

## 17.10 String Streams

The header file STRSTREAM.H declares classes which support in-memory string formatting. The **class** strstreambuf is derived from streambuf and provides operations for manipulating in-memory strings. The **class** strstreambase is derived from ios and specialises ios for string streams. Class istrstream is derived from strstreambase and istream and provides support for input stream operations on a strstreambuf object, whereas **class** ostrstream is derived from strstreambase and ostream and provides output stream operations on a strstreambuf object. The **class** strstream is derived from strstreambase and iostream and provides both input and output stream operations.

The **class** declarations of istrstream, ostrstream and strstream are of the following form:

```

class istrstream : public strstreambase,
                public istream // STRSTREA.H
{
public:
    istrstream (char* s) ;
    istrstream (signed char* s) ;
    istrstream (unsigned char* s) ;
    istrstream (char* s, int n) ;
    istrstream (signed char* s, int n) ;
    istrstream (unsigned char* s, int n) ;
    ~istrstream () ;
};

class ostrstream : public strstreambase,
                public ostream // STRSTREA.H
{
public:
    ostrstream () ;
    ostrstream (char* s, int n, int mode=ios::out) ;
    ostrstream (signed char* s, int n, int mode=ios::out) ;
    ostrstream (unsigned char* s, int n, int mode=ios::out) ;
    ~ostrstream () ;
// member functions
char* str() ;
int pcount() ;
};

class strstream : public strstreambase,
                  public iostream // STRSTREA.H
{
public:
    strstream () ;
    strstream (char* s, int n, int mode) ;
    strstream (signed char* s, int n, int mode) ;
    strstream (unsigned char* s, int n, int mode) ;
    ~strstream () ;
// member function

```

---

```
    char* str() ;
};
```

The first three overloaded constructors of `ostrstream` initialise an object with a NULL terminated string. The latter three forms of the `ostrstream` overloaded constructors initialise an object with a string array of `n` elements. Both classes `ostrstream` and `strstream` have default constructors. The three non-zero-argument constructors of `ostrstream` and `strstream` initialise an object with a string array of `n` elements for a given mode; the mode is `ios::out` by default for objects of **class** `ostrstream`.

Let's take a look at a program which uses the `istrstream` **class**:

```
// istrstr.cpp
// illustrates the C++ input string stream class istrstream
#include <strstrea.h> // strstream
#include <string.h> // strlen()

const int BUF_SIZ = 80 ;

void main ()
{
    cout << "enter an int, double and string: " ;

    int i (0) ;
    double d (0.0) ;
    char s[BUF_SIZ/2], buffer[BUF_SIZ] ;

    cin.getline (buffer, BUF_SIZ) ;

    istrstream iss (buffer, strlen (buffer)) ;

    iss >> i >> d >> s ;

    cout << "i: " << i << ", d: " << d << ", string: "
        << s << endl ;
}
```

with some user interaction:

```
enter an int, a double and a string: 9 12.2 c++
i: 9, d: 12.2, string: c++
```

## 17.11 Console Streams

The header file CONSTREA.H includes the **class** declarations of `conbuf` and `constream` and various console manipulators for console-mode applications. CONSTREA.H provides similar **class** and manipulator based functionality to that of the header file CONIO.H.

The `conbuf` **class** is derived from `streambuf` for console output and supports numerous operations; refer to your compiler's documentation for a more detailed discussion of `conbuf`'s operations. The **class** `constream` is derived from `ostream` and is of the form:

```

class constream : public ostream // CONSTREA.H
{
private:
    conbuf buf ;
public:
    constream() ;
    // member functions
    void     clrscr () ;
    conbuf* rdbuf() ;
    void     textmode (int mode) ;
    void     window (int l, int t, int r, int b) ;
};

```

Class `constream` encapsulates a `conbuf` data member and a single default constructor. The member function `clrscr()` clears the console screen, `rdbuf()` returns a pointer to the `conbuf` data member of the object operated on by the function, `textmode()` sets the screen mode to text mode and `window()` defines the rectangular dimensions of an active window.

The CONSTREA.H header file also declares several manipulators specifically for formatting console streams. Before we examine the manipulators, let's examine the **class** declaration of a two-argument version of the `omanip<T>` parametrised **template class** declaration declared in IOMANIP.H which was previously discussed in connection with user-defined parametrised manipulators:

```

template <class T, class S> // CONSTREA.H
class two_omanip
{
private:
    ostream& (*pf)(ostream&, T, S) ;
    T data1 ;
    S data2 ;
public:
    two_omanip<T, S>
        (ostream& (*pf_arg)(ostream&, T, S),
         T data1_arg, S data2_arg)
        : pf(pf_arg), data1(data1_arg), data2(data2_arg)
        { }
    friend ostream& operator << (ostream& s,
                                    two_omanip<T, S>& m)
    { return(*m.pf)(s, m.data, m.data2) ; }
};

```

`two_omanip` is useful when defining two-argument parametrised manipulators such as the `setxy(int,int)` manipulator discussed below.

Let's first examine the non-parametrised console stream manipulators declared in CONSTREA.H:

```

ostream& clreol      (ostream& s) ; // CONSTREA.H
ostream& delline     (ostream& s) ;
ostream& highvideo   (ostream& s) ;
ostream& insline     (ostream& s) ;
ostream& lowvideo    (ostream& s) ;

```

```
ostream& normvideo (ostream& s) ;
```

`clreol` clears to the end of a line in the text window. `delline` deletes a line in a text window, whereas `insline` inserts a line in the text window. `highvideo` selects the high-intensity characters, whereas `normvideo` selects the normal intensity characters.

The standard C++ parametrised console stream manipulators are:

```
omanip<int>      setattr     (int a) ; // CONSTREA.H
omanip<int>      setbk       (int cc) ;
omanip<int>      setclr      (int c) ;
omanip<int>      setcrsrtype (int t) ;
two_omanip<int,int> setxy      (int x, int y) ;
```

`setattr()` sets various screen attributes. `setbk()` sets the background character colour, whereas `setclr()` sets the foreground colour. Each allowable colour has an associated symbolic constant and numeric value (e.g. `BLACK` (0) and `WHITE` (15)). `setcrsrtype()` sets the cursor type and `setxy()` positions the cursor at the specified position.

The following program illustrates `constream`:

```
// constr.cpp
// illustrates the C++ console stream class constream
#include <constrea.h> // constream

void main ()
{
    // console objects
    constream win1, win2 ;

    // define windows and clear
    win1.window (2, 2, 39, 15) ;
    win2.window (41, 4, 78, 17) ;
    win1.clrscr () ;
    win2.clrscr () ;

    // O/P to win1
    win1 << setbk (BLUE)
        << setxy (7, 25)
        << "I'm window #1" ;

    // O/P to win2
    win2 << setbk (CYAN)
        << setxy (7, 25)
        << "I'm window #2" ;
}
```

## 17.12 Redirection

The standard input and output streams of `cin` and `cout` are, respectively, the keyboard and the console. C++ supports a feature called *redirection* in which the standard input and output

streams can be redirected to alternative devices. For MS-DOS the syntax for redirecting command-line input and output involves redirection characters and is of the general form:

- The operator > redirects output from the console to an alternative device such as a disk file or printer.
- The operator < redirects input from the keyboard to an alternative device such as a disk file or printer.
- The >> operator redirects output from the console to a disk file and appends the output to existing data (if any) in the file.

To illustrate redirection, consider first the program:

```
// redirect.cpp
// illustrates redirection
#include <iostream.h> // C++ I/O

void main ()
{
    char ch ;

    while (cin.get (ch)) // while I/P, get and echo
        cout << ch ;
}
```

and some user interaction when REDIRECT.EXE is executed at the command line of the C : root directory:

```
C:\>redirect
hello there!
hello there!
^Z
```

When carriage return is pressed REDIRECT.CPP echoes the user string to the display screen. The program is terminated either by entering Ctrl-Z or by pressing the F6 function key, since both of these are equivalent to end-of-file.

### 17.12.1 Redirecting Output from the Display Screen to a Disk File

To redirect output from the console to a disk file the > operator is used on the command line. The following user interaction redirects a user-entered string 'hi there!' from the display screen to a disk file called HI\_THERE.TXT:

```
C:\>redirect > hi_there.txt
hi there!
^Z
```

If the file HI\_THERE.TXT does not already exist MS-DOS creates the file. If the file does exist any existing data in the file is replaced by the output from the REDIRECT command.

The use of the program REDIRECT.CPP for redirecting output is identical to, say, redirecting a directory listing from the display screen to a disk file called DIR\_LIST.TXT using the dir command:

---

```
C:\>dir > dir_list.txt
```

Alternatively, if you require that any existing data in a given file is not deleted then the `>>` redirection character ensures that any new output is appended to existing data:

```
C:\>redirect >> hi_there.txt
hi there again!
^Z
```

The content of file HI\_THERE.TXT is now:

```
hi there!
hi there again!
```

### 17.12.2 Redirecting Input from the Keyboard to a Disk File

Similarly, to redirect input from the keyboard to a file or printer the `<` redirection character can be used. For example, the following redirects the input for REDIRECT.CPP from the file HI\_THERE.TXT and displays the contents to the display screen:

```
C:\>redirect < hi_there.txt
hi there!
hi there again!
```

### 17.12.3 Redirecting both Input and Output

It is possible to redirect both input and output with a single command using both the `<` and `>` redirection characters. For example, the following redirects FILE1.TXT to FILE2.TXT:

```
C:\>redirect < file1.txt > file2.txt
```

## 17.13 Command Line Arguments

There are occasions when it is required to pass information to a program from the command line. For instance, the MS-DOS COPY command allows one or more source files to be copied to a destination file which is resident in the same directory or another directory or disk:

```
C:\>copy src.txt dst.txt
```

The filenames SRC.TXT and DST.TXT are known as *command line arguments* and are passed to the COPY command before execution commences.

One of the most surprising features of `main()` is that it is a function (just like any other function) which has two optional arguments. All of the `main()` functions that we have seen to date have been of the form:

```
void main ()
{
// ...
}
```

but in fact *main()* has the following more general form:

```
void main (int argc, char* argv[])
{
//...
}
```

Although any argument names can be used, it has become convention that the names *argc* and *argv* are used as the first and second arguments to *main()*, respectively. The first argument, *argc*, is the argument count and is used to inform the program of the number of command line arguments. Note that *argc* will be at least equal to one, since the program name itself qualifies as a command line argument. The second argument, *argv*, is the argument vector, and is an array of pointers to **char** that point to the command line arguments. Since the argument vector is of type **char\*** the command line arguments are passed to *main()* as NULL-terminated strings, and if an alternative argument type is required then a conversion is necessary.

Let's take a look at a program which is passed a string command line argument:

```
// hello.cpp
// illustrates command line arguments
// and the passing of strings
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()

void main (int argc, char* argv[])
{
// check command-line arguments
if (argc != 2)
{
cout << "format: hello name" << endl ;
exit (EXIT_FAILURE) ;
}

// display message
cout << "hi there " << argv[1] << "!" << endl ;
}
```

With some user interaction (assuming that HELLO.EXE is executed at the command line of C: root directory):

```
C:\>hello zak
hi there zak!
```

The program name and string argument are separated by a space. The function body of *main()* first checks that the number of command line arguments is equal to two (program name plus the string argument). If the user enters the incorrect number of command line arguments a message is displayed, informing the user of the required format of command line arguments for successful program execution and the program terminates. If the correct number of command line arguments are entered the program simply displays a message to the user via the standard output stream.

The string following the program name is accessed as *argv[1]*, not *argv[0]*. Some compilers and operating systems flush the first element of *argv* when program execution

commences. Other compilers and operating systems place the disk and directory path and program name in `argv[0]` (e.g. `argv[0] = "C:\HELLO.EXE"` for the above example using the Borland C++ (version 5.0) compiler for Windows).

The following program passes numeric rather than character data to `main()` to evaluate the square root of a number:

```
// sq_root.cpp
// illustrates command-line arguments
// and the passing of numerics
#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit(), atof()
#include <math.h> // sqrt()

void main (int argc, char* argv[])
{
    // check command-line arguments
    if (argc != 2)
    {
        cout << "format: sq_root number" << endl ;
        exit (EXIT_FAILURE) ;
    }

    // display square root
    cout << sqrt (atof (argv[1])) << endl ;
}
```

with some user interaction:

```
C:/>sq_root 2
1.41421
```

The structure of SQ\_ROOT.CPP is identical to that of HELLO.CPP, except that the C++ library function `atof()` is used to convert a string to a floating-point **double** number. The declaration of `atof()` is:

```
double atof (const char* string) ; // STDLIB.H or MATH.H
```

The C++ library includes a variety of other functions which perform conversions between a string and a numeric, and vice versa. For a complete list refer to your compiler's documentation, but the most popular are `atoi()`, which converts a string to an **int**; `atol()`, which converts a string to a **long**; `strtod()`, which converts a string to a **double**; `strtol()`, which converts a string to a **long**; `strtoul()`, which converts a string to an **unsigned long**; `itoa()`, which converts an **int** to a string; and `ltoa()`, which converts a **long** to a string.

Let's conclude our discussion of command line arguments by developing our own program which mimics the MS-DOS COPY command. It must be mentioned that the COPY command is a comprehensive command and has the general syntax:

```
COPY [/A|/B]src[/A|/B][+src[/A|/B][+...]]
      [dst[/A|/B]][/V][/Y|/-Y]
```

where `src` is the source file(s) to be copied, `dst` is the new destination file, `/A` indicates an ASCII text file, `/B` indicates a binary file, `/V` verifies that the destination file is written correctly, `/Y` and `/-Y` suppress and verify that an existing destination file is to be overwritten and `+` is used for the appending of source files. However, we shall develop a simplified version of `COPY` which performs a copy of a source file to a destination file:

```
// cpp_cpy.cpp
// illustrates command-line arguments
#include <fstream.h> // C++ file I/O
#include <stdlib.h> // exit()
#include <string.h> // strcmp()

void main (int argc, char* argv[])
{
    .
    // check command-line arguments
    if (argc != 3)
    {
        cerr << "format: cpp_cpy src_file dst_file" << endl ;
        exit (EXIT_FAILURE) ;
    }

    // prevent self copy
    if (strcmp (argv[1], argv[2]) == 0)
    {
        cerr << "source file must be different from
                destination file" << endl ;
        exit (EXIT_FAILURE) ;
    }

    // open source and destination files
    ifstream src_file ;
    ofstream dst_file ;

    src_file.open (argv[1], ios::binary) ;
    if (!src_file)
    {
        cerr << "sorry-can't open file \""
            << argv[1]
            << "\" " << endl ;
        exit (EXIT_FAILURE) ;
    }
    dst_file.open (argv[2], ios::binary) ;
    if (!dst_file)
    {
        cerr << "sorry-can't open file \""
            << argv[2]
            << "\" " << endl ;
        exit (EXIT_FAILURE) ;
    }

    // copy file
    while (src_file)
        dst_file.put (char(src_file.get ()) ) ;
}
```

---

```
// close files
src_file.close () ;
dst_file.close () ;
}
```

with some user interaction:

```
C:\>cpp_cpy src.txt dst.txt
```

CPP\_CPY.CPP first performs a check to confirm that the number of command line arguments (i.e. three) is correct. Then the C++ library function *strcmp()* is used to confirm that the names of the source and destination files are different so as to prevent the source file being copied onto itself. If these two checks are successfully passed the source and destination files are opened with further checks for each file object to confirm that the files are opened for reading and writing respectively. Next, the contents of the source file are copied to the destination file using the *istream::get()* and *ostream::put()* member functions. Finally, both the opened source and destination files are closed.

## 17.14 The C Approach to Streams

Let's now examine the C language approach to streams. Although C's mechanism for handling streams is not object-oriented, it is nevertheless extremely powerful and cannot be ignored if you are considering migrating from programming in C to C++. This section begins by introducing the STDIO.H header file and the C predefined streams. Then the functions *printf()* and *scanf()* are discussed in detail, since they are the most frequently used functions for performing output and input operations, respectively, on the standard input and output streams. The section concludes by examining file handling in C.

### 17.14.1 The STDIO.H Header File

The STDIO.H (standard input and output) header file is the C equivalent to the C++ IOS-TREAM.H header file. STDIO.H incorporates the necessary structure and function declarations for the high-level manipulation of streams.

### 17.14.2 Predefined Streams

In a similar manner to the C++ predefined standard stream objects *cin*, *cout*, *cerr* and *clog*, the C language defines five predefined open streams in the STDIO.H header file:

```
extern FILE _streams[ ];
/*...*/
#define stdin  (&_streams[0])
#define stdout (&_streams[1])
#define stderr (&_streams[2])
#define stdaux (&_streams[3])
#define stdprn (&_streams[4])
```

The FILE structure is discussed later with reference to disk files.

The stream `stdin` is dedicated to the standard input device, which is generally the keyboard. The stream `stdout` refers to the standard output device, which is usually the display screen. The stream `stderr` is the standard error output device, which again is usually the display screen. The streams `stdaux` and `stdprn` are, respectively, the standard auxiliary and printing devices.

### 17.14.3 Output to the Display Screen Using the `printf()` Function

The C approach to directing a stream to the `stdout` stream (i.e. the display screen) is to use the `printf()` function, which has the following declaration:

```
int printf (const char* fmt_string[, argument,
...]) ; /* STDIO.H */
```

The `fmt_string` argument, as the name suggests, is called the *format string* and contains text and *format specifiers* or *conversion characters*. The `printf()` function returns the number of bytes or characters output if successful, else the EOF (=−1) identifier (STDIO.H) is returned if unsuccessful. Before discussing the available conversion characters let's examine a program which displays a string:

```
/*
printf0.c
illustrates the printf() function
*/
#include <stdio.h> /* C I/O */
void main ()
{
    printf ("hello, world\n");
}
```

with output:

```
hello, world
```

`printf()` outputs the string of characters between the double quotation characters (" . . . "). The character '\n' is the new line escape sequence and advances the current position to the first position of the next line. The C++ equivalent to '\n' is the `endl` manipulator. Escape sequences were discussed in Chapter 4; refer to Table 4.1 for a list of the available escape sequences.

Let's now consider an example program which uses a conversion character:

```
/*
printf1.c
further illustrates the printf() function
*/
#include <stdio.h> /* C I/O */
```

---

```
#include <math.h> /* sqrt() */

void main ()
{
    double number = 2.0 ;

    printf ("the square root of %f is: %f\n",
            number, sqrt (number)) ;
}
```

with output:

the square root of 2.000000 is: 1.414214

A conversion character sequence starts with the percentage sign (%) and is followed by the format code. The conversion character in the above program is f, which performs a conversion to a decimal floating-point number. The *printf()* function now has three arguments, each separated by the comma operator. Note that for each conversion character there is a corresponding function argument and that the order of the conversion characters matches the order of the arguments.

Conversion characters have the following general form:

%[flags][width][precision][F|N|h|l|L]type

where:

[flags]	Flag characters (optional)
[width]	Width specifiers (optional)
[precision]	Precision specifiers (optional)
[F N h l L]	Input size modifiers (optional)
type	Conversion type character (required)

Table 17.4 lists the available conversion characters and the available flag characters are listed in Table 17.5. For instance, %‐10d left-justifies a signed decimal integer with a total character field width of 10. Note that + takes precedence over a blank space if both are specified.

The available width specifiers are shown in Table 17.6. The width specifiers set the minimum field width for output and can be specified directly through the use of a decimal or indirectly through the use of an asterisk. If a field width is specified directly using the n specifier and the output width is less than n characters the output is padded with blank spaces, whereas if a field width is specified directly using the On specifier and the width is less than n characters the output is padded with zeros. If an asterisk is used the next argument in the call to *printf()* specifies the field width. Note that if the result of a conversion is wider than the specified field width the field is expanded accordingly to accommodate the conversion result.

The precision specifiers are shown in Table 17.7. A precision specifier sets the number of characters or digits for output and is preceded by the dot operator (.) so as to distinguish it from a width specifier. Similarly to the width specifier, a precision specifier can be specified directly through the use of a decimal or indirectly through the use of an asterisk. If the output value exceeds n characters the output is truncated or rounded depending on the type of conversion character. For conversion characters d, i, o, u, x and X, if the input value has fewer than n digits the value is left-padded with zeros, whereas if the input value has more than n digits the value is not truncated. For e, E and f the last input value digit is rounded when the

**Table 17.4** Conversion characters for *printf()*.

Conversion character	Input	Output
Character		
c	Character	Single character
s	String pointer	Character string
%	None	The % sign
Numeric		
d	Integer	Signed decimal integer
e	Floating-point	Scientific notation: e[+/-]number
E	Floating-point	Scientific notation: E[+/-]number
f	Floating-point	Signed decimal floating-point
g	Floating-point	e or f form, whichever is shorter
G	Floating-point	E or f form, whichever is shorter
i	Integer	Signed decimal integer
o	Integer	Unsigned octal integer
u	Integer	Unsigned decimal integer
x	Integer	Unsigned hexadecimal integer (a, b, c, d, e, f)
X	Integer	Unsigned hexadecimal integer (A, B, C, D, E, F)
Pointer		
n	Pointer to <code>int</code> pointer	Number of characters output up to that point
p		A pointer

**Table 17.5** Flag characters for *printf()*.

Flag	Meaning
-	Left-justify, right filled with blanks
+	Signed conversion
blank space	If non-negative, output starts with a blank instead of +
#	Use one of the alternate forms below
Alternate forms	
c, d, i, s, u	No effect
0	0 prepended to non-zero argument
x, X	0x, 0X prepended to argument
e, E, f	Decimal point shown even if no digit follows
g, G	Same as e, E except that trailing zeros are not removed

**Table 17.6** Width specifiers for *printf()*.

Width specifier	Meaning
n	A minimum of n characters are output (blank padding)
On	A minimum of n characters are output (zero padding)
*	Argument list contains width specifier

n digits are exceeded, while n specifies the maximum number of significant digits to output for g and G. In the case of the s conversion character, n specifies the maximum number of characters to output.

Table 17.8 illustrates the input size modifiers. The input size modifiers specify how the *printf()* function is to interpret the next input argument.

Consider the following program, which illustrates a few of the character conversion specifications mentioned above:

**Table 17.7** Precision specifiers for *printf()*.

Precision specifier	Meaning
None	Default precision used: c = no effect d, i, o, u, x, X = 1 e, E, f = 6 g, G = all significant digits s = print to NULL character
.0	e, E, f: no decimal point is output d, u, o, u, x: precision is set to defaults
.n	n characters or decimal places are output
.*	Argument list contains precision specifier

**Table 17.8** Input size modifiers for *printf()*.

Input size modifier	Meaning
F	Argument interpreted as a far pointer
N	Argument interpreted as a near pointer (memory model-dependent)
h	Argument interpreted as a <b>short int</b> for d, i, o, u, x, X
l	Argument interpreted as a <b>long int</b> for d, i, o, u, x, X and as a <b>double</b> for e, E, f, g, G
L	Argument interpreted as a <b>long double</b> for e, E, f, g, G

```
/*
printf2.c
further illustrates the printf() function
*/
#include <stdio.h> /* C I/O */
#include <math.h> /* sqrt() */

void main ()
{
    char ch      = 'C' ;
    char string[] = "love of wisdom" ;
    int i       = 1 ;
    float vat     = 17.5 ;
    double number   = 2.0 ;
    double mass     = 5.974e21 ;

    printf ("character ch: %c\n", ch) ;
    printf ("philosophy: %s\n", string) ;
    printf ("lvalue of i: %p; rvalue of i: %d\n", &i, i) ;
    printf ("V.A.T is currently: %+3.1f %%\n", vat) ;
    printf ("the square root of %g is: %f\n",
           number, sqrt (number)) ;
    printf ("the mass of earth, including atmosphere, is:
           %-10.3E tonnes\n", mass) ;
}
```

with output:

```
character ch: c
philosophy: love of wisdom
lvalue of i: 22F2; rvalue of i: 1
V.A.T. is currently: +17.5%
the square root of 2 is: 1.414214
the mass of earth, including atmosphere, is: 5.974E+21
tonnes
```

It is worth mentioning the function *sprintf()* (also declared in *STDIO.H*) which is similar in form and name to *printf()* but writes formatted output to a string buffer, rather than *stdout*:

```
int sprintf (char* buffer, const char* format[, argument, ...]);
```

An illustration of *sprintf()* is:

```
/*
sprintf.c
illustrates the sprintf() function
*/
#include <stdio.h> /* C I/O */

void main ()
{
    char string0[80] ;
    char string1[] = "a string" ;

    sprintf (string0, "this is %s\n", string1) ;

    printf ("%s", string0) ;
}
```

with output:

```
this is a string
```

From the above discussion it is clear that the *printf()* function is extremely versatile, but at the same very cumbersome when compared with the C++ standard output stream approach.

#### 17.14.4 Input from the Keyboard Using the *scanf()* Function

Let's now examine the *scanf()* function, which scans and formats input from the *stdin* stream (i.e. the keyboard) and has the following declaration:

```
int scanf (const char* fmt_string[, address, ...]) ; /* STDIO.H */
```

*scanf()* returns the number of input fields scanned, converted and stored. If the end-of-file is reached then *scanf()* returns EOF. Conversion characters have the following general form:

```
%[*][width][F|N][h|l|L]type
```

where:

[ * ]	Assignment suppression character (optional)
[width]	Width specifiers (optional)
[F   N]	Pointer size modifier (optional)
[h   l   L]	Argument type modifier (optional)
type	Conversion type character (required)

Similarly to `printf()`, an input conversion character sequence is preceded with the percent sign (%) and is followed by the format code. For each conversion character there is a corresponding function argument, and the order of the conversion characters matches the order of the arguments.

The assignment suppression character (\*) suppresses assignment of the next input field by allowing the input field to be scanned but not assigned to the next address argument. The width specifier defines the maximum number of characters to be read from the input field. If the input field has fewer characters than the value specified in the width specifier, all characters in the input field are read. If a non-white space or non-convertible character is encountered, all character scanning is terminated and all characters read up to that point are converted and stored. The pointer input size modifiers (F and N) override the default size of the address argument, whereas the argument type modifiers (h, l and L) override the default type of the address argument. The available conversion characters for `scanf()` are listed below in Table 17.9.

To input a **long double** using the `scanf()` function combine the l and f conversion characters (i.e. %lf). The pointer input size and argument type modifiers for `scanf()` are listed in Table 17.10.

It is important to note that `scanf()` interprets any white space character (blank space, new line, form feed, horizontal tab, vertical tab) as a signal to terminate input. This is illustrated in the following program:

**Table 17.9** Conversion characters for `scanf()`.

Conversion character	Type of argument	Input
<b>Character</b>		
c	Pointer to <b>char</b>	Character
s	Pointer to string	String
%	% character stored	% character
<b>Numeric</b>		
d	Pointer to decimal <b>int</b>	Decimal integer
D	Pointer to decimal <b>long</b>	Decimal integer
e, E	Pointer to <b>float</b>	Floating-point
f	Pointer to <b>float</b>	Floating-point
g, G	Pointer to <b>float</b>	Floating-point
i	Pointer to <b>int</b>	Decimal integer
l	Pointer to <b>long</b>	Decimal integer
o	Pointer to <b>intc</b>	Octal integer
O	Pointer to <b>long</b>	Octal integer
u	Pointer to <b>unsigned int</b>	Unsigned decimal
U	Pointer to <b>unsigned long</b>	Unsigned decimal
x	Pointer to <b>int</b>	Hexadecimal integer
X	Pointer to <b>long</b>	Hexadecimal integer
<b>Pointer</b>		
n	Pointer to <b>int</b>	
p	Pointer	Hexadecimal integer

**Table 17.10** Input size and argument type modifiers for `scanf()`.

Modifier	Meaning
Input size modifiers	
F	Overrides default size
N	Overrides default size
Argument type modifiers	
h	Converts input to <code>short int</code> for d,i,o,u,x
l	Converts input to <code>long int</code> for d,i,o,u,x Converts input to <code>double</code> for e,f,g No effect for c,D,I,n,O,p,s,U,X
L	Converts input to <code>long double</code> for e,f,g

```
/*
scanf0.c
illustrates the scanf() function
*/
#include <stdio.h> /* C I/O */

void main ()
{
    char name[20] ;

    printf ("enter your forename and surname: ") ;
    scanf ("%s", name) ;
    printf ("hi %s\n", name) ;
}
```

with some user interaction:

```
enter your forename and surname: eddie edwards
hi eddie
```

It may be evident from the above program that input variables are passed by their memory address by the use of pointers; remember that the name of an array in C/C++ is a synonym for its memory address.

Let's now take a look at a few programs which illustrate several features of the `scanf()` function. The first program illustrates that a non-white space character is discarded by `scanf()`:

```
/*
scanf1.c
further illustrates the scanf() function
*/
#include <stdio.h> /* C I/O */

// Point data structure
struct Point
{
```

```

double x, y, z ;
};

void main ()
{
    struct Point p ; /* note: keyword struct in definition */

    printf ("enter x, y and z members of a Point (x,y,z): ") ;
    scanf ("%lf,%lf,%lf", &p.x, &p.y, &p.z) ;
    printf ("point p: (%.2f, %.2f, %.2f)\n", p.x, p.y, p.z) ;
}

```

with some user interaction:

```

enter x, y and z members of a Point (x,y,z): (1,2,3)
point p: (1.00, 2.00, 3.00)

```

The above program illustrates the use of the "%lf,%lf,%lf" format specification for inputting the data members of a Point data structure object in the form "(x,y,z)". The parentheses and commas are read and discarded by *scanf()* and the three **double** floating-point numbers read. Note the use of the address-of operator (&) for inputting the floating-point numbers. If the non-white space characters are neglected then the input data must be separated by white space characters:

```

/*
scanf2.c
further illustrates the scanf() function
*/
/*...*/
void main ()
{
/*...*/
    scanf ("%lf%lf%lf", &p.x, &p.y, &p.z) ;
/*...*/
}

```

with some user interaction:

```

enter x, y and z members of a Point (x,y,z): 1 2 3
point p: (1.00, 2.00, 3.00)

```

However, if the input values 1, 2 and 3 are separated by a non-white space character such as a comma then *scanf()* will fail<sup>5</sup>:

```

enter x, y and z members of a Point (x,y,z): 1,2,3
point p: (1.00, -NAN, 4.1145e+45)

```

The following program illustrates the use of the assignment suppression (\*) character:

---

<sup>5</sup> ±NAN is the identifier of the IEEE not-a-number.

```

/*
scanf3.c
further illustrates the scanf() function
*/
/*...*/
void main ()
{
/*...*/
scanf ("%*c%lf%c%lf%c%lf", &p.x, &p.y, &p.z) ;
/*...*/
}

```

with some user interaction:

```

enter x, y and z members of a Point (x,y,z): [1,2;3]
point p: (1.00, 2.00, 3.00)

```

The assignment suppression character (\*) suppresses assignment of the next input field. Thus, the x,y and z Point data members can be input with any non-white space characters between and enclosing the members.

The following program illustrates the use of a width specifier with *scanf()* to prevent a user from entering characters outside the bounds of a string array:

```

/*
scanf4.c
further illustrates the scanf() function
*/
#include <stdio.h> /* C I/O */

void main ()
{
    char string[10] ;

    printf ("enter a string: ") ;
    scanf ("%9s", string) ; /* leave room for NULL terminator */
    printf ("string entered: %s\n", string) ;
}

```

with some user interaction:

```

enter a string: aspacelessstring
string entered: aspaceles

```

Try removing the width format specifier from *scanf()* in the above program and enter a string of ten characters or more.

The *scanf()* function also contains a format specifier '%[ . . . ]' often referred to as the *scanfset*. *scanf()* will only scan for characters specified within the square brackets:

```

/*
scanf5.c

```

---

```

illustrates the scanset feature of the scanf() function
*/
#include <stdio.h> /* C I/O */

void main ()
{
    char string[80] ;

    printf ("enter a string: ") ;
    scanf ("%[aeiou]", string) ;
    printf ("string entered: %s\n", string) ;
}

```

with some user interaction:

```

enter a string: utopia
string: u

```

As soon as a character is input which is not specified within the square brackets (in the present case a vowel) **string** will be NULL terminated.

If the character ^ is used as the first character between the square brackets **scanf()** scans for all characters except those specified within the square brackets:

```

/*
scanf6.c
further illustrates the scanset feature of the scanf()
function
*/
/*...*/
void main ()
{
    /*...*/
    scanf ("%[^aeiou]", string) ;
    /*...*/
}

```

with some user interaction:

```

enter a string: scanf
string: sc

```

Further, **scanf()** can scan for a range of characters by specifying a hyphen (-) format specifier:

```

/*
scanf7.c
further illustrates the scanset feature of the scanf()
function
*/
/*...*/

```

```
void main ()
{
/*...
scanf ("%[a-zA-Z]", string) ; /* case sensitive */
/*...
}
```

with some user interaction:

```
enter a string: Non-Whitespace
string: Non
```

In conclusion, the function `sscanf()` is similar to `scanf()` except that it scans and formats input from a string pointed to by `buffer` rather than `stdin` and has the following declaration:

```
int sscanf (const char* buffer,
            const char* fmt_string
            [,address, ...]) ; /* STDIO.H */
```

An example of `sscanf()` is:

```
/*
sscanf.c
illustrates the sscanf() function
*/
#include <stdio.h> /* C I/O */

void main ()
{
    char* polyhedra[5] = {"tetrahedron",
                           "hexahedron",
                           "octahedron",
                           "dodecahedron",
                           "icosahedron"} ;
    int faces[5] = {4, 6, 8, 12, 20} ;
    char buffer[5][50] ;
    char* poly[20] ;
    int face ;
    int i ;

    for (i=0; i<5; i++)
        sprintf (buffer[i], "%s %d", polyhedra[i], faces[i]) ;

    for (i=0; i<5; i++)
    {
        sscanf (buffer[i], "%s %d", &poly, &face) ;
        printf ("%12s: %2d faces\n", poly, face) ;
    }
}
```

with output:

```
tetrahedron: 4 faces
hexahedron: 6 faces
octahedron: 8 faces
dodecahedron: 12 faces
icosahedron: 20 faces
```

SSCANF.CPP first copies the array of strings of polyhedra names, polyhedra, and number of faces, faces, to the temporary buffer, buffer, using the *sprintf()* function. The *sscanf()* function then scans and formats the array of strings buffer and stores the formatted input in the memory addresses specified in the arguments poly and face. The *printf()* function displays the formatted output shown of arguments poly and face.

### 17.14.5 The *vprintf()*, *vscanf()*, *vsprintf()* and *vsscanf()* Functions

The function declarations of *vprintf()*, *vscanf()*, *vsprintf()* and *vsscanf()* are given in STDARG.H and are:

```
int vprintf (const char* format, va_list arg_list) ;
int vscanf  (const char* format, va_list arg_list) ;

int vsprintf (char* buffer,
              const char* format, va_list arg_list) ;
int vsscanf (const char* buffer,
              const char* format, va_list arg_list) ;
```

*vprintf()* and *vsprintf()* write formatted output to stdout and to a string respectively. *vscanf()* and *vsscanf()* scan and format input from stdin and from a string respectively. The *v...printf()* and *v...scanf()* functions are similar to *printf()* and *scanf()* except that they accept a pointer to a list of arguments rather than a list of arguments. The **typedef** *va\_list* is declared in STDARG.H and is a pointer to **void**:

```
typedef void* va_list ;
```

### 17.14.6 File Input and Output

As with standard input and output in C, the C file-handling mechanism is function-based and revolves around the STDIO.H library header file. The majority of the C file-handling functions make use of a pointer to a FILE structure which encapsulates various information about a given file, such as the file's name, mode and so on. We will begin this section by first examining the FILE structure and then several of the most popular C functions used for file handling.

### 17.14.7 The FILE Structure

The FILE structure encapsulates information relevant to the manipulation of a file stream, is declared in the STDIO.H header file and is of the form:

```
typedef struct
```

```
{
int          level ;
unsigned      flags ;
char         fd ;
unsigned char hold ;
int          bsize ;
unsigned char* buffer ;
unsigned char* curp ;
unsigned      istemp ;
short         token ;
} FILE ;
```

The `level` data member indicates how many characters remain in the file buffer. The `flags` member informs the operating system of the status of a file, such as read or write only, read and write, binary or text, file data is incoming or outgoing, error and end-of-file indicators. The file descriptor is denoted by `fd` and provides an association between a file and an object of the `FILE` structure. The `hold` member is used for holding file access if no buffer exists. The file buffer is denoted by `buffer` and is of size `bsize`. The current or active pointer of the file buffer is denoted by `curp`. The `istemp` member is a temporary file indicator and `token` is for file validity checking.

### 17.14.8 The `fopen()` Function

The declaration of the `fopen()` C library function is:

```
FILE* fopen (const char* filename, const char* mode) ;
```

`fopen()` opens a file specified by `filename` (which can also incorporate a path) in a mode specified by the second argument `mode`. If `fopen()` is successful in opening the specified file, a pointer to a `FILE` structure stream object is returned which is your link with the operating system for handling the file, else if `fopen()` is unsuccessful the NULL pointer is returned.

The permissible modes in which a file can be opened are listed in Table 17.11. The mode strings listed in Table 17.11 "r", "w", "a", "+" and "b" represent reading, writing, appending, updating and binary respectively. By default, a file is assumed to be a text file. If a file is opened for reading it is assumed that the file exists, and an error will be returned if it does not exist. If a file is opened for writing and it does not exist then it will be created, and if it already exists then all existing data will be deleted and the file length truncated to zero. Thus, beware of writing to an existing file. If a file is opened for appending it will be created if it does not exist. If a file is opened for reading and writing and it does not exist then it will be created.

Before we examine an example program which illustrates `fopen()`, let's discuss the `fclose()` and `fgetc()` functions.

### 17.14.9 The `fclose()` Function

Having opened a file using the `fopen()` function it is the responsibility of the programmer to close the file after use. It is important to formally close a file when access to the file is no longer required so that the file is not accidentally damaged, and because the operating system

**Table 17.11** *fopen()* mode values.

<i>mode</i>	<i>Meaning</i>
Text	Open for read only
"r"	Create for write only
"w"	Open to append or if file does not exist then create for write only
"a"	Open for read and write
"r+"	Open for read and write
"w+"	Create for read and write
"a+"	Open or create to append
Binary	
"rb"	Open for read only
"wb"	Create for write only
"ab"	Open to append or if file does not exist then create for write only
"rb+"	Open for read and write
"wb+"	Create for read and write
"ab+"	Open or create to append

stipulates a limit on the number of files which can be simultaneously open at any given time<sup>6</sup>. The function *fclose()* closes a file whose pointer to the FILE structure is given by *f\_ptr*:

```
int fclose (FILE* f_ptr) ;
```

*fclose()* returns zero if the file was successfully closed, else EOF if an error occurred. If *fclose()* is successful the FILE pointer, *f\_ptr*, is no longer valid and if a file is to be re-opened a further call to *fopen()* is required.

#### 17.14.10 The *fgetc()* Function

The *fgetc()* function reads a character at a time from an input file stream given a FILE structure pointer *f\_ptr* returned by *fopen()*:

```
int fgetc (FILE* f_ptr) ;
```

If successful, *fgetc()* returns the next character from the specified file input stream after it is converted to **int**. If *fgetc()* is unsuccessful or end-of-file is reached then EOF is returned.

#### 17.14.11 Reading a Disk File

The following program illustrates the C *fopen()*, *fclose()* and *fgetc()* functions for reading a file and displaying it to the screen:

```
/*
o&r&d&c.c
illustrates opening, reading, displaying and closing a file
using the C fopen(), fgetc() and fclose() file handling
functions
*/
```

6 The Borland C++ compiler (version 5.0) for Windows specifies the number of simultaneously open files by the define FOPEN\_MAX in STDIO.H, which is set to the define \_NFILE\_ in \_NFILE.H and can equal either 20 or 40.

```

#include <stdio.h>    /* C I/O */
#include <stdlib.h>   /* exit() */

void main ()
{
    char file_name[50] ;
    FILE* f_ptr ;
    int c ;

    /* get a filename */
    printf ("enter a filename: ") ;
    gets (file_name) ;

    /* open file */
    if ((f_ptr = fopen (file_name, "r")) == NULL)
    {
        printf ("sorry-can't open file \'%s\'\n", file_name) ;
        exit (EXIT_FAILURE) ;
    }

    /* read and display file */
    while ((c = fgetc (f_ptr)) != EOF)
        putchar (c) ;

    /* close file */
    fclose (f_ptr) ;
}

```

with some user interaction:

```
enter a filename: o&r&d&c.c
```

and outputs itself to the display screen.

The `gets()` function gets a string of characters, `string`, from the `stdin` stream until a new-line character is encountered and has the declaration:

```
char* gets (char* string) ; /* STDIO.H */
```

`gets()` returns a pointer to the string when successful and the `NULL` pointer when an error or end-of-file is reached.

The `putchar()` function outputs the character `c` to the `stdout` stream:

```
int putchar (int c) ; /* STDIO.H */
```

`putchar()` returns the character when successful and `EOF` when unsuccessful.

The above program illustrated a use of `fopen()` to open a file requested by a user. The file is opened for reading only ("r") and the `FILE` structure pointer returned by `fopen()` is tested against the `NULL` pointer within the `if`-statement. If the file-opening procedure is unsuccessful a message is displayed to `stdout` using `printf()` and the program is terminated using `exit()`:

```
/* ... */
```

```
if ((f_ptr = fopen (file_name, "r")) == NULL)
{
    printf ("sorry-can't open file \">%s\"\\n", file_name) ;
    exit (EXIT_FAILURE) ;
}
```

If the file is successfully opened a **while**-loop is used to get each character of the file using *fgetc()*. Each character obtained from the file is tested against EOF to see if the end of file has been reached:

```
while ((c = fgetc (f_ptr)) != EOF)
    putchar (c) ;
```

Finally, the file opened is closed using the *fclose()* function:

```
fclose (f_ptr) ;
```

### 17.14.12 The *fgets()* Function

The previous program example worked fine reading the contents of a file a character at a time, but we can alternatively read the contents of a file line by line using the *fgets()* function:

```
char* fgets (char* buffer, int n, FILE* f_ptr) ; /* STDIO.H */
```

*fgets()* will continue to read a line of data from a file specified by *f\_ptr* until either *n-1* characters have been read or a new-line character is read, placing each character in the string array pointed to by *buffer*. If successful, *fgets()* returns the string pointed to by *buffer*, else it returns NULL if an end of file or error occurred. Note the similarity between the C *fgetc()* and *fgets()* functions and the C++ member functions *istream::get()* and *istream::getline()*.

Let's now modify O&R&D&C.C to use *fgets()*:

```
/*
o&r&d&c1.c
illustrates opening, reading, displaying and closing a file
using the C fopen(), fgets() and fclose() file handling
functions
*/
#include <stdio.h> /* C I/O */
#include <stdlib.h> /* exit() */

#define MAX_CHAR 80

void main ()
{
    char file_name[50] ;
    FILE* f_ptr ;
    char buffer[MAX_CHAR] ;

    /* get a filename */
    printf ("enter a filename: ") ;
```

```

gets (file_name) ;

/* open file */
if ((f_ptr = fopen (file_name, "r")) == NULL)
{
    printf ("sorry-can't open file \"%s\"\n", file_name) ;
    exit (EXIT_FAILURE) ;
}

/* read and display file */
while (fgets (buffer, MAX_CHAR, f_ptr) != NULL)
    printf ("%s", buffer) ;

/* close file */
fclose (f_ptr) ;
}

```

### 17.14.13 The *fputc()* Function

The *fputc()* function outputs a character *c* to the stream pointed to by the FILE structure pointer *f\_ptr*:

```
int fputc (int c, FILE* f_ptr) ;
```

If *fputc()* is successful the character output is returned, else EOF is returned.

### 17.14.14 Writing to a Disk File

To illustrate the *fputc()* function, let's revisit the CPP\_CPY.CPP program, which illustrated command line arguments in C++, by developing a program which operates similarly to the MS-DOS COPY command:

```

/*
c_cpy.c
illustrates the fputc() function
*/
#include <stdio.h> /* C I/O */
#include <stdlib.h> /* exit() */
#include <string.h> /* strcmp() */

void main (int argc, char* argv[])
{
FILE* fin_ptr, * fout_ptr ;
char c ;

/* check command-line arguments */
if (argc != 3)
{

```

```

printf ("format: c_cpy src_file dst_file\n") ;
exit (EXIT_FAILURE) ;
}

/* prevent self copy */
if (strcmp (argv[1], argv[2]) == 0)
{
printf ("source file must be different from
        destination file\n") ;
exit (EXIT_FAILURE) ;
}

/* open source and destination files */
if ((fin_ptr = fopen (argv[1], "rb")) == NULL)
{
printf ("sorry-can't open file \'%s\'\n", argv[1]) ;
exit (EXIT_FAILURE) ;
}
if ((fout_ptr = fopen (argv[2], "wb")) == NULL)
{
printf ("sorry-can't open file \'%s\'\n", argv[2]) ;
exit (EXIT_FAILURE) ;
}

/* copy file */
while ((c = fgetc (fin_ptr)) != EOF)
    fputc (c, fout_ptr) ;

/* close files */
fclose (fin_ptr) ;
fclose (fout_ptr) ;
}

```

with some user interaction:

```
C:\>c_cpy autoexec.bat autoexec.old
```

after which a copy of the file AUTOEXEC.BAT is made and written to AUTOEXEC.OLD

C\_CPY.CPP first performs a check to confirm that the number of command line arguments is correct, and then uses the C++ library function *strcmp()* to confirm that the names of the source and destination files are different. The source and destination files are then opened for reading and writing, respectively. The contents of the source file are copied to the destination file using the *fputc()* function, after which both the opened source and destination files are closed.

### 17.14.15 The *fputs()* Function

The *fputs()* function outputs a string pointed to by *string* to the stream pointed to by the FILE structure pointer *f\_ptr*:

```
int fputs (const char* string, FILE* f_ptr) ;
```

*fputs()* returns the last character written if successful, else EOF if unsuccessful. Note that *fputs()* does not append the new line character and does not copy the NULL terminator. The following program modifies C\_CPY.C to use the *fputs()* function instead of *fputc()*:

```
/*
c_cpy1.c
illustrates the fputs() function
*/
/*...*/
#define MAX_CHAR 80

void main (int argc, char* argv[])
{
    FILE* fin_ptr, * fout_ptr ;
    char buffer[MAX_CHAR] ;
    /*...*/
    while (fgets (buffer, MAX_CHAR, fin_ptr) != NULL)
        fputs (buffer, fout_ptr) ;
    /*...*/
}
```

#### 17.14.16 The *fread()* and *fwrite()* Functions

The *fread()* and *fwrite()* functions allow a block of data (text or binary) to be read from or written to a file.

The *fread()* function reads n items, each of size size, from a file pointed to by *f\_ptr* and places them in memory at a location pointed to by *buffer*:

```
size_t fread (void* buffer, size_t size, size_t n,
              FILE* f_ptr) ; /* STDIO.H */
```

If *fread()* is successful the number of items read is returned, else a short count is returned if an error is encountered or end-of-file is reached.

In contrast to *fread()*, the *fwrite()* function writes n items, each of size size, from a memory location pointed to by *buffer* to a file pointed to by *f\_ptr*:

```
size_t fwrite (const void* buffer, size_t size, size_t n,
               FILE* f_ptr) ; /* STDIO.H */
```

If *fwrite()* is successful the number of items written is returned, else a short count is returned.

The following program helps to illustrate the *fread()* and *fwrite()* functions:

```
/*
fr&fw.c
illustrates the fread() and fwrite() functions
*/
#include <stdio.h> /* C I/O */
#include <stdlib.h> /* exit(), rand(), srand() */
#include <time.h> /* time() */
```

```
#define MAX_NUM 10

void main ()
{
FILE* f_ptr ;
char filename[] = "FR&FW.TXT" ;
int rand_num, i ;
time_t t ;

/* fwrite(): */

/* open file for writing */
if ((f_ptr = fopen (filename, "wb")) == NULL)
{
printf ("sorry-can't open file \\"%s\"\n", filename) ;
exit (EXIT_FAILURE) ;
}

/* write random numbers (0:99) to file */
rand((unsigned)time (&t)) ;
for (i=0; i<MAX_NUM; i++)
{
rand_num = rand () % 100 ;
if ((fwrite (&rand_num, sizeof (int), 1, f_ptr)) == 0)
{
printf ("sorry-write error\n") ;
exit (EXIT_FAILURE) ;
}
}

/* close file */
fclose (f_ptr) ;

/* fread(): */

/* open file for reading */
if ((f_ptr = fopen (filename, "rb")) == NULL)
{
printf ("sorry-can't open file \\"%s\"\n", filename) ;
exit (EXIT_FAILURE) ;
}

/* read and display random numbers from file */
for (i=0; i<MAX_NUM; i++)
{
if ((fread (&rand_num, sizeof (int), 1, f_ptr)) == 0)
{
printf ("sorry-read error\n");
exit (EXIT_FAILURE) ;
}
printf ("%2d\n", rand_num) ;
```

```
}

/* close file */
fclose (f_ptr) ;
}
```

The above program first opens a file for writing in binary mode called FR&FW.TXT. Ten random numbers, between zero and 100, are then written one at a time using a **for**-loop to FR&FW.TXT using the *fwrite()* function. FR&FW.TXT is then closed for writing and reopened for reading in binary mode, when the random numbers previously written are read using  *fread()* and displayed to the screen.

#### 17.14.17 The *fprintf()* and *fscanf()* Functions

In a similar way to *printf()* and *scanf()*, the C++ library includes the functions *fprintf()* and *fscanf()*, which write formatted output to a file stream and format input from a file stream, respectively. The declarations of *fprintf()* and *fscanf()* are to be found in STDIO.H and are of the form:

```
int fprintf (FILE* f_ptr, const char* format[, argument, ...]) ;

int fscanf (FILE* f_ptr, const char* format[, address, ...]) ;
```

The operation of *fprintf()* and *fscanf()* is identical to that of *printf()* and *scanf()*, except that they operate on file streams rather than the standard input and output streams.

#### 17.14.18 The *vfprintf()* and *vfscanf()* Functions

The functions *vfprintf()* and *vfscanf()* write formatted output to and scan and format input from a stream, respectively, and are declared in the STDARG.H header file:

```
int vfprintf (FILE* f_ptr, const char* format,
              va_list arg_list) ;

int vfscanf (FILE* f_ptr, const char* format,
             va_list arg_list) ;
```

The *vfprintf()* and *vfscanf()* functions are similar to *fprintf()* and *fscanf()*, except that they accept a pointer to a list of arguments rather than a list of arguments.

#### 17.14.19 Random File Access

When we discussed random file access in C++ we saw the use of objects of classes **ifstream**, **ofstream** and **fstream** for manipulating file input, output, and input and output streams, respectively. For stream input the **istream::seekg()** member function was used for moving to a specified position relative to the beginning, end or current pointer position of the stream, whereas the **istream::tellg()** member function returns the current position of an input stream. In the case of stream output the **seekp()** and **tellp()** member functions of **class ostream** are used. Recall that **ifstream** is derived from classes **fstreambase** and **istream** and that **ofstream** is derived from classes **fstreambase** and **ostream**.

Random file access in C is similar in principle to that discussed previously in C++, in that access is described in terms of a file pointer, except that functions rather than member functions are used to operate on both input and output file streams. The four main functions for performing random file access in C are described in the next subsection.

### 17.14.20 The *fseek()*, *ftell()*, *fgetpos()* and *fsetpos()* Functions

The key functions for performing random file access in C are *fseek()*, *ftell()*, *fgetpos()* and *fsetpos()*, which are all declared in STDIO.H:

```
int fseek (FILE* f_ptr, long offset, int whence) ;

long int ftell (FILE* f_ptr) ;

int fgetpos (FILE* f_ptr, fpos_t* pos) ;

int fsetpos (FILE* f_ptr, const fpos_t* pos) ;
```

where **fpos\_t** is a **long** **typedef** which represents the file position type and is declared in STDIO.H.

The *fseek()* function sets the current file pointer in the file pointed to by *f\_ptr* to *offset* bytes from the position specified in the *whence* argument. The *whence* argument can be one of the define identifiers listed in Table 17.12.

To set the file pointer position *offset* bytes from the start of a file, *whence* should be set to SEEK\_SET, whereas SEEK\_END sets the file pointer position *offset* bytes from the end of a file and SEEK\_CUR sets the file pointer position *offset* bytes from the current position. The *fseek()* function returns zero if successful or EBADF, indicating a bad file pointer, EINVAL, indicating an invalid argument, or ESPIPE, indicating an illegal seek.

The *ftell()* function returns a **long int** value, which is the offset in bytes measured from the start of the file, specified by *f\_ptr*, to the current file position. If *ftell()* is unsuccessful the value -1L is returned and the global variable *errno* is set to either EBADF, indicating a bad file pointer, or ESPIPE, which indicates an illegal seek.

The *fgetpos()* function retrieves the current file pointer position, *pos*, which corresponds to the file specified by *f\_ptr*. If *fgetpos()* is successful zero is returned, else the global variable *errno* is set to either EBADF or EINVAL.

The *fsetpos()* function sets the current file pointer position to *pos* for the file specified by *f\_ptr*. If *fsetpos()* is successful zero is returned, else *errno* is set to either EBADF or EINVAL.

The following program illustrates the *fseek()* and *fgetpos()* functions and is a C version of the program CGP.CPP:

```
/*
c_rnd_ac.c
illustrates random file access in C
```

**Table 17.12** File pointer identifiers for *fseek()*.

<i>whence</i>	<i>Value</i>	<i>Meaning</i>
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current file pointer position
SEEK_END	2	End of file

```

using the fseek(), ftell(), fgetpos() and fsetpos() functions
*/
#include <stdio.h> /* C I/O */
#include <stdlib.h> /* exit() */

#define MAX_CHAR 80

void main ()
{
    FILE* f_ptr ;
    char filename[] = "C_RND_AC.C";
    char buffer[MAX_CHAR] ;
    long start, end, offset ;
    fpos_t position ;

    /* open file */
    if ((f_ptr = fopen (filename, "rb")) == NULL)
    {
        printf ("sorry-can't open file \"%s\"\n", filename) ;
        exit (EXIT_FAILURE) ;
    }

    /* determine file length */
    fseek (f_ptr, 0, SEEK_SET) ; /* go to start */
    fgetpos (f_ptr, &position) ;
    start = position ;

    fseek (f_ptr, 0, SEEK_END) ; /* go to end */
    fgetpos (f_ptr, &position) ;
    end = position ;

    /* get an offset */
    printf ("enter an offset(+) from the beginning of file
           \"%s\"\nwhich has a file length of %d characters: ",
           filename, (end-start)) ;
    offset = atoi (gets (buffer)) ;

    /* go to offset */
    fseek (f_ptr, offset, SEEK_SET) ;

    /* display file to screen */
    while (fgets (buffer, MAX_CHAR, f_ptr) != NULL)
        printf ("%s", buffer) ;

    /* close file */
    fclose (f_ptr) ;
}

```

with some user interaction:

```
enter an offset(+) from the beginning of file "C_RND_AC.C"
```

---

```

which has a file length of 1159 characters: 1040
while (fgets (buffer, MAX_CHAR, f_ptr) != NULL)
    printf ("%s", buffer) ;

/* close file */
fclose (f_ptr) ;
}

```

### **17.14.21 The *feof()*, *ferror()*, *fflush()*, *flushall()*, *freopen()*, *clearerr()*, *remove()*, *rename()* and *rewind()* Functions**

The C file stream library is very comprehensive, and as a result it is simply not possible to examine every function in detail. It is therefore recommended that you consult your compiler's documentation for a more complete discussion. However, there are a few C file stream functions that are frequently used and warrant a mention. All of the following functions are declared in the STDIO.H header file.

The *feof()* function tests the file stream specified by *f\_ptr* for an end-of-file:

```
int feof (FILE* f_ptr) ;
```

If the end-of-file is reached a non-zero value is returned, else zero is returned. By using *feof()* the **while**-loop of C\_RND\_AC.C could be written as:

```

while (!feof (f_ptr))
{
    fgets (buffer, MAX_CHAR, f_ptr) ;
    printf ("%s", buffer) ;
}
```

The *ferror()* function tests the file stream specified by *f\_ptr* for read and write errors:

```
int ferror (FILE* f_ptr) ;
```

*ferror()* returns a non-zero value if an error is detected. Once a stream error has been set it remains set until either *clearerr()* or *rewind()* is called. The functions *clearerr()* and *rewind()* have the following similar declarations:

```

void clearerr (FILE* f_ptr) ;

void rewind (FILE* f_ptr) ;
```

Both *clearerr()* and *rewind()* reset the error and end-of-file indicators to zero for the file stream specified by *f\_ptr*, but *rewind()* also moves the file pointer position to the start of the file pointed to by *f\_ptr*.

The function *fflush()* has the following declaration:

```
int fflush (FILE* f_ptr) ;
```

and flushes the file stream specified by `f_ptr` when the stream has buffered output. If `fflush()` is successful zero is returned, else EOF is returned. The similar function `flushall()`:

```
int flushall () ;
```

flushes all buffers corresponding to *open* input streams and writes buffered output for *open* output streams. `flushall()` returns the number of open streams.

The `freopen()` function associates a new file, `filename`, in place of an open stream, `stream`, and has the following declaration:

```
FILE* freopen (const char* filename,  
              const char* mode, FILE* stream) ;
```

The new file is opened according to the value of `mode`, which may have the same values as for the `fopen()` function. If `freopen()` is successful the argument `stream` is returned, else NULL is returned if an error occurs. `freopen()` is useful for redirecting the standard streams `stdin`, `stdout` and `stderr` to a specified file.

The `rename()` function renames a file specified by `oldname` to the file specified by `newname`:

```
int rename (const char* oldname, const char* newname) ;
```

and returns zero if successful, else -1 with `errno` set to either EACCES, indicating that permission is denied, ENONET, indicating that the specified file or directory is non-existent, or ENOTSAM, indicating that the drives of the two files are different.

Finally, the `remove()` function deletes the file specified by `filename`:

```
int remove (const char* filename) ;
```

and returns zero if successful, else -1 with `errno` set to either EACCES or ENOENT.

## 17.15 A Device-Independent Bitmap **class** for Windows

Let's conclude this chapter by discussing a device-independent bitmap (DIB) **class** for the Windows operating system. The Windows bitmap file format is a relatively simple and general-purpose format for the storing of device-independent bitmaps, which are usually attributed either the BMP or DIB file extension. Owing to the popularity of the Windows operating system and acknowledging that most commercial graphics programs support the Windows bitmap file format, it is a useful format to examine. Also, the device-independent bitmap **class**, `DIBitmap`, presented in this section will prove useful in Chapter 20 for storage and manipulation of raytraced images.

There are several different bitmap file formats, of which the Windows DIB is just one. If you are interested in alternative formats or completely confused after my description of the DIB format, then an excellent source of further reading of the major bitmap file formats can be found in Luse (1993).

The following two subsections first examine the Windows bitmap file format and then discuss the implementation of the `DIBitmap` **class**.

### 17.15.1 The Windows Device-Independent Bitmap File Format

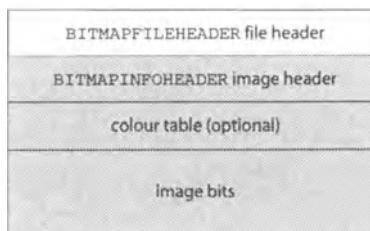
The Windows bitmap file format generally consists of the following four components: (1) a file header defined by the BITMAPFILEHEADER structure; (2) an image header defined by the BITMAPINFOHEADER structure; (3) a colour table or a palette array; and (4) an array of bits that define the actual bitmap image. Figure 17.5 schematically illustrates the Windows bitmap file format. Contrary to the majority of bitmap file formats, the Windows bitmap file format stores the image bits from the bottom row to the top row, with each row beginning with the leftmost pixel.

The BITMAPFILEHEADER structure is declared in the WINDOWS.H header file and has five fields, which are listed in Table 17.13.

The BITMAPFILEHEADER structure is present at the start of all DIB files. The bfType field indicates whether or not the file format is compatible with that of a Windows bitmap file. The signature 'BM' (4D42h) identifies the file as a Windows bitmap file. The bfSize field indicates the total size of the bitmap in bytes. The fields bfReserved1 and bfReserved2 are reserved structure members and their values should be set to zero. The bfOffBits field holds the offset position, in bytes, from the beginning of the file to the start of the image bits.

The BITMAPINFOHEADER structure follows the BITMAPFILEHEADER structure and encapsulates the fields listed in Table 17.14, which describe the bitmap image.

The biSize field indicates the size of the BITMAPINFOHEADER structure and should be set to 40. biWidth and biHeight designate the width and height of the bitmap in pixels respectively. biPlanes is the number of colour planes of the bitmap image, and at present the Windows bitmap file format only supports single-plane bitmaps of either 1, 4, 8 or 24 bits per pixel. biBitCount is the number of colour bits per pixel and can be set to either 1, 4, 8 or 24. The biCompression field indicates the compression scheme used, if any. The Windows bitmap file format supports only the RLE image compression scheme, with only 4- and 8-bit images allowing compression. A value of 1 for biCompression indicates RLE-8 compression, and a value of 2 for biCompression indicates RLE-4 compression. Since compression is only supported for 4- and 8-bit images, and because the majority of applications do not compress DIB images, the DIBitmap **class** described below will ignore image compression. biSizeImage indicates the size of the bitmap bits in bytes, but should generally only be used



**Fig. 17.5** The components of the Windows bitmap file format.

**Table 17.13** Fields of the BITMAPFILEHEADER structure.

Field	Type	Description
bfType	UINT ( <b>unsigned int</b> )	Signature 'BM' (bitmap)
bfSize	DWORD ( <b>unsigned long</b> )	File size in bytes
bfReserved1	UINT	Set to 0
bfReserved2	UINT	Set to 0
bfOffBits	DWORD	Offset to image bits from start of file

**Table 17.14** Fields of the BITMAPINFOHEADER structure.

Field	Type	Description
biSize	DWORD	Size of this structure in bytes (should be 40)
biWidth	LONG ( <b>signed long</b> )	Width of the bitmap in pixels
biHeight	LONG	Height of the bitmap in pixels
biPlanes	WORD ( <b>unsigned short</b> )	Number of colour planes (set to 1)
biBitCount	WORD	Number of colour bits per pixel (1, 4, 8 or 24)
biCompression	DWORD	Compression scheme used
biSizeImage	DWORD	Size of the bitmap bits in bytes
biXPelsPerMeter	LONG	Horizontal resolution in pixels per metre
biYPelsPerMeter	LONG	Vertical resolution in pixels per metre
biClrUsed	DWORD	Number of colours in bitmap or set to zero
biClrImportant	DWORD	Number of colours in bitmap that are considered important or set to zero

if compression is used because there are a number of applications which do not assign a value to biSizeImage. Also, bear in mind that each row of the bitmap bits is padded to the right with an integral number of double words, so that the number of bytes in each bitmap row is evenly divisible by 4. The two fields biXPelsPerMeter and biYPelsPerMeter indicate the horizontal and vertical resolution of the bitmap in pixels per metre. biClrUsed indicates the number of colours used in the bitmap. If biClrUsed is set to zero and the number of colour bits per pixel is either 1, 4 or 8 then the BITMAPINFOHEADER structure is followed by a colour table. Finally, the biClrImportant field is the number of colours considered important in the bitmap image.

An additional structure declared in the WINDOWS.H header file, which will prove useful later, is the BITMAPINFO structure, which encapsulates both a BITMAPINFOHEADER object field, bmiHeader, and a pointer to the RGBQUAD structure field, bmiColors:

```
typedef struct tagBITMAPINFO // WINDOWS.H
{
    BITMAPINFOHEADER bmiHeader ;
    RGBQUAD           bmiColors[1] ;
} BITMAPINFO ;
```

The RGBQUAD structure encapsulates the four fields listed in Table 17.15. The rgbBlue, rgbGreen and rgbRed fields are each of type **unsigned char**, and respectively represent the blue, green and red primary intensities of a single colour as an 8-bit value. The rgbReserved field is an unused reserved data member.

So why the emphasis on device independence, and what is a colour table? Basically, the DIB format is referred to as *device-independent* because it incorporates a colour table or an array of palette entries. The colour table is a concise way of representing an RGB colour by assigning an integer value to a particular colour and then using the value as an index to the colour table rather than the RGB colour itself. If each bitmap image contains its own colour table the bitmap does not have to be compatible with the specification of a particular display device. In the case of a monochrome bitmap consisting of 1 colour bit per pixel, the most significant bit of each image byte is used as an index to the two-entry colour table. For a 16 colour bitmap consisting of four colour bits per pixel the four most significant bits (0:15) act as an index to the 16-entry colour table. For a 256 colour bitmap each byte corresponds exactly to a single pixel with a 256-entry colour table. In the case of a 24-bit colour bitmap a colour table is not required because each set of three bytes correspond exactly to an RGB colour, thus giving a total of  $2^{24}$

**Table 17.15** Fields of the RGBQUAD structure.

<i>Field</i>	<i>Type</i>	<i>Description</i>
rgbBlue	BYTE ( <code>unsigned char</code> )	Blue intensity (0:255)
rgbGreen	BYTE	Green intensity (0:255)
rgbRed	BYTE	Red intensity (0:255)
rgbReserved	BYTE	Reserved byte (set to zero)

**Table 17.16** A typical colour table for a 16-colour bitmap.

<i>Index</i>	<i>Red</i>	<i>Green</i>	<i>Blue</i>	<i>RGB colour</i>
0	0	0	0	Black
1	128	0	0	Maroon
2	0	128	0	Dark green
3	128	128	0	Mustard
4	0	0	128	Navy blue
5	128	0	128	Purple
6	0	128	128	Olive
7	128	128	128	Dark grey
8	192	192	192	Grey
9	255	0	0	Red
10	0	255	0	Green
11	255	255	0	Yellow
12	0	0	255	Blue
13	255	0	255	Magenta
14	0	255	255	Cyan
15	255	255	255	White

(16 777 216) different colours. Table 17.16 illustrates a typical non-dithered colour table corresponding to a 16-colour bitmap.

### 17.15.2 The DIBitmap class

This section examines an implementation of a **class** called **DIBitmap** for the manipulation of bitmapped images in the Windows bitmap file format. Unless you are familiar with programming for Windows much of what follows may appear confusing, particularly the use of data type **typedefs** specific to Windows and the use of the Windows API functions. However, try to focus your attention on the design of the **DIBitmap class** and the reading and writing of data structures and bitmap bits from a disk file using a C approach. The **class** declaration of **DIBitmap** is listed below in **DIB.H** with corresponding implementation file **DIB.CPP**:

```
// dib.h
// device independent bitmap class for Windows

#ifndef _DIB_H // prevent multiple includes
#define _DIB_H

#define STRICT // Windows strict conformance

#include <windows.h> // Windows
#include <mem.h> // memcpy()
#include <math.h> // log()
```

```

class DIBitmap
{
private:
    HANDLE           handle ;
    BYTE huge*      bits ;
    BITMAPINFO FAR* bmp_info ;
    int             width ;
    int             height ;
    int             pixel_bits ;
    WORD             mode ;

public:
    enum { ACTUAL, STRETCH } ;
    // constructors & destructor
    DIBitmap (const char* filename) ;
    DIBitmap (int w, int h, int bit_count,
              WORD m=DIB_RGB_COLORS) ;
    DIBitmap (const DIBitmap& dib) ;
    ~DIBitmap () ;
    // member functions
    int Width () const { return width ; }
    int Height () const { return height ; }
    DWORD NumberOfColours () const ;
    int PixelBits () const { return pixel_bits ; }
    WORD Mode () const { return mode ; }
    const BITMAPINFO FAR* BMPInfo () const
        { return bmp_info ; }
    BITMAPINFO FAR* BMPInfo () { return bmp_info ; }
    const BITMAPINFOHEADER FAR* BMPInfoHeader () const
        { return &bmp_info->bmiHeader ; }
    BITMAPINFOHEADER FAR* BMPInfoHeader ()
        { return &bmp_info->bmiHeader ; }
    const RGBQUAD FAR* BMPColours () const
        { return bmp_info->bmiColors ; }
    RGBQUAD FAR* BMPColours ()
        { return bmp_info->bmiColors ; }
    RGBQUAD Colour (int index) const ;
    const BYTE huge* Bits () const { return bits ; }
    BYTE huge* Bits () { return bits ; }
    HBITMAP HandleToBitmap (const HDC& hdc) ;
    void DisplayBitmap (const HDC& hdc,
                        int x0=0, int y0=0, int w=0, int h=0,
                        WORD display_mode=DIBitmap::ACTUAL) ;
    BOOL Write (const char* filename) ;
}; // class DIBitmap

#endif // _DIB_H

```

DIBitmap encapsulates seven **private** data members. handle indicates the global memory block in which the bitmap BITMAPINFOHEADER structure, colour table (if one is present) and the array of bits will be stored. It is important to note that the allocated global memory block is not just for the array of bits but is of sufficient size to hold the bitmap image header,

colour table and array of bits. The `bits` data member is a **huge** pointer to store a pointer to the bitmap bits. The `bits` pointer is a **huge** pointer because bitmap images are generally greater than 64 kbyte in size and consequently occupy more than one memory segment. `bmp_info` is a far pointer to the `BITMAPINFO` structure discussed above. The width and height of the bitmap are designated by the `width` and `height` data members, whereas `pixel_bits` indicates the number of colour bits per pixel (i.e. 1, 4, 8 or 24). The `mode` data member indicates the type of bitmap colour table. Various access member functions are defined to allow access to `DIBitmap`'s data members, such as `Width()`, `Height()`, `PixelBits()`, `Mode()`, `BMPInfo()` and `Bits()`.

There are two global functions, `ImageColours()` and `ImageBits()`, defined in `DIB.CPP` which, respectively, return the number of image colours given the number of colour bits per pixel and, conversely, the number of colour bits per pixel given the number of colours:

```
DWORD ImageColours (WORD bit_count)
{
    if (bit_count == 1 || bit_count == 4 || bit_count == 8)
        return 1 << bit_count ;
    else if (bit_count == 24)
        return 16777216 ;
    return 0L ;
}

WORD ImageBits (DWORD num_colours)
{
    if (num_colours == 2 || num_colours == 16 ||
        num_colours == 256 || num_colours == 16777216)
        return (WORD) (log (num_colours) / log (2.0)) ;
    else
        return 0 ;
}
```

Before examining the remaining member functions of `DIBitmap`, let's take a look at the two constructors, copy constructor and destructor of `DIBitmap`. The one-argument constructor defines a `DIBitmap` object given a bitmap path and filename:

```
DIBitmap::DIBitmap (const char* filename)
: handle (NULL), bits (NULL), bmp_info (NULL),
  width (0), height (0), pixel_bits (0), mode (DIB_RGB_COLORS)
{
    // open bmp file
    HFILE hfile ;
    if (HFILE_ERROR == (hfile = _lopen(filename, READ)))
    {
        MessageBox (NULL,
                    "can't open file", "DIBitmap()", 
                    MB_ICONEXCLAMATION|MB_OK) ;
        return ;
    }
    // read bmp header
    BITMAPFILEHEADER bfh ;
```

```

if (_lread(hfile, (void huge*)&bfh, sizeof bfh) != sizeof bfh)
{
    MessageBox (NULL,
                "header file problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
// check that bmp is a Windows bmp
if (bfh.bfType != 'BM')
{
    MessageBox (NULL,
                "DIB is not of Windows format", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
// check bmp info header size and width
DWORD info_header_size ;
BITMAPINFOHEADER bih ;
if (_lread(hfile, &info_header_size, sizeof info_header_size)
     != sizeof info_header_size
     && info_header_size != sizeof(BITMAPINFOHEADER) )
{
    MessageBox (NULL,
                "header size problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
bih.biSize = info_header_size ;
if (_lread(hfile, &bih.biWidth,
            (int)info_header_size-sizeof(DWORD))
     != (int)info_header_size-sizeof(DWORD) )
{
    MessageBox (NULL,
                "header width problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
// colour bits per pixel
pixel_bits = bih.biBitCount ;

// width and height
width = (int)bih.biWidth ;
height = (int)bih.biHeight ;

// size of bytes
int bytes_per_row = ((width*bih.biBitCount+31)/32)*4 ;
DWORD bmp_size      = (DWORD)bytes_per_row * (DWORD)height ;

```

```
// 1, 4 or 8 bits
if (pixel_bits == 1 || pixel_bits == 4 || pixel_bits == 8)
{
// if biClrUsed=0 then a colour table is present
if (!bih.biClrUsed)
    bih.biClrUsed = NumberOfColours () ;

int colour_table_size = (int)NumberOfColours () *
                           sizeof (RGBQUAD) ;

// if compression used
if (bih.biCompression)
    bih.biSizeImage = bmp_size ;

// handle
handle = GlobalAlloc (GMEM_ZEROINIT|GMEM_MOVEABLE,
                      bih.biSize+colour_table_size+bmp_size) ;
if (!handle)
{
    MessageBox (NULL,
                "handle problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    return ;
}
bmp_info = (LPBITMAPINFO)GlobalLock (handle) ;
bmp_info->bmiHeader = bih ;

// get colour table
if (_lread(hfile, (BYTE FAR*)bmp_info +
            (int)bih.biSize, colour_table_size)
        != colour_table_size)
{
    MessageBox (NULL,
                "colour table problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}

// get bits
bits = (BYTE FAR*)bmp_info + (int)bih.biSize +
       colour_table_size ;
_llseek (hfile, bfh.bfOffBits, 0) ;
if (_hread(hfile, bits, bmp_size) != bmp_size)
{
    MessageBox (NULL,
                "reading bits problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
```

```

    // close file
    _lclose (hfile) ;
}
else // 24 bit (assume no compression)
{
// handle
handle = GlobalAlloc (GMEM_ZEROINIT|GMEM_MOVEABLE,
                      bih.biSize+bmp_size) ;
if (!handle)
{
    MessageBox (NULL,
                "handle problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    return ;
}
bmp_info = (LPBITMAPINFO)GlobalLock (handle) ;
bmp_info->bmiHeader = bih ;

// get bits
bits = (BYTE FAR*)bmp_info + (int)bih.biSize ;
_llseek (hfile, bfh.bfOffBits, 0) ;
if (_hread(hfile, bits, bmp_size) != bmp_size)
{
    MessageBox (NULL,
                "reading bits problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    _lclose (hfile) ;
    return ;
}
// close file
_lclose (hfile) ;
}
} // DIBitmap()

```

The above constructor makes use of several Windows file-handling functions, declared in WINDOWS.H, which are prefixed with an underscore and differ slightly from those discussed in previous sections. The `_lopen()` function opens an existing file, `filename`, in a specified mode, `openmode`, and sets the current file pointer to the start of the file and has the following signature:

```
HRESULT _lopen (LPCSTR filename, int openmode) ;
```

The return value of `_lopen()` is a file handle (`HRESULT (typedef int)`) if the function is successful, else `HRESULT_ERROR`.

The `_lread()`, `_lwrite()`, `_lclose()`, `_llseek()` and `_hread()` are similar to their C counterparts `fread()`, `fwrite()`, `fclose()` and so on, but accept far pointers and can therefore be used with global memory blocks:

```
UINT _lread (HFILE fh, void huge* buff, UINT buff_size) ;
UINT _lwrite (HFILE fh, const void huge* buff, UINT buff_size) ;
HFILE _lclose (HFILE fh) ;
```

---

```
LONG _llseek (HFILE fh, LONG offset, int origin) ;
long _hread (HFILE fh, void huge* buff, long buff_size) ;
```

Returning now to the `DIBitmap` constructor listed above, the constructor begins by using the `_lopen()` function to open the specified bitmap file for reading. If the file is unsuccessfully opened then the API function `MessageBox()` is called to display an error message, and the constructor returns. A more suitable error-handling mechanism would be to use the C++ exception-handling mechanism, but as a first implementation we shall simply use the `MessageBox()` approach.

If the file is successfully opened it is sequentially read from the beginning of the file, first reading the `BITMAPFILEHEADER` structure, then the `BITMAPINFOHEADER` structure, followed by the colour table (if one is present for the cases of 1, 4 or 8 colour bits per pixel) and finally the array of image bits.

Once the `BITMAPFILEHEADER` structure has been read the `bfType` field is checked for the ‘BM’ signature. The `BITMAPINFOHEADER` `biSize` and `biWidth` fields are then read and checked. After the number of colour bits per pixel, bitmap width and height are set, the size of the bitmap is determined, ensuring that the bitmap row size is evenly divisible by 4.

Reading the remainder of the bitmap file depends on whether or not the bitmap has 24 colour bits per pixel. If the bitmap has either 1, 4 or 8 colour bits per pixel a colour table is present. The size of the colour table is the number of image colours multiplied by the size of a single `RGBQUAD` structure. Having calculated the size of the colour table a global memory block is allocated which is sufficient in size to accommodate the `BITMAPINFOHEADER` structure, the colour table and the array of image bits. A similar procedure is followed for the 24-bit case except that a colour table is not present.

The four-argument `DIBitmap` constructor allows a `DIBitmap` object to be defined which is of a given width, height, number of colour bits per pixel and mode, and has the following definition:

```
DIBitmap::DIBitmap (int w, int h, int bit_count, WORD m)
: handle (NULL), bits (NULL), bmp_info (NULL),
  width (w), height (h), pixel_bits (bit_count), mode (m)
{
    // DIB info header
    BITMAPINFOHEADER bih ;
    bih.biSize          = sizeof (BITMAPINFOHEADER) ;
    bih.biWidth         = width ;
    bih.biHeight        = height ;
    bih.biPlanes        = 1 ;
    bih.biBitCount      = pixel_bits ;
    bih.biCompression   = BI_RGB ;
    bih.biXPelsPerMeter = 0L ;
    bih.biYPelsPerMeter = 0L ;
    bih.biClrUsed       = NumberOfColours () ;
    bih.biClrImportant  = 0L ;

    // 1, 4 or 8 bits and 1 plane
    if (pixel_bits == 1 || pixel_bits == 4 || pixel_bits == 8)
    {
        // remainder of DIB info header
        int bytes_per_row   = ((width*bih.biBitCount+31)/32)*4 ;
        bih.biSizeImage     = bytes_per_row * height ;
    }
}
```

```

// handle
int colour_table_size = (int)NumberOfColours () *
                           sizeof (RGBQUAD) ;
DWORD file_size = bih.biSize + colour_table_size +
                  bih.biSizeImage ;
handle = GlobalAlloc (GMEM_ZEROINIT|GMEM_MOVEABLE,
                      file_size) ;
if (!handle)
{
    MessageBox (NULL,
                "handle problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    return ;
}
bmp_info = (LPBITMAPINFO)GlobalLock (handle) ;
bmp_info->bmiHeader = bih ;

// get system palette
if ((GetDeviceCaps (GetDC (NULL), RASTERCAPS) & RC_PALETTE)
    || (GetSystemPaletteEntries (GetDC (NULL), 0,
                                (UINT)NumberOfColours (),
                                (LPPALETTEENTRY)bmp_info->bmiColors) == 0))
{
    MessageBox (NULL,
                "palette problem", "DIBitmap()", 
                MB_ICONEXCLAMATION|MB_OK) ;
    return ;
}
for (int i=0; i<NumberOfColours(); i++)
    bmp_info->bmiColors[i].rgbReserved = 0 ;

// bits
bits = (BYTE FAR*)bmp_info +
       (int)bmp_info->bmiHeader.biSize +
       colour_table_size ;
}
else if (pixel_bits == 24)
{
    // remainder of DIB info header
    bih.biSizeImage = 3L * width * height ;

    // handle (no colour table for 24 bit image)
    DWORD file_size = bih.biSize + bih.biSizeImage ;
    handle = GlobalAlloc (GMEM_ZEROINIT|GMEM_MOVEABLE,
                          file_size) ;
    if (!handle)
    {
        MessageBox (NULL,
                    "handle problem", "DIBitmap()", 
                    MB_ICONEXCLAMATION|MB_OK) ;
        return ;
    }
}

```

```

        }
        bmp_info = (LPBITMAPINFO)GlobalLock (handle) ;
        bmp_info->bmiHeader = bih ;

        // bits
        bits = (BYTE huge*)bmp_info +
               (int)bmp_info->bmiHeader.biSize ;
    }
else
{
    MessageBox (NULL,
                "DIB should be 1, 4, 8 or 24 bits",
                "DIBitmap()",
                MB_ICONEXCLAMATION|MB_OK) ;
    return ;
}
} // DIBitmap()

```

The above constructor begins by filling in as many fields of the BITMAPINFOHEADER as possible. If the bitmap is a 24-bit image a global memory block is allocated and the far pointer to the BITMAPINFO structure data member, `bmp_info`, and the array of bits, `bits`, are set. In the case of either a 1-, 4- or 8-bit image the colour table has to be defined. The Windows API function `GetSystemPaletteEntries()` is used to retrieve a selection of palette entries from the system palette. The API function `GetDeviceCaps()` is also used to confirm that the device is palette based.

The majority of the `DIBitmap` copy constructor is self-explanatory, but note the use of the Windows API function `hmemcpy()`, which copies from a source buffer to a destination buffer and supports huge memory objects, which were allocated using the `GlobalAlloc()` function. The `DIBitmap` destructor first unlocks the global memory block associated with the handle data member using the `GlobalUnlock()` function, and then proceeds to deallocate the global memory block using the `GlobalFree()` function. The `DIBitmap` copy constructor and destructor definitions are listed below:

```

DIBitmap::DIBitmap (const DIBitmap& dib)
: handle (NULL), bits (NULL), bmp_info (NULL),
width (dib.width), height (dib.height),
pixel_bits (dib.pixel_bits), mode (dib.mode)
{
    handle = GlobalAlloc (GMEM_ZEROINIT|GMEM_MOVEABLE,
                           GlobalSize(dib.handle)) ;
    if (!handle)
        MessageBox (NULL,
                    "handle problem","DIBitmap()", MB_ICONEXCLAMATION|MB_OK) ;

    bmp_info = (LPBITMAPINFO)GlobalLock (handle) ;
    hmemcpy (bmp_info, dib.bmp_info, GlobalSize(dib.handle)) ;

    if (pixel_bits == 1 || pixel_bits == 4 || pixel_bits == 8)
        bits = (BYTE FAR*)bmp_info +
               (int)bmp_info->bmiHeader.biSize +

```

```

        (int)NumberOfColours()*sizeof(RGBQUAD) ;
else // 24 bit
    bits = (BYTE FAR*)bmp_info +
           (int)bmp_info->bmiHeader.biSize ;
}

DIBitmap::~DIBitmap ()
{
if (handle)
{
    GlobalUnlock (handle) ;
    GlobalFree (handle) ;
}
}

```

The *NumberOfColours()* member function simply returns the number of image colours using the global function *ImageColours()*:

```

DWORD DIBitmap::NumberOfColours () const
{
return ImageColours (pixel_bits) ;
}

```

*DIBitmap::Colour()* returns an RGBQUAD colour object corresponding to the specified colour table index:

```

RGBQUAD DIBitmap::Colour (int index) const
{
    RGBQUAD rgbq ;
    if (pixel_bits != 24 &&
         index >= 0 && index < NumberOfColours())
        rgbq = BMPColours()[index] ;
    else
    {
        rgbq.rgbRed = 0 ; rgbq.rgbGreen = 0 ; rgbq.rgbBlue = 0 ;
        rgbq.rgbReserved = 0 ;
    }
    return rgbq ;
}

```

*DIBitmap::HandleToBitmap()* uses the Windows API function *CreateDIBitmap()* to return a handle to a bitmap, HBITMAP, given a handle to a device context:

```

HBITMAP DIBitmap::HandleToBitmap (const HDC& hdc)
{
return CreateDIBitmap (hdc,
                      this->BMPInfoHeader(),
                      CBM_INIT,
                      (const void FAR*)this->Bits(),
                      this->BMPInfo(),
                      this->Mode()) ;
}

```

```
}
```

The signature of the *CreateDIBitmap()* function is:

```
HBITMAP CreateDIBitmap (HDC hdc,
                        BITMAPINFOHEADER FAR* bih,
                        DWORD initialiser,
                        const void FAR* bits,
                        BITMAPINFO FAR* bi,
                        UINT mode) ;
```

The *DisplayBitmap()* member function displays a DIBitmap object to a specified device context, hdc:

```
void DIBitmap::DisplayBitmap (const HDC& hdc, int x0, int y0,
                             int w, int h, WORD display_mode)
{
    if (pixel_bits == 1 || pixel_bits == 4 || pixel_bits == 8)
    {
        // stretch or compress
        if (display_mode == DIBitmap::STRETCH)
        {
            SetStretchBltMode (hdc, COLORONCOLOR) ;
            StretchDIBits (hdc,
                            x0, y0, w, h,
                            0, 0, this->width, this->height,
                            (const void FAR*)this->Bits(),
                            this->BMPIInfo(),
                            this->Mode(),
                            SRCCOPY) ;
        }
        // display actual size
    else
    {
        HDC      hdcMem ;
        POINT   bmp_size, bmp_origin ;

        bmp_origin.x = 0 ; bmp_origin.y = 0 ;

        // create memory DC and
        // select bitmap object
        hdcMem = CreateCompatibleDC (hdc) ;
        SelectObject (hdcMem, this->HandleToBitmap(hdc)) ;
        SetMapMode (hdcMem, GetMapMode(hdc)) ;

        bmp_size.x = width ; bmp_size.y = height ;

        // convert device to logical coordinates
        DPtoLP (hdc, &bmp_size, 1) ;
        DPtoLP (hdc, &bmp_origin, 1) ;
```

```

        BitBlt (hdc, x0, y0, bmp_size.x, bmp_size.y,
                 hdcMem, bmp_origin.x, bmp_origin.y,
                 SRCCOPY) ;

        DeleteDC (hdcMem) ;
    }
}

else // 24 bit (display actual size)
{
// row-wise scan
for (int j=0; j<height; j++)
{
    for (int i=0; i<width; i++)
    {
        // note orientation of screen coor.
        SetPixel (hdc, i, (height-j),
                  RGB(bits[3*((DWORD)j*width+i)],
                      bits[3*((DWORD)j*width+i)+1],
                      bits[3*((DWORD)j*width+i)+2]));
    }
}
}

} // DisplayBitmap()

```

*DisplayBitmap()* will display a 24-bit image using the original dimensions of the image or display a 1-, 4- or 8-bit image using either the original dimensions of the image or by stretching the image to fit the display window. In the case of 1-, 4- or 8-bit images the origin of the image can also be translated to a point ( $x_0, y_0$ ) from the window origin. An image is stretched using the Windows API function *StretchDIBits()*:

```

int StretchDIBits (HDC hdc,
                    int xdest, int ydest,
                    int wdest, int hdest,
                    int xsrc, int ysrc, int wsrc, int hsrc,
                    const void FAR* bits,
                    BITMAPINFO FAR* bi,
                    UINT mode,
                    DWORD rop) ;

```

For the 1-, 4- or 8-bit images an image could have been displayed actual size using the API function *SetDIBitsToDevice()*, but an alternative approach has been taken which is based on the *DrawBitmap()* function presented in Petzold (1992, p. 631). First, a memory device context is created using *CreateCompatibleDC()*, and the bitmap image is then selected into the memory device context using *SelectObject()*. The mapping mode is then set to the same as that of the display device context. Both the origin and size of the bitmap coordinates are then converted from device coordinates to logical coordinates and the *BitBlt()* function is called to copy the bitmap from the memory device context to the display device context.

For a 24-bit image it is a simple matter of using the Windows API function *SetPixel()* to set each pixel in the display window to the specified RGB colour. Care is required in ensuring that the RGB components are accessed correctly from the *DIBitmap::bits* array and that

the image is oriented correctly due to the display window origin being centred at the top left corner rather than the bottom left corner. The signature of *SetPixel()* is:

```
COLORREF SetPixel (HDC hdc, int x, int y, COLORREF colour) ;
```

where COLORREF is a DWORD **typedef**.

The definition of the DIBitmap::Write() function is:

```
BOOL DIBitmap::Write (const char* filename)
{
    // open bmp file for writing
    HFILE hfile ;
    if (HFILE_ERROR == (hfile = _lopen(filename, WRITE)))
    {
        if (HFILE_ERROR == (hfile = _lcreat(filename, 0)))
        {
            MessageBox (NULL,
                "DIBitmap::Write()", "can't open/create file",
                MB_ICONEXCLAMATION|MB_OK) ;
            return FALSE ;
        }
    }

    // approx size
    DWORD size = GlobalSize (handle) ;

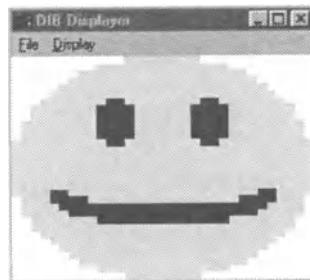
    BITMAPFILEHEADER bfh ;
    bfh.bfType      = 'BM' ;
    bfh.bfSize      = sizeof bfh + size ;
    bfh.bfReserved1 = 0 ;
    bfh.bfReserved2 = 0 ;
    bfh.bfOffBits   = sizeof bfh + (BYTE FAR*)bits -
                    (BYTE FAR*)bmp_info ;

    // write headers and bits to file
    if ((_hwrite(hfile, &bfh, sizeof bfh) != sizeof bfh) ||
        (_hwrite(hfile, (void huge*)bmp_info, size) != size))
        return FALSE ;

    // close file
    _lclose (hfile) ;
    return TRUE ;
} // Write()
```

Note the use of the Windows API *\_hwrite()* function for the writing of huge memory objects (greater than 64 kbyte) to a disk file. These objects were allocated using the API function *GlobalAlloc()*:

```
long _hwrite (HFILE hf, const void huge* buffer,
              long buffer_size) ;
```



**Fig. 17.6** The DIB Displayer main window displaying the SMILE.BMP bitmap.

The `DIBitmap` **class** presented above is tested using a simple Windows DIB file format displayer program called DIB Displayer. A typical example of DIB Displayer displaying, stretched to fit the window, the SMILE.BMP 32×32 16-colour bitmap (which was generated using the Borland Resource Workshop bitmap editor) is shown in Fig. 17.6.

The DIB Displayer program is a minimal Windows program consisting of a single-document interface and two pull-down menus, File and Display. File | Open displays the *File Open* dialog box from the Windows common dialog box library to extract the path and filename of a bitmap image. The one-argument constructor of `DIBitmap` is then used to create a pointer, `bmp`, to a `DIBitmap` object. File | Exit simply closes the DIB Displayer application. Display | Actual.size displays the bitmap actual size while Display | Stretch.to.window stretches the bitmap to the size of DIB Displayer's client area.

The Windows main program function, `WinMain()`, and window procedure function, `WndProc()`, are listed in `DIB_TST.CPP`:

```
// dib_tst.cpp
// a DIB displayer for Windows

#define STRICT // Windows strict conformance

#include <windows.h> // Windows
#include <commdlg.h> // Windows common dialog
#include <stdlib.h> // _MAX_PATH, ...

#include "dib.h" // DIBitmap class
#include "min_max.h" // Min() & Max() template functions
#include "dib_tst.rh" // resource script header file

long FAR PASCAL _export WndProc (HWND, UINT, UINT, LONG) ;

char szAppName [] = "DIBDisplayer" ;

int PASCAL WinMain (HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpszCmdLine, int nCmdShow)
{
    WNDCLASS wndclass ;
    if (!hPrevInstance)
```

```

{
    wndclass.style      = CS_HREDRAW|CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance ;
    wndclass.hIcon     = LoadIcon (hInstance, szAppName) ;
    wndclass.hCursor   = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH)GetStockObject
                                (WHITE_BRUSH) ;
    wndclass.lpszMenuName = (char*)MAIN_WINDOW_MENU ;
    wndclass.lpszClassName = szAppName ;

    RegisterClass (&wndclass) ;
}

HWND hwnd = CreateWindow (szAppName, "DIB Displayer",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL) ;

ShowWindow (hwnd, nCmdShow) ;
UpdateWindow (hwnd) ;

MSG msg ;
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
} // WinMain()

long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
                                  UINT wParam, LONG lParam)
{
    static char szFileName  [_MAX_PATH] ,
                szTitleName [_MAX_FNAME+_MAX_EXT] ;
    static char* szFilter[] = { "Bitmap files (*.bmp)" ,
                                "*.bmp" } ;

    static DIBitmap* bmp (NULL) ;
    static OPENFILENAME ofn ;
    static short      cxClient, cyClient ;
    static WORD       wDisplay = CM_DISPLAYACTUAL ;
    HDC             hdc ;
    HMENU           hMenu ;
    PAINTSTRUCT     ps ;

    switch (message)
    {

```

```
case WM_CREATE:
    ofn.lStructSize      = sizeof (OPENFILENAME) ;
    ofn.hwndOwner        = hwnd ;
    ofn.lpstrFilter       = szFilter[0] ;
    ofn.lpstrFile         = szFileName ;
    ofn.nMaxFile          = _MAX_PATH ;
    ofn.lpstrFileTitle    = szTitleName ;
    ofn.nMaxFileTitle     = _MAX_FNAME + _MAX_EXT ;
    ofn.lpstrDefExt       = "bmp" ;
    return 0 ;

case WM_SIZE:
    cxClient = LOWORD (lParam) ;
    cyClient = HIWORD (lParam) ;
    return 0 ;

case WM_COMMAND:
    hMenu = GetMenu (hwnd) ;

    switch (wParam)
    {
        case CM_FILEOPEN:
            if (GetOpenFileName (&ofn))
            {
                if (bmp)
                    delete bmp ;
                bmp = new DIBitmap
                    (szFileName) ;

                if (bmp == NULL)
                    MessageBox (hwnd,
                               szAppName,
                               "can't create DIB object",
                               MB_ICONEXCLAMATION|MB_OK) ;
                InvalidateRect (hwnd, NULL, TRUE) ;
            }
            return 0 ;

        case CM_DISPLAYACTUAL:
        case CM_DISPLAYSTRETCH:
            CheckMenuItem (hMenu, wDisplay,
                           MF_UNCHECKED) ;
            wDisplay = wParam ;
            CheckMenuItem (hMenu, wDisplay,
                           MF_CHECKED) ;
            InvalidateRect (hwnd, NULL, TRUE) ;
            return 0 ;

        case CM_FILEEXIT:
            SendMessage (hwnd, WM_CLOSE, 0, 0L) ;
            return 0 ;
    }
}
```

```

        }

break ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    // display dib
    if (bmp)
    {
        if (wDisplay == CM_DISPLAYACTUAL)
            bmp->DisplayBitmap (hdc) ;
        else
            bmp->DisplayBitmap (hdc,
                0, 0,
                cxClient, cyClient,
                DIBitmap::STRETCH) ;
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    if (bmp)
        delete bmp ;

    PostQuitMessage (0) ;
    return 0 ;
}

return DefWindowProc (hwnd, message, wParam, lParam) ;
} // WndProc()

```

DIB Displayer's main menu is implemented in the resource script file DIB\_TST.RC, with identifiers listed in DIB\_TST.RH:

```

// dib_tst.rc
// resource script file

#include "dib_tst.rh"

DIBDisplayer ICON "dib_tst.ico"

MAIN_WINDOW_MENU MENU
{
    POPUP "&File"
    {
        MENUITEM "&Open...", CM_FILEOPEN
        MENUITEM SEPARATOR
        MENUITEM "E&xit", CM_FILEEXIT
    }
    POPUP "&Display"
    {
        MENUITEM "&Actual size", CM_DISPLAYACTUAL, CHECKED
        MENUITEM "Stretch to window", CM_DISPLAYSTRETCH
    }
}

```

```

        }
    }

// dib_tst.rh
// resource script header file

#define MAIN_WINDOW_MENU 101

#define CM_FILEOPEN          102
#define CM_FILEEXIT          103
#define CM_DISPLAYSTRETCH    104
#define CM_DISPLAYACTUAL     105

```

The Windows module definition file DIB\_TST.DEF is:

```

; DIB_TST.DEF
; module definition file

NAME           DIB_TST

DESCRIPTION    'DIB Displayer for Windows by G. M. Seed,
1996'
EXETYPE       WINDOWS
STUB           'C:\BC5\BIN\WINSTUB.EXE'
CODE           PRELOAD MOVEABLE DISCARDABLE
DATA           PRELOAD MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      8192

```

It is beyond the scope of this book to describe in detail the implementation of DIB\_TST.CPP, DIB\_TST.RC and DIB\_TST.DEF, but the interested reader is referred to Petzold (1992).

## 17.16 Summary

You may have gathered from this chapter that there are a great deal of C++ stream classes and associated member functions and C stream functions. The C and C++ stream libraries are complex, and it does take time before you will feel reasonably comfortable with them. However, the C++ and C stream libraries are very powerful, and because of the importance and necessity attached to input and output it is well worth spending some time and effort in developing a good understanding of the libraries.

Of the many classes in the complex C++ standard stream **class** library, the most frequently used are **istream**, **ostream** and **iostream** and the file stream classes **ifstream**, **ofstream** and **fstream**.

C++ predefines four open streams for use with the standard input and output streams, namely **cin**, **cout**, **cerr** and **clog**. Stream input and output can be formatted either by using manipulators or member functions of **class ios**. Manipulators can be either parametrised or non-parametrised and can be overloaded and defined for user-defined classes.

The insertion and extraction operators are overloaded for all of the C++ integral data types and can be overloaded in a device-independent manner for user-defined classes. Provided the

insertion and extraction operators are overloaded in a device-independent manner there is no difference between the manipulation of standard stream objects and file stream objects.

The C++ standard stream library provides a programmer with excellent support for the handling of file streams.

This chapter discussed a **class** called `ReadFile` for reading a collection of objects from an object data file. The `ReadFile` **class** illustrated just how powerful an object-oriented approach to file handling can be.

We have seen the use of redirection and command line arguments. Redirection allows the standard input and output streams to be redirected to an alternative device. Command line arguments allow a programmer to pass information to a program before the program commences execution.

Although not object-oriented, C's approach to the handling of streams is nevertheless extremely powerful and popular, and this chapter has presented an overview of the C file-handling mechanism.

The chapter concluded with a discussion of a Windows device-independent bitmap file format **class** called `DIBitmap`. The `DIBitmap` **class** and its associated member functions greatly assists the handling of bitmap images stored in a DIB file format. The `DIBitmap` **class** will prove useful in Chapter 20 for the manipulation of raytraced images.

## Exercises

17.1 Write a program that displays the first 20 lines of a user-specified file.

17.2 For the following `Vector` **class**:

```
class Vector
{
private:
    double a, b, c ;
public:
    Vector (double _a, double _b, double _c)
        : a (_a), b (_b), c (_c) {}
    friend ostream& operator << (ostream& s,
                                    const Vector& v) ;
};
```

define a one-argument parametrised manipulator called `VectorPrecision` using **class** `omanip` so that a `Vector` object can be output to a specified precision:

```
Vector v ;
//...
cout << "v: " << VectorPrecision (3) << v << endl ;
```

17.3 Write a program that reads an arbitrary number of integers from a file that are in the following format:

```
6
5
4
2
4
1
```

and subsequently displays the numbers to the display screen.

- 17.4 The set of numbers of Exercise 17.3 is assumed to represent the numbers that turned up on a die. Write a program which reads the data file of Exercise 17.3 and then displays the mean and standard deviation of the numbers on the display screen. The mean,  $\bar{x}$ , and variance,  $s^2$ , of a sample  $x_1, x_2, \dots, x_n$  are defined as:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i, \quad s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

The square root of the variance is the standard deviation.

- 17.5 Write a program which uses command line arguments to display the current time or date, with both the time and date displayed by default. Use the C++ `time()` and `ctime()` library functions:

```
time_t time (time_t* timeptr) ;
char* ctime (const time_t* timeptr) ;
```

both declared in TIME.H. The `ctime()` function returns a pointer to a string which contains 26 characters of the following form:

Wed Feb 28 22:40:15 1996\n\0

assuming a 24-hour clock.

- 17.6 Write the following to a file called 17\_6.DAT using the C `fprintf()` function:

```
5
15.15
stream
```

- 17.7 Rewrite Exercise 17.3 using C functions.

# The Preprocessor

*This chapter describes the preprocessor directives. Although not strictly part of the C++ language, the preprocessor directives, particularly the #include directive, are frequently used when programming in C++.*

*The preprocessor is increasingly becoming redundant because of equivalent C++ language-based features. The const keyword allows constant identifiers to be defined. Macros are replaced by inline functions, with the template feature defining and declaring type-independent functions and classes. We shall see in the next chapter that the namespace feature associates a name to a given scope. All of const, inline, template and namespace are statically type-checked, whereas the preprocessor bypasses the static type checking mechanism of C++.*



## 18.1 The Preprocessor

This chapter examines the preprocessor, which C++ inherited from the C language. C++ programmers do not like the preprocessor. In fact, use of the preprocessor is strongly discouraged by the original designer of C++ (Stroustrup, 1994). The key problems with the preprocessor are that it is not part of the C++ programming language, it is not object-oriented, it is unstructured and it bypasses the strong type-checking mechanism of C++.

The compilation of a program generally involves several different phases or passes. The first pass on a program is the preprocessor pass. The second pass is generally a syntax and semantic pass, which is followed by a token and a code optimisation pass. This chapter is concerned with the first pass and the preprocessor's associated directives.

## 18.2 Preprocessor Directives

To date, we have seen various preprocessor directives used, such as #include, #define, #ifndef and #endif. A complete list of the preprocessor directives is given in Table 18.1. Each preprocessor directive is prefixed by the number sign, #, and must be placed on its own

**Table 18.1** Preprocessor directives.

<i>Preprocessor directive</i>	<i>Meaning</i>
#	Null directive
#define	Defines a symbolic constant or macro
#error	Stop compilation and generate an error message
#include	Include files in source code
#line	Line numbers
#pragma	Implementation-specific directive
#undef	Undefine previously defined macro conditional directives
Conditional directives	
#elif	Else if
#else	Else
#endif	End of if
#if	Conditional if
#ifdef	If defined
#ifndef	If not defined

line. Preprocessor directives can be placed anywhere within a program but are usually placed at the start of a program.

We shall examine each of the preprocessor directives in more detail in the following subsections.

### 18.2.1 The # Null Directive

The operator # on a single line is the null directive and is ignored by the preprocessor.

### 18.2.2 The #define Directive

The #define preprocessor directive defines a symbolic constant. For example, the following #define has been extracted from the C++ standard header file MATH.H:

```
#define M_PI 3.14159265358979323846 // MATH.H
```

and associates the symbolic constant M\_PI with  $\pi$ . Note that no semicolon is specified to terminate a preprocessor directive and that no assignment operator is used to assign the approximate value of  $\pi$  to the #define M\_PI. It was mentioned above that a preprocessor directive must be placed on its own line, and consequently a directive terminates when a new line is encountered.

#define can also be used to define a parametrised macro:

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
```

which defines a parametrised macro, MIN(), that determines whether a is less than b. Note that white space between the macro name and the open parenthesis '(' is not allowed.

The general syntax of #define is:

```
#define MacroName <token_sequence>
```

Each occurrence of MacroName in a program will be identically replaced by the optional (between < and >) body of the parametrised macro token\_sequence. Such a process of

replacement is generally referred to as *macro replacement* or *macro expansion*. Because, by definition, a preprocessor directive must be placed on its own line, the backslash character (\) is used for token sequences that extend onto a new line:

```
#define MEM_ERROR "Insufficient memory\
available"
```

The `token_sequence` can be null:

```
#define EMPTY
```

Following a suitable macro definition the macro name can be used to define subsequent macros or form part of expressions:

```
#define PI_OVER2 M_PI / 2
```

You may have noticed an excessive amount of parentheses in the `#define MIN()` macro above. To illustrate why each macro argument is enclosed in parentheses, consider a macro called `SQUARE()` which simply multiplies `x` by itself:

```
#define SQUARE(x) ((x) * (x))
```

Consider using `SQUARE()` to find the square of an expression:

```
int sq = SQUARE (1+1) ;
```

This statement will be expanded by the preprocessor to:

```
int sq = ((1+1) * (1+1)) ;
```

which will, as expected, assign the value of four to `sq`.

Consider now the case when the parentheses are neglected from the definition of `SQUARE()`:

```
#define SQUARE(x) x * x
```

Now, `SQUARE(1+1)` will be expanded as:

```
int sq = 1 + 1 * 1 + 1 ;
```

which assigns the value of three, and not four, to `sq` because the `*` operator has a higher precedence than the `+` operator. Therefore, always remember to use parentheses in the definition of parametrised macros.

It is important to note that in the above `#defines` the type of `M_PI` or `a` and `b` in `MIN()` was not declared. You may be thinking that defining a type-independent `#define` is great, but in fact it is `#define`'s greatest weakness because it bypasses the type-checking mechanism of the compiler – which means that you're on your own when it comes to weird and wonderful compilation and run-time errors.

Since a macro is expanded inline there is no associated function call overhead. Thus an increase in performance can be experienced by using a macro instead of a function call. Remember though, as with `inline` functions, that because a macro is expanded inline any

increase in performance can quickly be lost if the macro is large in size or is called a large number of times. Also, if a macro is called a large number of times the resultant size of the program's executable file will be significantly greater than if a function call had been used instead.

The following program illustrates the various properties of `#define` discussed above:

```
// define.cpp
// illustrates the #define preprocessor directive
#include <iostream.h> // C++ I/O
#include <math.h> // M_PI

// common errors: '=' and ';'
#define GEGA = 1e09
#define GMS "Geometric Modelling Society" ;

#
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define EMPTY
#define PI_OVER2 M_PI / 2
#define SQUARE_WITHOUT(x) x * x
#define SQUARE(x) ((x) * (x))
#define CUBE(x) (SQUARE (x) *\n        (x))
void main ()
{
    cout << "MIN(1,2): " << MIN (1, 2) << endl
    << "MIN(1.1,2.2): " << MIN (1.1, 2.2) << endl
    << "MIN('a','b'): " << MIN ('a', 'b') << endl
    << "M_PI: " << M_PI << endl
    << "PI_OVER2: " << PI_OVER2 << endl
    << "SQUARE_WITHOUT(1+1): "
                    << SQUARE_WITHOUT (1+1)
                    << endl
    << "SQUARE(1+1): " << SQUARE (1+1) << endl
    << "CUBE(2): " << CUBE (2) << endl ;
}
```

with output:

```
MIN(1,2): 1
MIN(1.1,2.2): 1.1
MIN('a','b'): a
M_PI: 3.14159
PI_OVER2: 1.5708
SQUARE_WITHOUT(1+1): 3
SQUARE(1+1): 4
CUBE(2): 8
```

C++ provides a language-based alternative for each of `#define`'s main uses. C++ offers the keyword **const** for constant identifiers, the keyword **inline** for functions expanded inline and the keyword **template** for parametrised function classes or types. Also, the C++

**namespace** feature discussed in the next chapter allows a name to be associated with a given scope. Such language-based features as **const**, **inline**, **template** and **namespace** make `#define` redundant:

```
// cpp_ver.cpp
// a C++ version of DEFINE.CPP
#include <iostream.h> // C++ I/O
#include <math.h> // M_PI

template <class T>
inline T Min (T a, T b)
{
    return a < b ? a : b ;
}

const double PI_OVER2 = M_PI / 2 ;

void main ()
{
    cout << "Min(1,2): " << Min (1, 2) << endl
        << "Min(1.1,2.2): " << Min (1.1, 2.2) << endl
        << "Min('a','b'): " << Min ('a', 'b') << endl
        << "PI_OVER2: " << PI_OVER2 << endl ;
}
```

with output:

```
Min(1,2): 1
Min(1.1,2.2): 1.1
Min('a','b'): a
PI_OVER2: 1.5708
```

### 18.2.3 The `#error` Directive

The `#error` directive stops compilation and displays `error_message`:

```
#error error_message
```

Note that `error_message` is not enclosed within double quotation marks. The program below illustrates the `#error` directive:

```
// error.cpp
// illustrates the #error preprocessor directive

#error preprocessor error message

void main ()
{
}
```

### 18.2.4 The #include Directive

The #include directive has the following general syntax:

```
#include <file_name>
#include "file_name"
#include macro_identifier
```

The first two versions of #include replace the line containing the # include directive with the contents of the specified file file\_name between either the angle brackets (<>) or the double quotation marks (""). The file file\_name is generally a header file with the filename extension .H or .HPP, but the extension name is optional. The file file\_name is allowed to contain further #include directives. If file\_name is enclosed by angle brackets then file\_name refers to a standard include file and is searched for in the include directories of the compiler. Alternatively, if file\_name is enclosed between double quotation marks then file\_name generally refers to a user-defined file, with file searching commencing in the current working directory. If a file enclosed within double quotation marks is not found in the current working directory then the file search algorithm reverts to that of a file enclosed within angle brackets. A standard include file can also be placed between double quotation marks. If file\_name contains an explicit directory path then only that directory will be searched.

The macro version of #include expands macro\_identifier to a filename of either the <> or "" formats. Therefore, neither of the < or " characters is allowed as the first character of macro\_identifier.

The following program illustrates the three versions of the #include directive:

```
// inc.cpp
// illustrates the #include preprocessor directive

#define IOSTREAM "c:\bc5\include\iostream.h"

#include <math.h>
#include "inc.h"
#include IOSTREAM

void main ()
{
}
```

### 18.2.5 The #line Directive

The #line directive allows you to specify the next line number, line\_number, and filename, file\_name, of the source file and has the following general syntax:

```
#line line_number <"file_name">
```

where line\_number is a positive integer. The filename token is optional, and if neglected is assumed to be the last filename specified in the #line directive. If no previous #line directive exists then the filename is assumed to be the source filename. Consider the program:

```
// line.cpp
```

```
// illustrates the #line preprocessor directive
#include <iostream.h> // C++ I/O

void main ()
{
    #line 1
    cout << "line: " << __LINE__
          << "; file: " << __FILE__ << endl ;
    #line 9 "file.cpp"
    cout << "line: " << __LINE__
          << "; file: " << __FILE__ << endl ;
}
```

with output:

```
line: 1; file: LINE.CPP
line: 9; file: file.cpp
```

The `__LINE__` and `__FILE__` predefined macros contain the current line number and filename of the source file and will be described in more detail later.

The `#line` directive is generally used for program debugging, error reporting and line number and filename referencing of code extracted from files other than the source file.

### 18.2.6 The `#pragma Directive`

The `#pragma` directive allows implementation-specific directives to be defined and has the following syntax:

```
#pragma implementation_specific_name
```

Since the exact details of `#pragma` vary from compiler to compiler it is recommended that you consult your compiler's documentation. For instance, the Borland C++ compiler (version 5.0) defines 12 `#pragmas`, four of which are `argsused`, `hdrfile`, `startup` and `exit`. The `#pragma argsused` disables the warning message 'Parameter name is never used in function `FunctionName`'. The `#pragma hdrfile "filename.csm"` sets the name of the precompiled header file to `FILE_NAME.CSM`. The `startup` and `exit` `#pragma`'s allow a programmer to specify functions to be called upon program startup and exit.

### 18.2.7 The `#undef Directive`

The `#undef` directive allows you to undefine a previously defined macro, and has the syntax:

```
#undef macro_identifier
```

`#undef` is illustrated in the following program:

```
// undef.cpp
// illustrates the #undef preprocessor directive
#include <iostream.h> // C++ I/O
```

```
#define BUF_SIZ 80 // define directive

void main ()
{
    cout << "BUF_SIZ: " << BUF_SIZ << endl ;
    #undef BUF_SIZ           // undefine directive
    #define BUF_SIZ 120      // redefine directive
    cout << "BUF_SIZ: " << BUF_SIZ << endl ;
}
```

with output:

```
BUF_SIZ: 80
BUF_SIZ: 120
```

### 18.2.8 The #elif, #else, #endif, #if, #ifdef and #ifndef Conditional Directives

The #if, #elif, #else and #endif conditional directives operate in a similar manner to the conditional operators:

```
#if test_expression
<statement_block>
<#elif test_expression statement_block>
<#elif test_expression statement_block>
//...
<#elif test_expression statement_block>
<#else <newline> statement_block>
#endif
```

If the #if test\_expression evaluates to logical-true the corresponding statement\_block is preprocessed, else if the #if test\_expression evaluates to logical-false the #if statement\_block is ignored. If the #if test\_expression evaluates to logical-true, then after the corresponding statement\_block is processed control is directed to the #endif directive, else if the #if test\_expression evaluated to logical-false control is directed to the next optional #elif directive. If the #elif test\_expression evaluates to logical-true its corresponding statement\_block is preprocessed, else control is directed to the next optional #elif directive. This process is repeated until either an #elif or #else directive is encountered. An #else directive is an alternative condition if all previous #if and #elif directives evaluated to logical-false. Note that for each #if there must a corresponding #endif.

The following program illustrates the #if, #elif, #else and #endif directives for controlling compilation:

```
// if.cpp
// illustrates the #if, #elif, #else and #endif
// preprocessor directives
#include <iostream.h> // C++ I/O

#define STRING_TYPE 1
```

---

```
#if STRING_TYPE == 0
#include <string.h>      // C string
#elif STRING_TYPE == 1
#include <cstring.h>     // C++ string
#else
#include "my_str.h"       // user-defined string
#endif

void main ()
{
    string str ("hi!") ;
}
```

In addition to the above conditional preprocessor directives there is the `defined` operator which has the following general syntax:

`defined(identifier)`

or

`defined identifier`

The parentheses are optional and `defined` can be preceded by the logical negation operator (`!`). For example:

```
#if !defined(_BOOL_H)
#include "bool.h"
#endif
```

The main advantage of the `defined` operator is that it can be used in conditional expressions:

```
#if defined(BOOL) || !defined(_BOOL_H) &&
!defined(_BOOLEAN_H)
//...
```

The `#ifdef` and `#ifndef` directives have the following general syntax:

```
#ifdef identifier
<statement_block>
#endif

#ifndef identifier
<statement_block>
#endif
```

and allow you to test whether `identifier` is defined or not defined, respectively. Both `#ifdef` and `#ifndef` can use `#else`, but not `#elif`. An illustration of `#ifndef` is given in the program below:

```
// ifndef.cpp
```

```
// illustrates the #ifndef preprocessor directive

#ifndef BOOL
#define int BOOL ;
#define FALSE 0
#define TRUE 1
#endif

void main ()
{
    BOOL bool (TRUE) ;
}
```

### 18.3 The # and ## Operators

The # and ## preprocessor operators are used in conjunction with the #define directive. The # operator converts the macro argument that it operates on to a string enclosed by double quotation marks. The ## operator concatenates two tokens. The following program illustrates # and ##:

```
// #.cpp
// illustrates the # and ## preprocessor operators
#include <iostream.h> // C++ I/O

#define STRINGISE(s) #s
#define CONCATENATE(x, y) x ## y

void main ()
{
    cout << STRINGISE (gymnopedies) << endl
        << CONCATENATE ("inter", "national") << endl ;
}
```

with output:

```
gymnopedies
international
```

### 18.4 Predefined Macros

C++ predefines the six implementation-independent macros shown in Table 18.2. Refer to your compiler's documentation regarding additional compiler-dependent predefined macros.

The \_\_cplusplus macro is defined as equal to 1 if the program being compiled is a C++ program, else \_\_cplusplus is undefined. Note that \_\_cplusplus does not end with two underscore characters and contains entirely lowercase alphanumeric characters, unlike the other five predefined macros. With a quick browse through the C++ standard library header

**Table 18.2** Predefined macros.

<i>Predefined macro name</i>	<i>Meaning</i>
<code>__cplusplus</code>	C++ program
<code>__DATE__</code>	Calendar data ( <i>mmm dd yyyy</i> )
<code>__FILE__</code>	Source code filename
<code>__LINE__</code>	Current line number
<code>__STDC__</code>	ANSI-C compliant
<code>__TIME__</code>	Compilation time ( <i>hh:mm:ss</i> )

files you will find `__cplusplus` used as the test expression of an `#ifdef` conditional directive to ensure that a C++ header file is included in a source file with a C++ extension. The following example has been extracted from the C++ string `class` header file `CSTRING.H`:

```
// cstring.h
//...
#ifndef __cplusplus
#error Must be a C++ source file to include cstring.h
#endif
//...
```

The `__DATE__` macro is a string literal that contains the calendar date on which the preprocessor commenced processing the source file. The format of the date string is *mmm dd yyyy*, where *mmm* represents the month (e.g. Jan), *dd* represents the day (e.g. 16) and *yyyy* represents the year (e.g. 1996).

The `__FILE__` macro is a string literal that contains the name of the source file being processed. This macro is updated by the `#include` and `#line` directives.

The `__LINE__` macro is an integer number which represents the current line number being processed. This macro is updated by the `#line` directive.

The `__STDC__` macro is defined as equal to 1 if a compiler is ANSI C-compliant, else `__STDC__` is undefined.

Finally, the `__TIME__` macro is a string literal that contains the time at which the preprocessor commenced processing the source file. The format of the time string is *hh:mm:ss*, where *hh* represents the hour (e.g. 22), *mm* represents the minutes (e.g. 58) and *ss* represents the seconds (e.g. 58).

The following program illustrates the predefined macros:

```
// predef.cpp
// illustrates the predefined macros
#include <iostream.h> // C++ I/O

void main ()
{
    cout << " __cplusplus: " << __cplusplus << endl
    << " __DATE__: " << __DATE__ << endl
    << " __FILE__: " << __FILE__ << endl
    << " __LINE__: " << __LINE__ << endl ;

    #ifdef __STDC__
```

```

    cout << "__STDC__: "      << __STDC__      << endl ;
#endif

cout << "__TIME__: "      << __TIME__      << endl ;
}

```

which generated the following output when I ran the program:

```

__cplusplus: 1
__DATE__: Jun 16 1995
__FILE__: PREDEF.CPP
__LINE__: 10
__TIME__: 16:23:35

```

Let's conclude by mentioning the ever popular *assert()* macro defined in the ASSERT.H header file:

```
assert (expression)
```

If expression evaluates to logical-false a message is directed to stderr (generally the display screen) and program execution is aborted by calling the *abort()* function. The form of the message displayed is:

```
Assertion failed: expression, file file_name, line line_number
```

line\_number is the line number in which the *assert()* macro appears in the source file file\_name.

If the directive `#define NDEBUG` is placed in the source file before the `#include <assert.h>` directive then the *assert()* macro is effectively turned off.

The following program illustrates *assert()*:

```

// assert.cpp
// illustrates the assert() macro
#include <iostream.h> // C++ I/O

//#define NDEBUG

#include <assert.h> // assert()

void main ()
{
#define FALSE 0

    assert (FALSE) ;
}

```

with output:

```

Assertion failed: FALSE, file ASSERT.CPP, line 13
Program aborted

```

## 18.5 Summary

The preprocessor allows a programmer to control the compilation process by operating on directives. A number of implementation-independent directives are supplied by the preprocessor, such as `#define` and `#include` and a compiler-dependent directive called `#pragma`. The `#define` directive can be used to define identifiers and parametrised macros and several predefined macros are available to a programmer.

Since its earliest days C++ has gradually introduced language-based features, such as `const`, `inline`, `template` and `namespace` that have made the preprocessor redundant. Admittedly there are still facilities offered by the preprocessor for which C++ offers no language-based equivalent, but whenever you have the option of choosing between a preprocessor directive and a language-based feature then choose the latter.

## Exercises

- 18.1 Develop two macros and `inline` functions which return a Boolean value depending on whether a given number is zero or infinite (which are to be defined as `1e-20` and `1e+20`, respectively).
- 18.2 Use the `#ifdef`, `#else`, `#endif` and `#include` directives and the `__cplusplus` predefined macro to include either the `IOSTREAM.H` or `STDIO.H` header files, depending on whether a user is compiling either a C++ or C program.
- 18.3 It is desired that the `WINDOWS.H` header file is included in a program that declares the following Boolean type with the `FALSE` and `TRUE` enumerators:

```
#include <windows.h> // Windows header file
//...
enum Boolean { FALSE, TRUE };
```

Unfortunately `WINDOWS.H` `#defines` `FALSE` and `TRUE`:

```
#define FALSE 0
#define TRUE 1
```

so that name clashes are experienced. Use the `#ifdef`, `#undef` and `#endif` conditional directives so that the `WINDOWS.H` header file and the Boolean `enum` can both define the `FALSE` and `TRUE` literals.

- 18.4 The function argument `i` is not used in the definition of `Function()`:

```
void Function (int i)
{
}

void main ()
{
    int i (0) ;

    Function (i) ;
}
```

Use the `#pragma` directive to eliminate the compilation warning ‘Parameter `i` is never used in function `Function()`’.

- 18.5 A header file, POINT.H, for a **class** Point is of the form:

```
// point.h
// class Point
class Point
{
//...
};
```

Modify the header file so that only C++ source files can include the POINT.H header file.

- 18.6 Develop an *Assert()* function which mimics the C++ standard library macro *assert()*.

- 18.7 Compare the use of the C++ standard library macro *assert()* declared in ASSERT.H with the *Assert()* **template** function defined in Chapter 14:

```
template <class T, class X>
inline void Assert (T exp, X x)
{
if (!NDEBUG) // compatible with assert() macro NDEBUG
if (!exp)
    throw x ;
}
```

# Namespaces

A namespace declaration defines a scope, and can therefore be used to prevent name clashes. The namespace feature is required because C++ defines a single global scope namespace outside the scope of if-else or switch statement blocks, loops, functions and classes. As programs steadily grow in size they correspondingly become more susceptible to name clashes. A large program will consist of numerous function and class library files, and namespaces allow functions and classes to be associated with a scope other than the global file scope. The namespace feature is particularly useful for eliminating name clashes between applications and function and class libraries.

A namespace name and the scope resolution operator are used to qualify access explicitly. A using declaration allows access to a namespace member without explicit qualification whereas a using directive allows access to all members of a namespace without explicit qualification.



## 19.1 Namespaces – What are They and Why do we Need Them?

The namespace is one of the newest major features to be added to the C++ programming language. C++ provides a namespace for **if-else** and **switch** statement blocks, loops, function declarations and definitions, **unions**, **structs**, **classes** or translational units<sup>1</sup>. Outside these namespaces C++ provides one single global namespace which can result in name clashes when multiple files are linked together. For example, consider including the WIN-DOWS.H library header file in the following program:

```
// n_clash.cpp
// illustrates name clashes

#include <windows.h> // Windows header file
```

<sup>1</sup> A translational unit in C++ is a file and all included files corresponding to #include directives.

```
class Polygon
{
//...
};

void main ()
{
    Polygon poly ;
}
```

This is straightforward enough, but this program will generate a number of confusing compilation errors because of the name clash between the **class** `Polygon` declaration in `N_CLASH.CPP` and the `Polygon` function declaration in `WINDOWS.H`:

```
// windows.h
#ifndef _WINDOWS_H // prevent multiple includes
#define _WINDOWS_H
//...
BOOL Polygon (HDC, const POINT FAR*, int) ;
//...
#endif // _WINDOWS_H
```

In addition, note the trouble that we have to go to prevent multiple definitions and `#includes` using a preprocessor macro guard in `WINDOWS.H`.

One solution to this problem, which you may have seen used in previous chapters, is simply to rename any conflicting names:

```
// n_clash1.cpp
// illustrates name clashes

#include <windows.h> // Windows header file

class PlanarPolygon
{
//...
};

void main ()
{
    PlanarPolygon poly ;
}
```

Renaming is a tedious solution and not always possible, since you may be working with shipped libraries.

## 19.2 The **namespace** Declaration

The general form of the **namespace** declaration is:

---

```
namespace Name
{
    // declarations & definitions
}
```

Note that there is no semicolon directly after the closing brace of the **namespace** declaration. Declarations within the braces of **namespace** Name do not clash with those declared in another **namespace** or global names because each **namespace** defines its own unique scope. For example, the following namespaces, X and Y, and global names are completely separate:

```
// nspace.cpp
// illustrates the namespace declaration

void Function () ; // global

namespace X
{
    void Function () ;

    void Gunction ()
    {
        //...
        Function () ; // X::Function()
    }
    //...
    class Point
    { /*...*/ };
}

namespace Y
{
    void Function () ;
    //...
    class Point
    { /*...*/ };
}
//...
void main ()
{
    X::Function () ;
    X::Point p ;

    Y::Function () ;
    Y::Point q ;
}
```

The name of a **namespace** and the scope resolution operator (:) are used to resolve access to a given **namespace**'s members explicitly. As with classes, explicit use of the scope resolution operator is not necessary for names enclosed within a **namespace**:

```
namespace X
```

```

{
void Function () ;
//...
void Gunction ()
{
//...
Function () ; // X::Function ()
}
}

```

A **namespace** is said to be *open*, whereas classes are not:

```

namespace X
{
void Function () ;
//...
}
//...
namespace X
{
void Function1 () ;
//...
}

```

which inherently leaves a **namespace** open for namespace clashing. Thus, when choosing a name for a **namespace** choose appropriately descriptive names and not simply X as shown above! Namespaces have been designed to be open so as to allow a programmer to place program code in several different, identically named, **namespace** declarations rather than one single **namespace** declaration. This is similar in principle to having multiple access specifiers within a **class** declaration.

It is possible to use an alias for a namespace name and is useful when referring to long namespace names:

```

namespace Library_Mathematical
{
}
//...
namespace Math = Library_Mathematical ;

```

Namespaces can be nested:

```

// ns_nest.cpp
// illustrates nested namespaces

```

```

namespace X
{
void Function () ;
//...
namespace Y
{
void Function () ;
}

```

---

```
//...
class Point
{
//...
public:
//...
double Z () ;
//...
};

}

}

//...
```

with the usual use of the scope resolution operator for qualification:

```
void X::Y::Function ()
{
//...
}

//...
double X::Y::Point::Z ()
{
//...
}
```

Let's briefly return to our original problem of a name clash between **class** Polygon and the Windows API *Polygon()* function. By using namespaces the name clash is easily resolved:

```
// no_clash.cpp
// modified N_CLASH.CPP using namespaces

#include <windows.h> // Windows header file
#include "polygon.h" // class Polygon
#include "sphere.h" // class Sphere

void main ()
{
Shapes::Polygon poly ;
Shapes::Sphere sphere ;
//...
}
```

With POLYGON.H and SPHERE.H:

```
// polygon.h
// class Polygon

namespace Shapes
{
class Polygon
```

```

{
//...
};

}

// sphere.h
// class Sphere

namespace Shapes
{
class Sphere
{
//...
};

}

```

In addition the declaration of **class** Polygon has been placed in the header file POLYGON.H within the scope of Shapes. Advantage is also taken of the property that namespaces are, by definition, *open* by also placing **class** declaration Sphere within Shapes in SPHERE.H

### 19.3 The **using** Declaration

The *using declaration* allows a member from a **namespace** to be accessed without explicit qualification:

```

// use_dec.cpp
// illustrates the using-declaration

namespace X
{
    int data = 0 ;
    void Function () ;
    //...
}
//...
void main ()
{
    using X::Function ; // using-declaration

    data = 1 ; // error: not accessible

    Function () ; // o.k.: X::Function()
}

```

A **using** declaration adds an object to local scope just as a *normal* object declaration:

```

namespace X
{
    int data = 0 ;

```

---

```
//...
}

void Function ()
{
    int data = 0 ;
    using X::data ; // error: multiple definition of data!
//...
}
```

## 19.4 The **using** Directive

The **using** directive allows *all* members of a **namespace** to be accessed without explicit qualification:

```
// use_dir.cpp
// illustrates the using-directive

namespace X
{
    int data = 0 ;
    void Function () ;
    //...
}
//...
void main ()
{
    using namespace X ; // using-directive

    data = 1 ;      // X::data
    Function () ; // X::Function()
}
```

Do note that a **using** directive allows access to all members of a given **namespace**, and if this is not what is intended then use the safer, more explicit, **using declaration** approach.

A **using** directive can easily introduce ambiguities between similarly named members of different namespaces:

```
namespace X
{
    int data = 0 ;
}
//...
namespace Y
{
    int data = 0 ;
}

void Function ()
```

```

{
using namespace X ;
using namespace Y ;
//...
data = 1 ; // error: X::data or Y::data?
}

```

In contrast with a **using** declaration, a **using** directive does not add an object to local scope:

```

namespace X
{
int data = 0 ;
//...
}

void Function ()
{
int data = 0 ;
using namespace X ; // o.k.: non-declaration
//...
}

```

Since a **using** directive does not add to local scope, the above **using** directive within *Function()* does not redefine *data*, but makes *X*::*data* accessible.

## 19.5 The Nameless **namespace** Declaration

A **namespace** can be declared unnamed:

```

namespace
{
//...
}

```

which is a useful feature of the **namespace** declaration when the name of a **namespace** is irrelevant provided that it does not clash with other nameless **namespace** declarations. Thus we observe from the above discussions that the nameless **namespace** is equivalent to:

```

namespace Name
{
//...
}
//...
using namespace Name ;

```

Since **namespace** declarations are open by definition, all nameless **namespace** declarations in the same scope have the same *name*.

## 19.6 Global Scope

On numerous occasions in previous chapters we have seen the use of the global scope qualification `::Function()`, meaning the global namespace declaration of `Function()`:

```
// global.cpp
// namespaces and global scope

void Function () ; // global

namespace X
{
    void Function () ;
    //...
    void Function1 ()
    {
        Function () ; // X::Function ()
        X::Function () ; // X::Function ()
        ::Function () ; // global Function()
    }
}
//...
```

In view of the previous section, note that all nameless **namespace** declarations in the same scope form part of the global scope.

Be careful of name clashes arising as a result of global objects and the **using** directive:

```
int data = 0 ; // global

namespace X
{
    int data = 0 ;
}

void Function1 ()
{
    using namespace X ;

    ::data = 1 ; // o.k.: global data
    X::data = 2 ; // o.k.: X::data
    data = 3 ; // error: global data or X::data?
}
```

## 19.7 Overloading Namespaces

Members of a **namespace** can be overloaded with the aid of either a **using** declaration or directive:

```
// overload.cpp
// overloading namespaces

namespace X
{
    void Function (int i) ;
}

using namespace X ;

namespace Y
{
    void Function (double d) ;
}

using namespace Y ;

void main ()
{
    Function (1.0) ; // call to Y::Function(double)
}
```

## 19.8 Namespaces and Inheritance

Namespaces integrate with inheritance. A **using** declaration that is a member of a derived **class** *must* name a member of a base **class**, while a **using** directive is simply not allowed as a **class** member:

```
// inherit.cpp
// illustrates namespaces and inheritance

class Base
{
    protected:
        int data ;
    public:
        Base () ;
        //...
};

class Derived : public Base
{
    public:
        // o.k.: u-dec & data is a member of Base
        using Base::data ;
        // error: Function() is not a member of Base
        using Base::Function ;
        // error: u-dir not allowed as class member
        using namespace Base ;
```

```
//...
};

//...
```

The **using** declaration offers a new angle on the problem of hiding a member in a base **class** by a similarly named derived **class** member. In the following program code **Derived**::*Function(double)* hides **Base**::*Function(int)*:

```
class Base
{
public:
    void Function (int i) ;
    //...
};

class Derived : public Base
{
public:
    void Function (double d) ;
    //...
};
//...
void main ()
{
    derived d ;
    //...
    d.Function (2) ; // call to Derived::Function(double)
}
```

A **using** declaration in **class** **Derived** adds **Base**::*Function()* to **Derived**'s scope:

```
// un_hide.cpp
// a namespaces solution to the problem of a derived class
// member hiding a base class member

class Base
{
public:
    void Function (int i) ;
    //...
};

class Derived : public Base
{
public:
    void Function (double d) ;
    using Base::Function ; // using declaration
    //...
};
//...
```

---

```
void main ()
{
    Derived d ;

    d.Function (2) ; // call to Derived::Function(int)
}
```

In program CHNG\_ACC.CPP of Chapter 15 we saw the use of a **publicly** placed access declaration in a **privately** derived **class** to recover the **public** access specification of a data member in the base **class**:

```
//...
class Base
{
public:
    int pub_bdm0 ;
    int pub_bdm1 ;
};

// privately derived class
class Derived : private Base
{
public:
    Base::pub_bdm1 ; // access-declaration
};

void main ()
{
    Derived d_obj ;
    d_obj.pub_bdm0 ; // error: not accessible
    d_obj.pub_bdm1 ; // o.k.
}
```

A **using** declaration now makes the access declaration redundant:

```
//...
// privately derived class
class Derived : private Base
{
public:
    using Base::pub_bdm1 ; // using-declaration
};
//...
```

Stroustrup (1994) discusses an important implication of the **using** declaration for eliminating ambiguities as a result of multiple inheritance:

```
class Base0
{
public:
```

---

```

    void Function (int i) ;
}
//...
class Base1
{
public:
    void Function (double d) ;
};
//...
class Derived : public Base0, public Base1
{
public:
    void DerivedFunction ()
    {
        // error: Base0::Function() or Base1::Function()?
        Function (1) ;
    }
};

```

Two **using** declarations resolve the above ambiguity:

```

// multiple.cpp
// namespaces and multiple inheritance

class Base0
{
public:
    void Function (int i) ;
};
//...
class Base1
{
public:
    void Function (double d) ;
};
//...
class Derived : public Base0, public Base1
{
public:
    using Base0::Function ;
    using Base1::Function ;

    void DerivedFunction ()
    {
        Function (1) ; // Base0::Function(int)
    }
};
//...

```

which is a neat solution to a tricky problem!

## 19.9 Summary

The C++ **namespace** declaration is a solution to the problem of conflicting names in a translational unit. A **namespace** has its own scope and can be either named or nameless, nested, and overloaded. Access to a **namespace**'s members is via the scope resolution operator. A **using** declaration allows access to a specified **namespace** member without explicit qualification, whereas a **using** directive allows access to *all* members of a specified **namespace** without explicit qualification.

Try not to draw too many comparisons between **class** declarations and **namespace** declarations. Classes and namespaces have their similarities, but a **namespace** is not a **class**.

Stroustrup (1993b) and (1994) highlight that the **using** directive should be viewed as a transitional tool and when writing new program code a **using** declaration approach should be adopted rather than a **using** directive.

## Exercises

- 19.1 A program uses two different library Point classes placed in the header files POINT1.H and POINT2.H:

```
// point1.h           // point2.h

template <class T>      template <class T>
class Point             class Point
{
//...
};


```

Use namespaces so that both header files can be included in a program file and objects defined for each **class**.

- 19.2 It is required that a number of geometric classes are nested within a single **namespace** called **Geometry**. The classes can be categorised as points and vectors, lines and curves, surfaces and geometric shapes each of which is to be placed within its own appropriate namespace. Design the outline of the namespaces.
  - 19.3 For Exercise 19.2 define concise alias names for each nested **namespace**.
  - 19.4 Use the property that namespaces are open by placing the namespaces of Exercise 19.2 into separate header files.
  - 19.5 With the help of a **using** declaration rewrite the *main()* function for the following program code so that explicit qualification is not required to define object *c*:

```
namespace Mathematical
{
    class Complex
    {
        private:
            double re, im;
        public:
            Complex ();
    }
}
```

---

```

        : re (0.0), im (0.0) {}
Complex (double _re, double _im=0.0)
    : re (_re), im (_im) {}
//...
};

}

//...
void main ()
{
    Mathematical::Complex c (1.0, 2.0);
}

```

19.6 Are the two program code fragments below equivalent?

```

void Function () ;

namespace X
{
    ::Function ();
}

```

and:

```

namespace
{
    void Function ();
}

namespace Y
{
    ::Function ();
}

```

19.7 A **class** called Base is declared within the namespace **BaseNameSpace**:

```

namespace BaseNameSpace
{
    class Base
    {
        private:
            int bdm ;
        public:
            Base ()
                : bdm (0) {}
    };
}

```

Publicly derive from Base a **class** called Derived which is declared within its own **namespace**. Ensure that Derived's default constructor initialises the **Base**::**bdm** data member.

# An Application: a Simple Raytracing Program

*The world is really wild at heart and weird on top  
Wild At Heart (Gifford, 1990)*

*The development of a simple raytracing program is presented in this final chapter. A raytracing program is an excellent realistic graphical application, of sufficient size, in which to apply many of the C++ and object-oriented programming features discussed in previous chapters.*

*The chapter begins by presenting an overview of the raytracing method and how to implement a raytracing program in C++ using object-oriented programming methodologies. Although the raytracing program introduces several new classes, the program relies heavily on classes developed throughout the book such as Vector, LinkedList, Vector3D, Point, Line, Complex, DIBitmap and World. Thus, the raytracing program demonstrates the importance of both code reusability and good class design for the development of larger programs.*

*A raytracing program is also a good graphical application in which to illustrate the object-oriented application of classes through graphical object abstraction, inheritance and polymorphism. Numerous new classes are introduced to assist in the implementation of a raytracing program which is designed with the view of being extended. For instance, classes RGBColour and NormalisedRGBColour model an RGB (Red-Green-Blue) colour with associated operations, whereas classes RectWindow, ViewPlane and Screen encapsulate the abstractions of a rectangular window, view plane and display screen so frequently used in computer graphics. A big part of a raytracing program is the ray-object intersections, and the chapter discusses the intersection of a ray for all world objects modelled, namely a plane, polygon, triangle, quadrilateral, tetrahedron, sphere and circle. A Surface class is presented which encapsulates the different components of lighting to produce realistic illumination, reflection and transparency effects. Consideration of simple illumination models, such as specular highlights, can produce visually effective images.*

*Class World pulls together all the features of the raytracing program by encapsulating data members which model a viewer, view plane and display screen, point light source, background and ambient light, a list of world objects and object and world data files. The chapter concludes by illustrating the raytracing program by means of a simple Windows program which allows a user to generate a raytraced image, generate and display an image, or simply display a previously generated image.*



## 20.1 Why an Application and Why Raytracing?

This chapter discusses the development of a raytracing program. Although the program is relatively simple it nevertheless models a realistic and popular graphical application. The program is of a sufficiently moderate size to help demonstrate the benefits of adopting an object-oriented approach for larger, more realistic programs – when object-oriented programming really comes into its own. It is intended that the raytracing program will help illustrate the object-oriented features of C++ when applied to a particular problem and help pull together many of the classes developed in previous chapters throughout this book.

The present chapter makes good use of classes presented in previous chapters and illustrates just how much *primitive* classes can be used time and time again. You will observe in the present chapter that several of the classes presented in previous chapters have required minimal modifications. Good classes are developed over a period of time through real application. Only through real application will flaws in **class** design surface. For instance, one minor modification which has been introduced in the present chapter is implicit conversion functions between Point and Vector3D objects. The use of conversion functions should be minimised, but is advantageous in the present case so as to eliminate a great deal of explicit **class** user conversion between Point and Vector3D objects. It could be argued that this need for conversion functions has arisen because the distinction between a point and a vector has been introduced by developing separate Point and Vector3D classes. However, there are subtle distinctions between a point and a vector, and it is nice to be able to exercise these differences when writing program code.

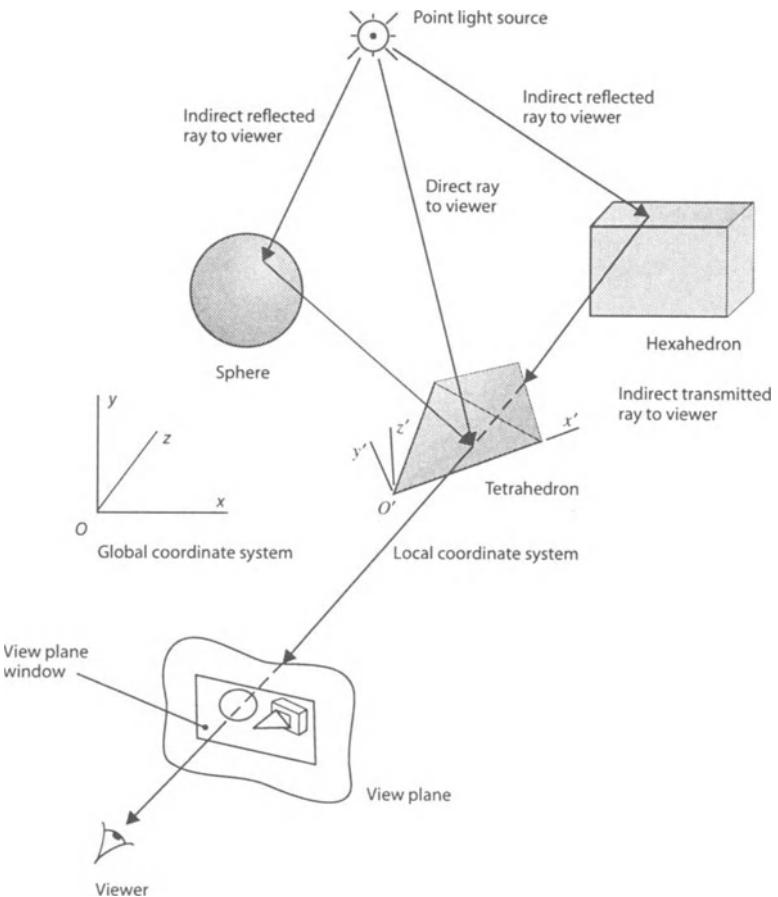
Although a relatively recent technique, raytracing is nevertheless a very popular one, and hence an excellent graphical application to illustrate the object-oriented features of C++. Raytracing naturally lends itself to object-oriented design because of the similarity between world objects (such as a viewer, a light source and objects) and **class** objects and the underlying mechanism of raytracing. Essentially, raytracing consists of projecting a ray from a viewer through a view plane and detecting whether or not the ray intersects an object, irrespective of the particular type of object. It is exactly this property of the ray-object intersection algorithms being independent of the image rendering process that allows the raytracing model to integrate neatly into a geometric **class** hierarchy.

It is important to note that in developing the raytracing program that emphasis has not been placed on the *behind the scenes* algorithms; i.e. the fastest ray-object intersection algorithms available. That is not to say that such algorithms are not important (because they are very important), but rather that emphasis is placed on the object-oriented issues of a raytracing program. Further, extravagant **double** precision and actual point intersection are used when **float** precision and parametric point intersection would generally be adopted. The philosophy behind such decisions is to design the relevant **class** member functions to be as general as possible so that they are easily transferable to applications other than raytracing, such as geometric and solid modelling.

Hopefully, the simple raytracing program presented in the present chapter will help illustrate just how sophisticated and clever raytracing programs such as POV-Ray are; for instance refer to Wells and Young (1993).

## 20.2 Recursive Raytracing

The raytracing method consists of projecting infinitesimally thin rays from a light source into a *world* domain to compute the light intensity of visible surfaces of objects. Figure 20.1



**Fig. 20.1** Tracing direct and indirect rays from a point light source through a view plane window to a viewer.

schematically illustrates the raytracing method. The world domain consists of a set of objects, a point light source, a view plane and a viewer, all positioned relative to a world *global* coordinate system (Cartesian in the present case). Three objects are shown in Fig. 20.1, namely a sphere, a tetrahedron and a hexahedron (i.e. a cube), each possessing its own *local* coordinate system and unique object description. Each object is not only geometrically unique but also unique in its surface description. Typically, surfaces are computationally modelled in terms of ambient, diffuse and specular light components and how reflective and transparent a surface is, although surface texture could also be modelled. For instance, Figure 20.1 illustrates fully opaque sphere and hexahedron objects, while the tetrahedron object is partially transparent.

Rays from the light source are characterised as either direct or indirect. A direct ray is a ray which arrives at the viewer via a single reflection from the light source, whereas an indirect ray has undergone multiple reflections or refractions before reaching the viewer. The raytracing method is referred to as *recursive* because essentially there is no difference in the raytracing algorithm in computing a direct or indirect ray. Initially, a ray is traced from a light source into the world domain and the intensity of light is computed at the first ray-object intersection. The process is then recursively repeated for *spawned* reflected and/or refracted rays from the ray-object intersection point and illumination effects accumulated. In theory, spawned rays

are generated indefinitely, although in practice an upper limit is placed on the number of spawned rays.

One of the key features of the recursive raytracing method is that the algorithm inherently incorporates object illumination, hidden surface removal, shadowing, reflection and refraction in a single model. The process of projecting a ray from a light source into the world domain and computing ray-object intersections is independent of a particular object, although the ray-object intersection algorithm is obviously different for different objects. The illumination model is also independent of a particular object. The illumination model is designed in terms of surface characteristics, incident ray, normal vector (at the intersection point of the incident ray and object) and reflected and transmitted rays.

From a computational perspective the recursive raytracing model is a *backwards* implementation, meaning that rays are traced from a viewer through the viewer's view plane into the world domain, rather than casting rays from the light source. A backwards implementation is adopted simply because we are only concerned with the finite number of rays that pass through a viewer's view plane, whereas in reality an infinite number of rays emanate from a given light source.

## 20.3 Tool Classes

Before discussing various ray-object intersection algorithms in detail in the next section, let us first remind ourselves of the classes and functions developed in previous chapters that will prove useful in the present chapter.

TOL.H/.CPP include various **const** definitions:

```
// tol.h
// various tolerances

#ifndef _TOL_H // prevent multiple includes
#define _TOL_H

extern const int REFLECTED_DEPTH ;
extern const int TRANSMITTED_DEPTH ;

extern const double TOLERANCE ;
extern const double MAX ;

#endif // _TOL_H

// tol.cpp
// various tolerances

#include "tol.h"

const int REFLECTED_DEPTH    = 5 ;
const int TRANSMITTED_DEPTH = 5 ;

const double TOLERANCE = 1e-03 ;
const double MAX      = 1e06 ;
```

The BOOL.H header file declares the Boolean **enum**:

---

```
// bool.h
// enumerated type Boolean

#ifndef _BOOL_H // prevent multiple includes
#define _BOOL_H

enum Boolean { B_FALSE, B_TRUE };

#endif // _BOOL_H
```

The data members of Boolean are named B\_FALSE (0) and B\_TRUE (1) so as not to name clash with the FALSE (0) and TRUE (1) #defines in WINDOWS.H. The MIN\_MAX.H file defines the *Min()* and *Max()* **template** functions and are based on the TMP\_FUN.CPP program of Chapter 13:

```
// min_max.h
// Min() & Max() template functions

#ifndef _MIN_MAX_H // prevent multiple includes
#define _MIN_MAX_H

// min of two objects
template <class T>
T Min (T& a, T& b)
{
    return a < b ? a : b ;
}

// max of two objects
template <class T>
T Max (T& a, T& b)
{
    return a > b ? a : b ;
}

#endif // _MIN_MAX_H
```

The Point **class** declaration for encapsulating a point in a three-dimensional space relative to an assumed Cartesian coordinate system is listed below:

```
// pt.h
// class Point

#ifndef _PT_H // prevent multiple includes
#define _PT_H

#include <iostream.h> // C++ I/O
#include <math.h> // fabs(), sqrt()
#include "bool.h" // Boolean enum
#include "vec3d.h" // Vector3D class
#include "tol.h" // tolerances
```

```

// class Point
class Point
{
protected:
    // protected data members
    double x, y, z ;
public:
    // constructors
    Point ()
        : x (0.0), y (0.0), z (0.0) {}
    Point (double x_arg, double y_arg, double z_arg=0.0)
        : x (x_arg), y (y_arg), z (z_arg) {}
    Point (double d)
        : x (d), y (d), z (d) {}
    Point (const Vector3D& v) // Vector3D to Point
        : x (v[0]), y (v[1]), z (v[2]) {}
    // copy constructor
    Point (const Point& p)
        : x (p.x), y (p.y), z (p.z) {}
    // public member functions
    double& X () { return x ; }           // non-const objects
    double& Y () { return y ; }
    double& Z () { return z ; }
    const double& X () const { return x ; } // const objects
    const double& Y () const { return y ; }
    const double& Z () const { return z ; }
    double DistanceBetweenTwoPoints
        (const Point& p=Point()) ;
    const double DistanceBetweenTwoPoints
        (const Point& p=Point()) const ;
    // overloaded operators
    Point& operator = (const Point& p) ;
    double& operator [] (int n) ;
    const double& operator [] (int n) const ;
    Vector3D operator - (const Point& p) ;
    Vector3D operator - (const Point& p) const ;
    Point operator + (const Point& p) ;
    Point operator * (const Point& p) ;
    Boolean operator == (const Point& p) ;
    Boolean operator != (const Point& p) ;
    Point operator - () ;
    operator Vector3D () // Point to Vector3D
        { return Vector3D (x, y, z) ; }
    operator Vector3D () const // Point to Vector3D
        { return Vector3D (x, y, z) ; }
    // friends
    friend ostream& operator << (ostream& s,
                                    const Point& p) ;
    friend istream& operator >> (istream& s, Point& p) ;
}; // class Point
#endif // _PT_H

```

with the implementation file in PT.CPP. The Point **class** has been discussed in numerous chapters since its debut in Chapter 9. The present implementation adds conversion functions for converting from a Point to a Vector3D object and vice versa.

The **class** declaration of Line is:

```
// line.h
// class Line

#ifndef _LINE_H // prevent multiple includes
#define _LINE_H

#include "pt.h"      // Point class
#include "bool.h"    // enum Boolean
#include "tol.h"     // tolerances

class Line
{
protected:
    Point p0, p1 ;
public:
    // constructors
    Line ()
        : p0 (), p1 () {}
    Line (const Point& p, const Point& q)
        : p0 (p), p1 (q) {}
    // copy constructor
    Line (const Line& l)
        : p0 (l.p0), p1 (l.p1) {}
    // member functions
    Point& P0 () { return p0 ; }
    const Point& P0 () const { return p0 ; }
    Point& P1 () { return p1 ; }
    const Point& P1 () const { return p1 ; }
    double Length () ;
    Boolean PointOnLine (const Point& p) ;
    Boolean Intersection (const Line& l, Point& int_p) ;
    // overloaded operators
    Line& operator = (const Line& l) ;
    Point& operator [] (int n) ;
    const Point& operator [] (int n) const ;
    Line operator - () // reverse sense
        { return Line (p1, p0) ; }
    // conversion function
    operator Vector3D () // Line to Vector3D
        { return Vector3D (p1.X()-p0.X(), p1.Y()-p0.Y(),
                           p1.Z()-p0.Z()) ; }
    // friend
    friend ostream& operator << (ostream& s, const Line& l) ;
}; // class Line

#endif // _LINE_H
```

In addition to the conversion function `Line::operator Vector3D()`, which converts a `Line` object to a `Vector3D` object, two other member functions, `PointOnLine()` and `Intersection()`, have been added to `Line`:

```
// line.cpp
// implementation file for class Line
#include "line.h" // header file
//...
// tests if a point is on a line
// returns TRUE if point on line, else FALSE
Boolean Line::PointOnLine (const Point& p)
{
    // quick test if point coincides with a line endpoint
    if (p == p0 || p == p1)
        return B_TRUE;

    // if line is in fact a point and failed previous test then
    // return logical false
    if (p0 == p1)
        return B_FALSE;

    double x10 = p1.X() - p0.X();
    double y10 = p1.Y() - p0.Y();
    double z10 = p1.Z() - p0.Z();

    if (fabs(z10) < TOLERANCE)
    {
        if ((fabs(y10) < TOLERANCE) && (fabs(x10) > TOLERANCE))
        {
            if ((fabs(p.Y()-p0.Y()) < TOLERANCE) &&
                  (p.X() > p0.X()) && (p.X() < p1.X()))
                return B_TRUE;
            else
                return B_FALSE;
        }
        else if ((fabs(x10) < TOLERANCE) &&
                   fabs(y10) > TOLERANCE)
        {
            if ((fabs(p.X()-p0.X()) < TOLERANCE) &&
                  (p.Y() > p0.Y()) && (p.Y() < p1.Y()))
                return B_TRUE;
            else
                return B_FALSE;
        }
    }
    else
    {
        double ux = (p.X()-p0.X()) / (p1.X()-p0.X());
        double uy = (p.Y()-p0.Y()) / (p1.Y()-p0.Y());
        if (fabs(uy-ux) < TOLERANCE)
            return B_TRUE;
        else
    }
}
```

```

        return B_FALSE ;
    }
}
if (fabs(y10) < TOLERANCE)
{
//...
}
if (fabs(x10) < TOLERANCE)
{
//...
}
double ux = (p.X()-p0.X()) / x10 ;
double uy = (p.Y()-p0.Y()) / y10 ;
double uz = (p.Z()-p0.Z()) / z10 ;
if ((fabs(uy-ux) < TOLERANCE) && (fabs(uy-uz) < TOLERANCE))
    return B_TRUE ;
return B_FALSE ;
} // PointOnLine()

// tests if two lines intersect
// returns TRUE if intersect, else FALSE
// returns, via reference, the point of
// intersection int_p
Boolean Line::Intersection (const Line& l, Point& int_p)
{
double s (0.0), t (0.0), denom (0.0) ;

// try x-y system first
double xlk = P1().X() - P0().X() ;
double xnm = l.P1().X() - l.P0().X() ;
double xmk = l.P0().X() - P0().X() ;
double ylk = P1().Y() - P0().Y() ;
double ynm = l.P1().Y() - l.P0().Y() ;
double ymk = l.P0().Y() - P0().Y() ;

denom = xnm*ylk - xlk*ynm ;
if (fabs(denom) > TOLERANCE)
{
    s = (xnm*ymk - xmk*ynm) / denom ;
    t = (xlk*ymk - ylk*xmk) / denom ;
}

// try y-z system
if (fabs(denom) < TOLERANCE)
{
double ylk = P1().Y() - P0().Y() ;
double ynm = l.P1().Y() - l.P0().Y() ;
double ymk = l.P0().Y() - P0().Y() ;
double zlk = P1().Z() - P0().Z() ;
double znm = l.P1().Z() - l.P0().Z() ;
double zmk = l.P0().Z() - P0().Z() ;
}
```

```

denom = ynm*zlk - ylk*znm ;
if (fabs(denom) > TOLERANCE)
{
    s = (ynm*zmk - ymk*znm) / denom ;
    t = (ylk*zmk - zlk*ymk) / denom ;
}
}

// try x-z system
if (fabs(denom) < TOLERANCE)
{
    double xlk = P1().X() - P0().X() ;
    double xnm = l.P1().X() - l.P0().X() ;
    double xmk = l.P0().X() - P0().X() ;
    double zlk = P1().Z() - P0().Z() ;
    double znm = l.P1().Z() - l.P0().Z() ;
    double zmk = l.P0().Z() - P0().Z() ;

    denom = xnm*zlk - xlk*znm ;
    if (fabs(denom) > TOLERANCE)
    {
        s = (xnm*zmk - xmk*znm) / denom ;
        t = (xlk*zmk - zlk*xmk) / denom ;
    }
}
// parallel lines
if (fabs(denom) < TOLERANCE)
    return B_FALSE ;

if (s<0.0 || t<0.0 || s>1.0 || t>1.0)
    return B_FALSE ;
else // intersection
{
    // pt. of intersection (use s)
    int_p.X () = P0().X () + (P1().X()-P0().X())*s ;
    int_p.Y () = P0().Y () + (P1().Y()-P0().Y())*s ;
    int_p.Z () = P0().Z () + (P1().Z()-P0().Z())*s ;
    return B_TRUE ;
}
} // Intersection()
//...

```

*Line*::*PointOnLine()* returns logical-true if a given point is on a line segment, else it returns logical-false. To determine whether a point  $p_i(x_i, y_i, z_i)$  lies on a line segment  $l$ , with end-points  $p_0(x_0, y_0, z_0)$  and  $p_1(x_1, y_1, z_1)$ , we first express  $l$  in parametric form:

$$x = (x_1 - x_0)t + x_0$$

$$y = (y_1 - y_0)t + y_0$$

$$z = (z_1 - z_0)t + z_0$$

with  $t \in [0,1]$ . Rearranging the above in terms of  $t$ , with appropriate subscripts, we have:

$$t_x = \frac{x_i - x_0}{x_1 - x_0}, \quad t_y = \frac{y_i - y_0}{y_1 - y_0}, \quad t_z = \frac{z_i - z_0}{z_1 - z_0},$$

If  $t_x = t_y = t_z$  then point  $p_i$  is on the line  $l$ , otherwise the  $p_i$  is not on the line. However, the degenerate cases of the line being parallel to one or two coordinate planes have to be tested. For instance, if  $z_1 - z_0 = 0$  then  $l$  is parallel to the  $xy$ -plane.

The `Line::Intersection()` member function tests whether two `Line` objects intersect and if so returns, via reference, the intersection point. The method of solution adopted is to represent the two lines  $l_{mn}(p_m(x_m, y_m, z_m), p_n(x_n, y_n, z_n))$  and  $l_{kl}(p_k(x_k, y_k, z_k), p_l(x_l, y_l, z_l))$  in parametric form with parametric variables  $t \in [0,1]$  and  $s \in [0,1]$  respectively, and equating the parametric equations of lines  $l_{mn}$  and  $l_{kl}$  we have:

$$\begin{aligned} x_k + (x_l - x_k)s &= x_m + (x_n - x_m)t \\ y_k + (y_l - y_k)s &= y_m + (y_n - y_m)t \\ z_k + (z_l - z_k)s &= z_m + (z_n - z_m)t \end{aligned}$$

As with the previous problem of testing whether a point lies on a line, the solution and implementation at first appear trivial, but there are possible degenerate cases that have to be caught by the implementation. Upon closer examination of the above system of equations we observe that we have an overdetermined system, since there are more equations (i.e. three) than the two unknowns  $s$  and  $t$ . Again, it is necessary to test for lines parallel to coordinate planes. For instance, considering the  $xy$ -plane and hence solving the first two equations above for  $s$  and  $t$  we find:

$$\begin{aligned} s &= \frac{(x_n - x_m)(y_m - y_k) - (x_m - x_k)(y_n - y_m)}{(x_n - x_m)(y_l - y_k) - (x_l - x_k)(y_n - y_m)} \\ t &= \frac{(x_l - x_k)(y_m - y_k) - (x_m - x_k)(y_l - y_k)}{(x_n - x_m)(y_l - y_k) - (x_l - x_k)(y_n - y_m)} \end{aligned}$$

If both  $0 \leq s \leq 1$  and  $0 \leq t \leq 1$  the two line segments intersect.

The `Vector` and `Vector3D` classes are essentially identical to `VEC_TMP.H` of Chapter 13 and `VEC3D.H` of Chapter 15 respectively:

```
// vec.h
// template class Vector

#ifndef _VEC_H // prevent multiple includes
#define _VEC_H

#include <iostream.h> // C++ I/O
#include <iomanip.h> // setprecision(), ...
#include <assert.h> // assert()

// template class Vector
template <class T>
```

```

class Vector
{
protected:
    // data members
    T* array ;
    int n_elements ;
    // member function
    void Allocate (int n) ;
public:
    // constructors
    Vector () { Allocate (3) ; }
    Vector (int n, T obj) ;
    // copy constructor
    Vector (const Vector& v) ;
    // destructor
    ~Vector () { delete [] array ; array = NULL ;
                  n_elements = 0 ; }
    // member functions
    int NumberElements () const { return n_elements ; }
    T& Value (int index) ;
    const T& Value (int index) const ;
    void New (int new_n) ;
    // overloaded operators
    Vector& operator = (const Vector& v) ;
    T& operator [] (int index) ;
    const T& operator [] (int index) const ;
    Vector operator + (const Vector& v) ;
    Vector operator - (const Vector& v) ;
    Vector operator * (const Vector& v) ;
    Vector operator += (const Vector& v) ;
    Vector operator -= (const Vector& v) ;
    Vector operator - () ;
    // friend
    friend ostream& operator << (ostream& s,
                                         const Vector<T>& v) ;
}; // class Vector
//...
#endif // _VEC_H

// vec3d.h
// class Vector3D

#ifndef _VEC3D_H // prevent multiple includes
#define _VEC3D_H

#include <math.h> // sqrt()
#include "vec.h" // template class Vector
#include "tol.h" // tolerances

class Vector3D : public Vector<double>
{

```

```

public:
    // constructors
    Vector3D ()
        : Vector<double> () {}
    Vector3D (double arg1, double arg2, double arg3=0.0)
        : Vector<double> ()
        { array[0]=arg1; array[1]=arg2; array[2]=arg3; }
    Vector3D (const Vector<double>& v)
        : Vector<double> ()
        { array[0]=v[0]; array[1]=v[1]; array[2]=v[2]; }
    // copy constructor
    Vector3D (const Vector3D& v)
        : Vector<double> (v) {}
    // destructor
    ~Vector3D () {}
    // member functions
    double Norm () ;
    void Normalise () ;
    double DotProduct (const Vector3D& v) ;
    Vector3D CrossProduct (const Vector3D& v) ;
    void DirectionCosines (double& cos_alpha,
                           double& cos_beta,
                           double& cos_gamma) ;
    // overloaded operator
    Vector3D operator * (double d) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s, Vector3D& v) ;
}; // class Vector3D

#endif // _VEC3D_H

```

with corresponding implementation file VEC3D.CPP. The following **template class** declarations of **LinkedList** and **LinkedListIterator** are identical to those declared in LL.H and LLI.H, respectively, of Chapter 13:

```

// ll.h
// template class LinkedList

#ifndef _LL_H // prevent multiple includes
#define _LL_H

#include <iostream.h> // C++ I/O
#include <assert.h> // assert()

#include "lli.h" // template class LinkedListIterator
#include "swap.h" // Swap template function
#include "bool.h" // type Boolean

template <class T>
class LinkedList
{

```

```

private:
    int num_nodes ;
    // nested Node class
    class Node
    {
    //...
    } * first, * last ; // class Node
public:
    enum { START, END };
    // constructors
    LinkedList ()
        : num_nodes (0), first (NULL), last (NULL) {}
    LinkedList (const LinkedList& l) ;
    // destructor
    ~LinkedList ()
        { delete first ; num_nodes = 0 ;
          first = last = NULL ; }
    // member functions
    int NumberNodes () const
        { return num_nodes ; }
    Boolean Empty ()
        { return num_nodes==0 ? B_TRUE : B_FALSE ; }
    int Search (const T& obj) ;
    void Insert (const T& obj, int loc) ;
    void Head (const T& obj) ;
    void Append (const T& obj) ;
    void Delete (int loc) ;
    void Sort () ;
    // overloaded operators
    LinkedList& operator = (const LinkedList& l) ;
    T& operator [] (int index) ;
    Boolean operator == (const LinkedList& l) ;
    Boolean operator != (const LinkedList& l) ;
    Boolean operator < (const LinkedList& l) ;
    // friends
    friend ostream& operator << (ostream& s,
                                    const LinkedList<T>& l) ;
    friend class LinkedListIterator<T> ;
}; // template class LinkedList
//...
#endif // _LL_H

// lli.h
// template class LinkedListIterator

#ifndef _LLI_H // prevent multiple includes
#define _LLI_H

#include "ll.h" // template class LinkedList

template <class T>

```

```

class LinkedListIterator
{
private:
    LinkedList<T>::Node* c_node ; // current node
    const LinkedList<T>* c_list ; // current list
public:
    // constructor
    LinkedListIterator (const LinkedList<T>& l)
        : c_node (l.first), c_list (&l) {}
    // member functions
    Boolean More ()    const { return c_node != NULL ?
                           B_TRUE : B_FALSE ; }
    T      Current () const { return c_node->data ; }
    void   Next ()    { c_node = c_node->next ; }
    void   ReStart () { c_node = c_list->first ; }
    // overloaded operators
    const T& operator ++ () // prefix
    { c_node=c_node->next ; return c_node->data ; }
    const T& operator ++ (int) // postfix
    { LinkedList<T>::Node* temp=c_node ;
        c_node=c_node->next ; return temp->data ; }
}; // template class LinkedListIterator

```

```
#endif // _LLI_H
```

The Complex **class** declaration listed below for the manipulation of complex numbers is an extended version of FR\_IO.CPP of Chapter 11:

```

// cplx.h
// class Complex

#ifndef _CPLX_H // prevent multiple includes
#define _CPLX_H

#include <iostream.h> // C++ I/O
#include <stdlib.h> // exit()
#include <math.h> // sqrt(), cos()...
#include "tol.h" // tolerances

class Complex
{
private:
    double re, im ;
public:
    // constructors
    Complex () ; // default argument
    Complex (double r, double i=0.0) ;
    // access member functions
    double& Real () { return re ; }
    double& Imag () { return im ; }
}

```

```

const double& Real () const { return re ; }
const double& Imag () const { return im ; }
// member functions
double Abs () ;
double Argument () ;
Complex Sqrt () ;
// overloaded operators/friends
friend Complex operator + (const Complex& c1,
                           const Complex& c2) ;
//...
}; // class Complex

#endif // C_CPLX_H

```

The *Abs()*, *Argument()* and *Sqrt()* member functions have been added to *Complex* and are defined in *CPLX.CPP*:

```

// cplx.cpp
// implementation file for class Complex

#include "cplx.h"
//...
// complex absolute value
double Complex::Abs ()
{
    return sqrt (re*re + im*im) ;
}

// complex argument
double Complex::Argument ()
{
    if (fabs(re) < TOLERANCE && fabs(im) > TOLERANCE)
        return 2.0 * atan (1.0) ;
    if (fabs(re) < TOLERANCE && fabs(im) < TOLERANCE)
        return 0.0 ;
    return atan2 (im, re) ;
}

// complex square root
Complex Complex::Sqrt ()
{
    return sqrt (Abs ()) * Complex (cos(Argument () / 2.0),
                                    sin(Argument () / 2.0)) ;
}
//...

```

*Complex::Abs()* returns the absolute value,  $|z|$ , of a complex number  $z (=x+iy)$ :

$$|z| = (x^2 + y^2)^{1/2}$$

while `Complex::Argument()` returns the argument,  $\theta$ , of  $z$ . The function `Complex::Sqrt()` returns the square root of a complex number and is defined as:

$$z^{1/2} = |z|^{1/2} \left( \cos \frac{\theta}{2} + i \sin \frac{\theta}{2} \right)$$

`Complex` is only used in the definition of `Surface::TransmittedRay()` for determining the direction of a ray that has passed through a transparent object.

A **class** which plays a key role in the raytracing program is the `World class` introduced in Chapter 17 and declared and defined in `WORLD.H` and `WORLD.CPP`, respectively. The `World class` not only encapsulates a set of objects, but now also encapsulates a set of world objects and defines a member function called `GenerateRayTracedImage()`, which, as the name suggests, generates a raytraced image. The `World class` will be discussed in more detail later in the chapter, but note that `World` encapsulates `ReadFile` and `WriteFile` data members for the manipulation of input and output file stream objects which are declared and defined in `RW_FILE.H` and `RW_FILE.CPP`.

The raytracing program generates 24-bit RGB Windows format device-independent bitmaps. It is required that the raytraced images are written to and read from a disk file and the `DIBitmap class`, developed in Chapter 17, is an ideal candidate for the manipulation of device-independent bitmap images. `DIBitmap` is declared in `DIB.H` with corresponding implementation file `DIB.CPP`.

Two classes which have been specifically developed for the present chapter to assist in the manipulation of RGB colours and for accumulating RGB colour intensities due to different light illumination components are the `RGBColour` and `NormalisedRGBColour` classes and are declared in `RGB.H` with corresponding implementation file `RGB.CPP`:

```
// rgb.h
// classes RGBColour and NormalisedRGBColour

#ifndef _RGB_H // prevent multiple includes
#define _RGB_H

#include <windows.h> // Windows
#include <iostream.h> // C++ I/O

// BYTE red, green and blue intensities
class RGBColour
{
private:
    BYTE red, green, blue ;
public:
    RGBColour (BYTE r=0, BYTE g=0, BYTE b=0)
        : red (r), green (g), blue (b) {}
    RGBColour (const RGBColour& c)
        : red (c.red), green (c.green), blue (c.blue) {}
    BYTE Red () const { return red ; }
    BYTE Green () const { return green ; }
    BYTE Blue () const { return blue ; }
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
```

```

        const RGBColour& rgb) ;
friend istream& operator >> (istream& s,
                               RGBColour& rgb) ;
}; // class RGBColour

// float red, green and blue intensities
class NormalisedRGBColour
{
private:
    float red, green, blue ;
public:
    NormalisedRGBColour (float r=0.0, float g=0.0,
                          float b=0.0)
        : red (r), green (g), blue (b) {}
    NormalisedRGBColour (const RGBColour& rgb)
        : red ((float)rgb.Red()/255),
          green ((float)rgb.Green()/255),
          blue ((float)rgb.Blue()/255) {}
    NormalisedRGBColour (const NormalisedRGBColour& c)
        : red (c.red), green (c.green), blue (c.blue) {}
    float& Red () { return red ; }
    float& Green () { return green ; }
    float& Blue () { return blue ; }
    // overloaded operators
    NormalisedRGBColour operator +
        (const NormalisedRGBColour& rgb) ;
    NormalisedRGBColour operator -
        (const NormalisedRGBColour& rgb) ;
    NormalisedRGBColour operator *
        (const NormalisedRGBColour& rgb) ;
    NormalisedRGBColour operator * (const float& f) ;
    NormalisedRGBColour operator / (const float& f) ;
    NormalisedRGBColour operator +=
        (const NormalisedRGBColour& rgb) ;
    NormalisedRGBColour operator ==
        (const NormalisedRGBColour& rgb) ;
    // conversion function
operator RGBColour () const ;
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const NormalisedRGBColour& rgb) ;
    friend istream& operator >> (istream& s,
                                   NormalisedRGBColour& rgb) ;
}; // class NormalisedRGBColour

#endif // _RGB_H

// rgb.cpp
// implementation file for classes RGBColour and
// NormalisedRGBColour

```

```
#include "rgb.h"

//...
// conversion function:
// convert from normalised RGB to RGB and truncate if
// required
NormalisedRGBColour::operator RGBColour () const
{
    int r (red*255), g (green*255), b (blue*255) ;

    if (r > 255)
        r = 255 ;
    if (g > 255)
        g = 255 ;
    if (b > 255)
        b = 255 ;

    return RGBColour (r, g, b) ;
}
//...
```

RGBColour and NormalisedRGBColour are essentially equivalent in that they both encapsulate RGB colour intensities, except that RGBColour encapsulates BYTE (**unsigned char**) RGB intensities in the range  $0 \leq r, g, b \leq 255$ , whereas NormalisedRGBColour encapsulates **float** RGB intensities in the range  $0 \leq r, g, b \leq 1$ . NormalisedRGBColour has been developed because it is generally easier and more accurate to manipulate normalised RGB floating-point number colour intensities than BYTE colour intensities in a raytracing program. When an RGB colour is finally determined it is a simple matter of converting from a normalised RGB colour to a non-normalised RGB colour, and NormalisedRGBColour supports two conversion functions for converting from a NormalisedRGBColour object to an RGBColour object and vice versa. As shown above in RGB.CPP, when converting from a NormalisedRGBColour object to a RGBColour object the individual RGB colour intensities are truncated if out of range.

Another **class** which will prove useful later is BoundingBox:

```
// bbox.h
// class BoundingBox
// an axis-aligned bounding box class

#ifndef _BBOX_H // prevent multiple includes
#define _BBOX_H

#include "pt.h" // class Point

class BoundingBox
{
private:
    Point nearest, furthest ;
public:
    // constructors
    BoundingBox () {}
```

```

BoundingBox (const Point& near_pt, const Point& far_pt)
    : nearest (near_pt), furthest (far_pt) {}
BoundingBox (const BoundingBox& bb)
    : nearest (bb.nearest), furthest (bb.furthest) {}

// member functions
Point& Nearest () { return nearest ; }
Point& Furthest () { return furthest ; }
Point Centre ()
{ return Point (nearest.X()+
                (furthest.X()-nearest.X())/2.0,
                nearest.Y()+
                (furthest.Y()-nearest.Y())/2.0,
                nearest.Z()+
                (furthest.Z()-nearest.Z())/2.0) ; }

double EnclosingRadius ()
{ return furthest.DistanceBetweenTwoPoints
  (this->Centre()) ; }

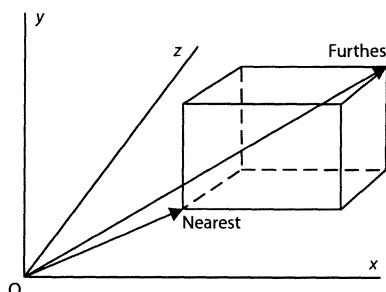
// friend/overloaded operator
friend ostream& operator << (ostream& s,
                                const BoundingBox& bb)
{ return s << bb.nearest << ", " << bb.furthest ; }

}; // class BoundingBox

#endif // _BBOX_H

```

BoundingBox models an axis-aligned bounding box consisting of six planes that are parallel to the coordinate planes and has the property that opposite faces of the box have parallel planes; see Fig. 20.2. The advantage of an axis-aligned bounding box compared with a non-axis-aligned bounding box is that an axis-aligned bounding box can be characterised solely in terms of *nearest* and *furthest* points. The terminology *bounding box* is used rather than simply *box* because BoundingBox will be used to define a bounding volume which fully encloses a given object without BoundingBox actually defining a box object. BoundingBox::Centre() and BoundingBox::EnclosingRadius() return the centre and radius of a sphere which fully encloses the bounding box object respectively.



**Fig. 20.2** An axis-aligned bounding box.

## 20.4 Ray-object intersections

For each ray traced in the world domain, the raytracing algorithm tests for an object intersection by the ray. If an object is intersected by a ray and multiple intersections exist then the nearest intersection point is chosen from the ray origin. Choosing the nearest ray-object intersection point ensures that only visible surfaces are seen by a viewer. If no intersection is found then the background colour is chosen. Since the program has no way of knowing in advance the orientation of objects relative to a viewer, because the viewer's position and view plane are variable, a given ray must be tested for intersection with every object in the world domain. Consequently, ray-object intersection algorithms play a key role in raytracing and are the most computationally expensive process of generating a raytraced image.

From an implementation point of view the ray-object intersection process is completely independent of the image-rendering process and therefore naturally lends itself to a geometric **class** hierarchy. Furthermore, the ray-object intersection process being detached from the rendering process allows new object types to be inserted easily in the raytracing program framework.

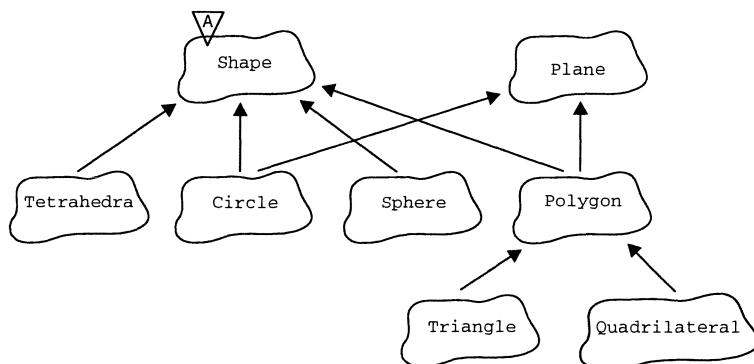
Let's now introduce the object hierarchy used for the raytracing program which extends the Shape object **class** hierarchy previously presented in Chapter 15; refer to SHAPE\_V.H and Fig. 15.12. The Shape **class** hierarchy used in the present chapter (see Fig. 20.3), is identical to that of Fig. 15.12 except that a Sphere **class** has been added. The declaration of abstract **class** Shape is:

```
// shape.h
// abstract class Shape

#ifndef _SHAPE_H // prevent multiple includes
#define _SHAPE_H

#include "vec3d.h" // class Vector3D
#include "pt.h" // class Point
#include "line.h" // class Line
#include "surf.h" // class Surface
#include "bbox.h" // class BoundingBox

class Shape
{
```



**Fig. 20.3** Shape class hierarchy of the RayTracer program.

```

protected:
    int      number ;
    Surface surface ;

public:
    Shape ()
        : number (0), surface () {}
    Shape (int n, const Surface& s)
        : number (n), surface (s) {}
    // destructor
    virtual ~Shape () { number = 0 ; }
    // member functions
    int&      Number ()          { return number ; }
    const int& Number () const { return number ; }
    Surface&     Surf ()          { return surface ; }
    const Surface& Surf () const { return surface ; }
    // pure virtual member functions
    virtual Vector3D Normal (const Point& p) = 0 ;
    virtual Point Centroid () = 0 ;
    virtual double Perimeter () = 0 ;
    virtual double SurfaceArea () = 0 ;
    virtual double Volume () = 0 ;
    virtual BoundingBox BBox () = 0 ;
    virtual Point NearestIntersection (const Line& l,
                                      Boolean& intersect) = 0 ;
    virtual void Print (ostream& s=cout) = 0 ;
    // overloaded operator
    friend ostream& operator << (ostream& s, Shape& shp)
        { shp.Print (s) ; return s ; }
}; // class Shape

#endif // _SHAPE_H

```

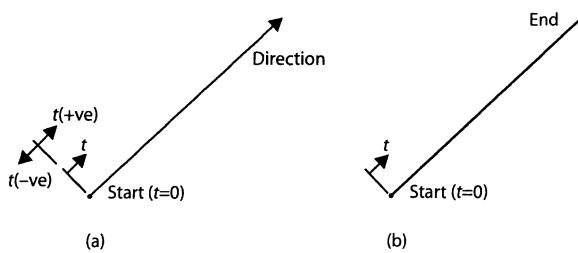
A new data member, `surface`, of **class** `Surface` has been added to **class** `Shape` with corresponding access member functions. The **class** `Surface` will be discussed later in connection with object surfaces, but for now it suffices to state that `Surface` encapsulates various surface characteristics. A **virtual** member function, `BBox()`, has also been added to `Shape` and returns an axis-aligned bounding box object for an object of a derived **class**. With respect to ray-object intersections, the key addition to **class** `Shape` is the `NearestIntersection()` **virtual** member function, which is passed a `Line` object and returns, via reference, a `Boolean` object indicating whether or not the `Line` object intersects the `shape` object acted on by the member function. If an intersection occurs the `Point` object returned indicates the actual intersection point. `Shape::NearestIntersection()` will be the focus of discussion in the following subsections.

The generally adopted method of testing for intersections between a ray and an object, in the majority of raytracing programs, is to define the ray in terms of a start point and a vector indicating the ray's direction; see Fig. 20.4(a). Such a ray could be implemented as a **class** called, say, `Ray`:

```

class Ray
{
private:

```



**Fig. 20.4** (a) A ray defined in terms of a start point and a direction vector which extends indefinitely in one direction. (b) A ray defined in terms of a line segment consisting of a start point and an end point.

```
Point      start ;
Vector3D  direction ;
//...
};
```

Alternatively, the present raytracing program makes use of the **Line class** discussed previously by defining a ray in terms of a line segment consisting of two end-points; see Fig. 20.4(b):

```
class Line
{
private:
    Point p0, p1 ;
//...
};
```

The main criterion for adopting the **Line** approach is to allow the developed ray-object intersection **class** member functions to be as generic as possible and to prevent developing separate routines for both **Line** and **Ray** objects.

### 20.4.1 Ray–Plane Intersection

This section examines the intersection of a line segment with a plane. If a line  $l(p_0, p_1)$  and plane  $(Ax + By + Cz + D = 0)$  intersect then they have a common point, say  $p_i(x_i, y_i, z_i)$ , and this point is found by simultaneously solving the following equations (refer to Fig. 20.5(a)):

$$Ax_i + By_i + Cz_i + D = 0$$

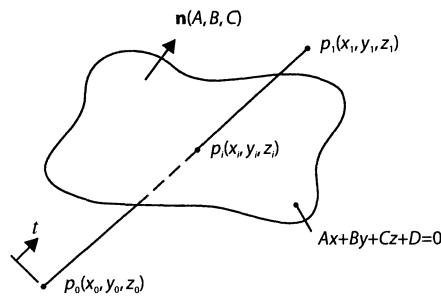
$$x_i = x_0 + (x_1 - x_0)t_i$$

$$y_i = y_0 + (y_1 - y_0)t_i$$

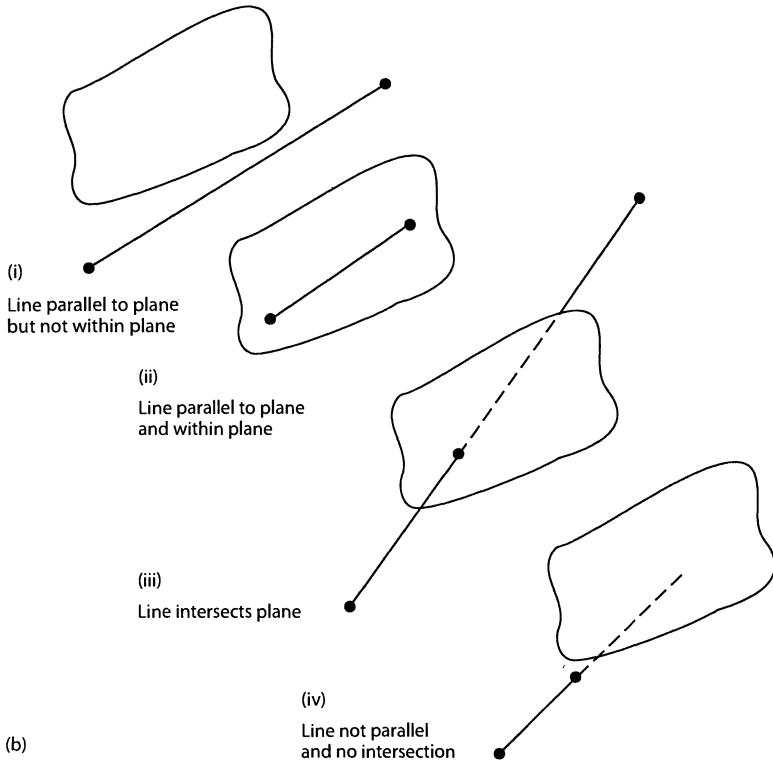
$$z_i = z_0 + (z_1 - z_0)t_i$$

Solving for  $t_i$ :

$$t_i = -\frac{Ax_0 + By_0 + Cz_0 + D}{A(x_1 - x_0) + B(y_1 - y_0) + C(z_1 - z_0)} = -\frac{\mathbf{n} \cdot \mathbf{p}_0 + D}{\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_0)}$$



(a)

**Fig. 20.5** (a) A line segment and plane intersection. (b) Four line–plane configurations.

where  $\mathbf{n}(A, B, C)$  denotes the normal vector to the plane. This equation characterises the four possible line–plane configurations shown in Fig. 20.5(b). If  $\mathbf{n} \cdot (p_1 - p_0) = 0$  the line is parallel to the plane, and if  $\mathbf{n} \cdot p_0 + D = 0$  and  $\mathbf{n} \cdot p_1 + D = 0$  the line also lies within the plane. If  $0 \leq t_i \leq 1$  the line intersects the plane at a single point  $p_i$ ; otherwise the line does not intersect the plane.

The above line–plane intersection algorithm is implemented in the member function `Plane::Intersection()`, but before listing this function let's remind ourselves of the **class** declaration of `Plane`:

```
// plane.h
// class Plane
```

```

#ifndef _PLANE_H // prevent multiple includes
#define _PLANE_H

#include "vec3d.h" // class Vector3D
#include "pt.h" // class Point
#include "line.h" // class Line
#include "tol.h" // tolerances

class Plane
{
protected:
    Vector3D normal ; // A, B, C of plane
    double d ; // D of plane
public:
    // constructors
    Plane () : d (0.0) {}
    Plane (double a, double b, double c, double d) ;
    Plane (const Vector3D& n, double d_arg) ;
    Plane (Point* p_array) ;
    Plane (const Vector<Point>& p_array) ;
    // copy constructor
    Plane (const Plane& p) ;
    // destructor
    ~Plane () ;
    // member functions
    Vector3D Normal (const Point& p) { return normal ; }
    double PerpendicularDistanceToPlane (const Point& p) ;
    Boolean PointOnPlane (const Point& p) ;
    Boolean PointOnPlane (const Point& p,
                         const Point& known_plane_pt) ;
    Boolean PlaneCoordinates (const Point& g_p,
                             const Point& l_o,
                             const Vector3D& I_a,
                             Point& plane_pt) ;
    Point GlobalCoordinates (const Point& l_p,
                            const Vector3D& Ip,
                            const Vector3D& Jq,
                            const Point& l_origin) ;
    Point Intersection (const Line& l,
                       int& intersect) ;
    // overloaded operator
    Plane& operator = (const Plane& p) ;
    // friend/overloaded operator
    friend ostream& operator << (ostream& s,
                                    const Plane& p) ;
}; // class Plane

#endif // _PLANE_H

```

The definition of `Plane::Intersection()` is:

```

// returns point of intersection between a plane and a line
// intersect=1 if line intersects plane, 0 if no
// intersection, -1 if parallel and -2 if in plane
Point Plane::Intersection (const Line& l, int& intersect)
{
    Point ip ; // intersection point

    // zero length normal (say, equivalent to: parallel to
    // plane)
    if (fabs (normal.Norm()) < TOLERANCE)
    {
        intersect = -1 ;
        return ip ;
    }

    // determine parametric t
    double denom = normal.DotProduct (l.P1()-l.P0()) ;
    // line parallel to plane
    if (fabs (denom) < TOLERANCE)
    {
        if (PointOnPlane(l.P0()) && PointOnPlane(l.P1()))
        {
            // parallel and on plane
            intersect = -2 ;
            return ip ;
        }
        // parallel but not on plane
        intersect = -1 ;
        return ip ;
    }

    double t = (-d - normal.DotProduct (l.P0())) / denom ;

    // intersection
    if (t >= 0.0 && t <= 1.0)
    {
        intersect = 1 ;
        ip.X() = l.P0().X() + ((l.P1().X()-l.P0().X()) * t) ;
        ip.Y() = l.P0().Y() + ((l.P1().Y()-l.P0().Y()) * t) ;
        ip.Z() = l.P0().Z() + ((l.P1().Z()-l.P0().Z()) * t) ;
    }
    // not parallel, not in plane and no
    // intersection
    else
        intersect = 0 ;
    return ip ;
}

```

and returns, via reference, an integer value of 0 if no intersection occurs, 1 if the line and plane intersect at a point, -1 if the line is parallel to the plane but does not lie within the plane and -2 if the line is parallel to the plane and lies within the plane.

The function `Plane::PerpendicularDistanceToPlane()` returns the perpendicular distance,  $d$ , from a point  $p_i(x_i, y_i, z_i)$  to a plane:

$$d^2 = \frac{(Ax_i + By_i + Cz_i + D)^2}{A^2 + B^2 + C^2} = \frac{(\mathbf{n} \cdot \mathbf{p}_i + D)^2}{\mathbf{n} \cdot \mathbf{n}}$$

and is implemented as:

```
double Plane::PerpendicularDistanceToPlane (const Point& p)
{
    // zero length normal
    if (fabs (normal.Norm()) < TOLERANCE)
        return -1.0 ;

    double numer = pow (normal.DotProduct(p)+d, 2) ;
    // note: denominator=1 since normal is normalised
    return sqrt (numer) ;
}
```

The two versions of `Plane::PointOnPlane()` return logical-true if a given point lies on a plane, else they return logical-false. The first version of `PointOnPlane()` simply tests the perpendicular distance of the point to the plane:

```
Boolean Plane::PointOnPlane (const Point& p)
{
    return (fabs(PerpendicularDistanceToPlane(p)) < TOLERANCE)
        ? B_TRUE : B_FALSE ;
}
```

The second version tests whether a point  $p_i$  lies on a plane by computing the dot product of the plane normal and a vector formed by  $p_i$  and a point known to lie within the plane,  $p_0$ . If point  $p_i$  lies within the plane then  $(p_i-p_0) \cdot \mathbf{n}=0$ , since the two vectors  $(p_i-p_0)$  and  $\mathbf{n}$  are perpendicular:

```
Boolean Plane::PointOnPlane (const Point& p,
                           const Point& known_plane_pt)
{
    return (fabs((p-known_plane_pt).DotProduct (normal))
        < TOLERANCE) ? B_TRUE : B_FALSE ;
}
```

There are two other member functions of **class** `Plane` which warrant closer examination, namely `PlaneCoordinates()` and `GlobalCoordinates()`:

```
Boolean Plane::PlaneCoordinates (const Point& g_p,
                                 const Point& l_o,
                                 const Vector3D& I_a,
                                 Point& plane_pt)
{
    // If not in plane
    if (fabs(I_a.DotProduct (normal)) > TOLERANCE)
```

```

    return B_FALSE ;
// local origin not in plane
if (!this->PointOnPlane(l_o))
    return B_FALSE ;
// point not in plane
if (!this->PointOnPlane(g_p))
    return B_FALSE ;

// Ip, Jq & Kr
Vector3D Ip = l_a ;
Vector3D Jq = Ip.CrossProduct (normal) ;
Vector3D Kr = normal ;

// normalise (Kr already normalised)
Ip.Normalise () ;
Jq.Normalise () ;

// local vector in plane
Vector3D l_v = g_p - l_o ;

double d_denom = Ip[0]*Jq[1]*Kr[2] - Ip[0]*Jq[2]*Kr[1] +
                  Ip[1]*Jq[2]*Kr[0] - Ip[1]*Jq[0]*Kr[2] +
                  Ip[2]*Jq[0]*Kr[1] - Ip[2]*Jq[1]*Kr[0] ;

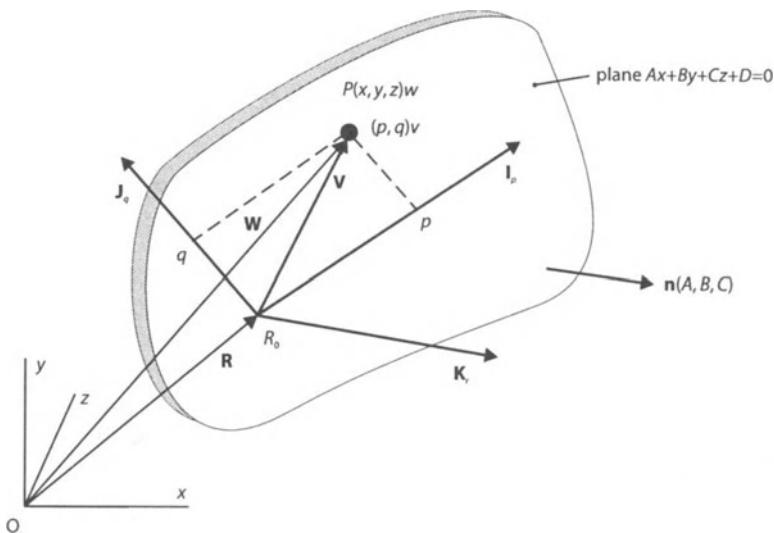
plane_pt.X () = (l_v[0]*(Jq[1]*Kr[2]-Kr[1]*Jq[2]) -
                  l_v[1]*(Jq[0]*Kr[2]-Kr[0]*Jq[2]) +
                  l_v[2]*(Jq[0]*Kr[1]-Kr[0]*Jq[1])) / d_denom ;
plane_pt.Y () = (l_v[0]*(Ip[2]*Kr[1]-Ip[1]*Kr[2]) -
                  l_v[1]*(Ip[2]*Kr[0]-Ip[0]*Kr[2]) +
                  l_v[2]*(Ip[1]*Kr[0]-Ip[0]*Kr[1])) / d_denom ;

return B_TRUE ;
} // Plane::PlaneCoordinates()

Point Plane::GlobalCoordinates (const Point& l_p,
                                const Vector3D& Ip,
                                const Vector3D& Jq,
                                const Point& l_origin)
{
    return Point (l_origin.X() + l_p.X()*Ip[0] + l_p.Y()*Jq[0],
                  l_origin.Y() + l_p.X()*Ip[1] + l_p.Y()*Jq[1],
                  l_origin.Z() + l_p.X()*Ip[2] + l_p.Y()*Jq[2]) ;
}

```

To understand `Plane::PlaneCoordinates()` consider the problem of determining the coordinates  $(p, q)$  of a point lying in a plane given the world coordinates of the point  $P(x, y, z)$ , local origin  $R_0(x_0, y_0, z_0)$  of the plane coordinate axes and local abscissa axis  $I_p$ ; see Fig. 20.6, which is based on Plastock and Kalley (1986, Fig. 8.11). If  $\mathbf{V}(V_x, V_y, V_z)$  is the plane vector of  $P$  with respect to the plane origin  $R_0$ , then  $\mathbf{V}=p\mathbf{I}_p+q\mathbf{J}_q+r\mathbf{K}_r$ , where  $\mathbf{I}_p$  and  $\mathbf{J}_q$  are the plane axes and  $\mathbf{K}_r$  is normal to the plane and therefore equivalent to  $\mathbf{n}(A, B, C)$ . Expressing  $\mathbf{V}$  in matrix form we have:



**Fig. 20.6** World and plane coordinates of a point  $P$ .

$$\begin{bmatrix} I_x & J_x & K_x \\ I_y & J_y & K_y \\ I_z & J_z & K_z \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix}$$

Solving this linear system of equations for  $p$  and  $q$  by Cramer's rule (noting that  $r=0$  since point  $P$  lies within the plane), we have:

$$p = \frac{V_x(J_yK_z - K_yJ_z) - V_y(J_xK_z - K_xJ_z) + V_z(J_xK_y - K_xJ_y)}{M}$$

$$q = \frac{V_x(I_zK_y - I_yK_z) - V_y(I_zK_x - I_xK_z) + V_z(I_yK_x - I_xK_y)}{M}$$

where  $M$  is given by:

$$M = I_xJ_yK_z - I_xJ_zK_y + I_yJ_zK_x - I_yJ_xK_z + I_zJ_xK_y - I_zJ_yK_x$$

Although the above implementation of `PlaneCoordinates()` computes the value of  $M$ , `d_denom`, it is observed that  $M=\pm 1$  depending on the sign of  $J_q$ , since  $J_q=I_p \times \mathbf{n}$ .

`Plane::GlobalCoordinates()` addresses the contrary problem of determining the world coordinates of a point given the local plane coordinates  $p$  and  $q$ , the plane axes  $I_p$  and  $J_q$  and the plane origin. Using the notation of Fig. 20.6 we have:

$$\mathbf{W} = \mathbf{R} + \mathbf{V} = \mathbf{R} + p\mathbf{I}_p + q\mathbf{J}_q$$

Both `Plane::PlaneCoordinates()` and `Plane::GlobalCoordinates()` are used by `Polygon::NearestIntersection()` for transforming a three-dimensional planar polygon and plane intersection point to plane coordinates and vice versa.

## 20.4.2 Ray–Polygon, Ray–Triangle and Ray–Quadrilateral Intersections

This section examines the intersection of a ray and a polygon of arbitrary number of vertices lying on a plane. Determining whether a ray intersects a polygon is a relatively straightforward but computationally expensive process, and the implementation of `Polygon::NearestIntersection()` is greatly simplified by the previously described member functions of `class Plane`. Since classes `Triangle` and `Quadrilateral` are both derived from `Polygon`, we are relieved of the task of defining separate `NearestIntersection()` member functions specifically for `Triangle` and `Quadrilateral`. However, it must be observed that a ray–triangle intersection algorithm is more efficient than a ray–polygon intersection algorithm; refer to, for example, Snyder and Barr (1987).

The `class` declaration of `Polygon` is:

```
// poly.h
// class Polygon

#ifndef _POLY_H // prevent multiple includes
#define _POLY_H

#include <limits.h> // INT_MAX

#include "vec3d.h" // class Vector3D
#include "pt.h" // class Point
#include "line.h" // class Line
#include "shape.h" // class Shape
#include "plane.h" // class Plane

class Polygon : public Shape, public Plane
{
private:
    void Allocate (int n) ;
protected:
    Point* verts ;
    Line* edges ;
    int vertices ;
public:
    // constructors
    Polygon (int n=3) ;
    Polygon (Point* p_array, int n_verts, int n=0,
             const Surface& s=Surface()) ;
    Polygon (const Vector<Point>& p_array, int n=0,
             const Surface& s=Surface()) ;
    // copy constructor
    Polygon (const Polygon& p) ;
    // destructor
    ~Polygon () ;
    // member functions
    int Vertices () const { return vertices ; }
    int Edges () const { return vertices ; }
    Point& Vertex (int n) ;
    const Point& Vertex (int n) const ;
}
```

```

Line&           Edge (int n) ;
const Line&   Edge (int n) const ;
// overridden member functions
Vector3D Normal (const Point& p) ;
Point Centroid () ;
double Perimeter () ;
double SurfaceArea () ;
double Volume () ;
BoundingBox BBox () ;
Point NearestIntersection (const Line& l,
                           Boolean& intersect) ;
void Print (ostream& s) ;
// overloaded operator
Polygon& operator = (const Polygon& p) ;
// friend/overloaded operator
friend istream& operator >> (istream& s, Polygon& p) ;
}; // class Polygon

#endif // _POLY_H

```

with corresponding implementation file POLY.CPP.

Most ray–polygon intersection algorithms adopt the method of parallel projection to the polygon vertices onto one of the coordinate axes planes so as to reduce the problem from three dimensions to two dimensions. In the present case, let us use a different approach based on the `Plane::PlaneCoordinates()` member function that `Polygon` inherits from `Plane`. Here's the definition of `Polygon::NearestIntersection()`:

```

Point Polygon::NearestIntersection (const Line& l,
                                   Boolean& intersect)
{
    Point ip ; // intersection point

    // first test if line intersects
    // the polygon plane
    int plane_intersect (2) ;
    const Point plane_ip = Plane::Intersection (l,
                                                plane_intersect) ;

    // no intersection or line parallel to plane
    if (plane_intersect == 0 || plane_intersect == -1)
    {
        intersect = B_FALSE ;
    }
    // line within plane
    else if (plane_intersect == -2)
    {
        // convert polygon and line 3D coor. to plane coor.
        Vector<Point> poly_plane_coor (vertices, Point()) ;
        Line          line_plane_coor ;

        Point l_origin = verts[0] ;

```

```

Vector3D Ip      = verts[1] - verts[0] ;
Ip.Normalise () ; // normalise Ip
Point plane_point ;

// plane polygon
for (int i=1; i<vertices; i++)
{
    PlaneCoordinates (verts[i], l_origin,
                      Ip, plane_point) ;
    poly_plane_coor[i] = plane_point ;
}
// plane line
PlaneCoordinates (l.P0(), l_origin, Ip, plane_point) ;
line_plane_coor.P0() = plane_point ;
PlaneCoordinates (l.P1(), l_origin, Ip, plane_point) ;
line_plane_coor.P1() = plane_point ;

// determine the nearest intersection of the line
// and polygon edges in local coor.
// plane polygon
Polygon poly_plane (poly_plane_coor, vertices) ;
Point l_int_p ; // nearest intersecton point
int int_count (0) ; // no. of intersections
double dist (0.0) ;
for (int j=0; j<poly_plane.Edges(); j++)
{
    // line intersects a polygon edge
    if (poly_plane.Edge(j).Intersection
        (line_plane_coor, l_int_p))
    {
        intersect = B_TRUE ;
        // convert local point to global point
        Point g_int_p = GlobalCoordinates (l_int_p,
                                           Ip, Ip.CrossProduct(normal),
                                           l_origin) ;
        if (int_count == 0)
        {
            // dist. to plane line origin
            dist = line_plane_coor.P0().
                  DistanceBetweenTwoPoints (l_int_p) ;

            ip = g_int_p ;
            int_count++ ;
            continue ; // next edge
        }
        if (int_count > 0 && line_plane_coor.P0().
DistanceBetweenTwoPoints(l_int_p) < dist)
        {
            dist = line_plane_coor.P0().
                  DistanceBetweenTwoPoints (l_int_p) ;
            int_count++ ;
        }
    }
}

```

```

        ip = g_int_p ;
    }
}
}

// if line in polygon plane but no intersection
if (int_count == 0)
    intersect = B_FALSE ;
}

// line out-of-plane intersection
else // plane_intersect=1
{
    // convert polygon and point 3D coor. to plane coor.
    Vector<Point> poly_plane_coor (vertices, Point()) ;
    Point pt_plane_coor ;

    Point l_origin = verts[0] ;
    Vector3D Ip = verts[1] - verts[0] ;
    Ip.Normalise () ; // normalise Ip
    Point plane_point ;

    // plane polygon
    for (int i=1; i<vertices; i++)
    {
        PlaneCoordinates (verts[i], l_origin,
                           Ip, plane_point) ;
        poly_plane_coor[i] = plane_point ;
    }
    // plane point
    PlaneCoordinates (plane_ip, l_origin, Ip, plane_point) ;
    pt_plane_coor = plane_point ;

    // test if plane point is in plane polygon:

    // construct horizontal line from test point
    Line test_line (pt_plane_coor,
                    Point (INT_MAX, pt_plane_coor.Y()), 1 ;
    int int_count (0) ;
    Boolean edge_int (B_FALSE) ;

    for (int j=0; j<vertices; j++)
    {
        // polygon edges
        if (j == vertices-1) // if last vertex
        {
            l.P0 () = poly_plane_coor[j] ;
            l.P1 () = poly_plane_coor[0] ;
        }
        else
        {
            l.P0 () = poly_plane_coor[j] ;
            l.P1 () = poly_plane_coor[j+1] ;
        }
    }
}
```

```

        }
        // test if point lies on polygon edge
        if (l.PointOnLine(pt_plane_coor))
        {
            edge_int = B_TRUE ;
            break ;
        }
        // test if polygon vertex is on test line
        // intersecton point of test line and polygon edge
        Point l_int_p ;
        if (!test_line.PointOnLine(poly_plane_coor[j]))
        {
            if (test_line.Intersection (l, l_int_p))
                int_count++ ;
            }
        }
        // point on polygon edge or inside polygon
        if (edge_int || ((int_count % 2) == 1))
        {
            intersect = B_TRUE ;
            ip = plane_ip ;
        }
        else // point outside polygon
            intersect = B_FALSE ;
    }
    return ip ;
} // NearestIntersection()

```

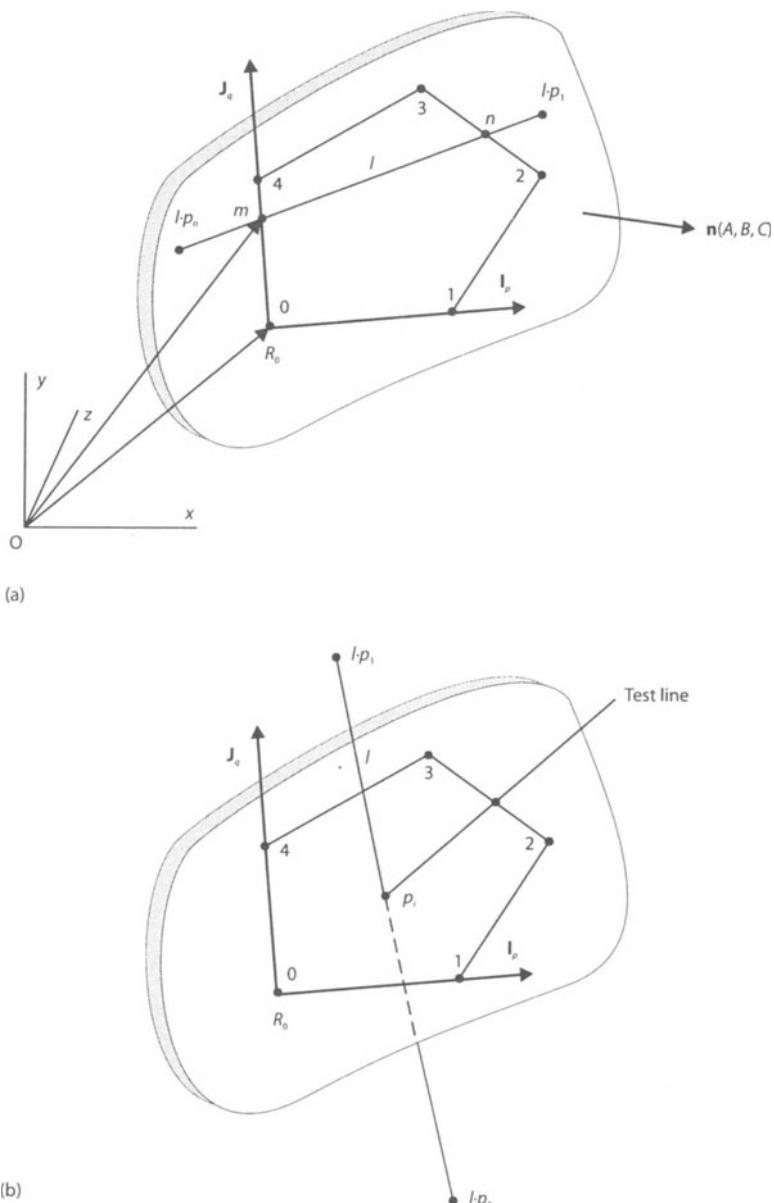
`Polygon` : `:NearestIntersection()` first obtains the intersection point, if any, between a line and a plane using `Plane` : `:Intersection()`. If there is a valid line–plane intersection then the problem reduces to the two separate cases of determining the nearest intersection point, if any, when the line lies within or passes through the polygon plane; see Figs. 20.7(a) and (b). If we consider first the case of the line within the plane, the algorithm proceeds to transform the polygon vertices and line end-points to plane coordinates relative to a plane origin defined as the first polygon vertex. Next, each plane polygon edge is tested for an intersection against the plane line. If an intersection is found then the intersection point is converted to global or world coordinates and compared with the previous intersection point, if any. If multiple intersection points exist then the nearest intersection point is defined as the point having the shortest distance to the origin of global coordinates. In the case of a line passing through the polygon plane, again a local set of plane axes is formed relative to the first two polygon vertices and both polygon vertices and line end-points are converted to plane coordinates. Next, an adaptation of the Jordan curve theorem (refer to `Polygon` : `:PointInPolygon()` of POLY.CPP in Chapter 13) is used to determine whether the line–plane intersection point is inside or outside the polygon.

As mentioned above, the implementations of `Triangle` and `Quadrilateral` are straightforward since they are both derived from `Polygon` and neither derived **class** attempts to redefine `Polygon` : `:NearestIntersection()`. The **class** declarations of `Triangle` and `Quadrilateral` are:

```

// tri.h
// class Triangle

```



**Fig. 20.7** Line–polygon intersections. (a) Line  $l$  lies within the polygon plane. (b) Line  $l$  passes through the polygon plane.

```
#ifndef _TRI_H // prevent multiple includes
#define _TRI_H

#include "pt.h"    // class Point
#include "poly.h"  // class Polygon

class Triangle : public Polygon
{
```

```

public:
    // constructors
    Triangle () ;
    Triangle (Point* p_array, int n=0,
               const Surface& s=Surface()) ;
    Triangle (const Vector<Point>& p_array, int n=0,
               const Surface& s=Surface()) ;
    // copy constructor
    Triangle (const Triangle& t) ;
    // destructor
    ~Triangle () ;
    // overloaded operator
    Triangle& operator = (const Triangle& t) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s,
                                    Triangle& t) ;
}; // class Triangle

#ifndef _TRI_H

// quad.h
// class Quadrilateral

#ifndef _QUAD_H // prevent multiple includes
#define _QUAD_H

#include "pt.h"      // class Point
#include "poly.h"    // class Polygon

class Quadrilateral : public Polygon
{
public:
    // constructors
    Quadrilateral () ;
    Quadrilateral (Point* p_array, int n=0) ;
    Quadrilateral (const Vector<Point>& p_array,
                   int n=0) ;
    // copy constructor
    Quadrilateral (const Quadrilateral& q) ;
    // destructor
    ~Quadrilateral () ;
    // overloaded operator
    Quadrilateral& operator = (const Quadrilateral& q) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s,
                                    Quadrilateral& q) ;
}; // class Quadrilateral

#endif // _QUAD_H

```

with corresponding implementation files TRI.CPP and QUAD.CPP.

### 20.4.3 Ray-Tetrahedra Intersection

Figure 20.8 illustrates a line-tetrahedron intersection. Since in the present case a tetrahedron is modelled as four planar triangular faces and **class Tetrahedra** encapsulates a pointer to **Triangle**, the member function **Tetrahedra::NearestIntersection()** simply loops through the four triangular faces testing each face for an intersection point. If multiple intersection points are found then the nearest, relative to the global origin, is returned.

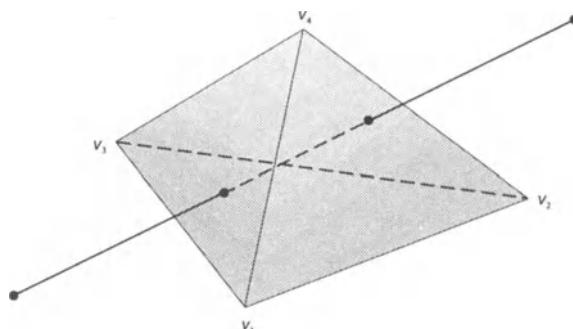
The declaration of **class Tetrahedra** and definition of **Tetrahedra::NearestIntersection()** are to be found in TET.H and TET.CPP respectively and are given below:

```
// tet.h
// class Tetrahedra

#ifndef _TET_H // prevent multiple includes
#define _TET_H

#include "shape.h" // abstract class Shape
#include "tri.h" // class Triangle

class Tetrahedra : public Shape
{
private:
    void Allocate () ;
protected:
    Triangle* faces ;
public:
    // constructors
    Tetrahedra () ;
    Tetrahedra (const Triangle& t0, const Triangle& t1,
                const Triangle& t2, const Triangle& t3,
                int n=0, const Surface& s=Surface()) ;
    // copy constructor
    Tetrahedra (const Tetrahedra& t) ;
    // destructor
    ~Tetrahedra () ;
    // member functions
```



**Fig. 20.8** Line-tetrahedron intersection.

```

Triangle& Face (int n) ;
const Triangle& Face (int n) const ;
Point* Vertices () ;
// overridden member functions
Vector3D Normal (const Point& p) ;
Point Centroid () ;
double Perimeter () ;
double SurfaceArea () ;
double Volume () ;
BoundingBox BBox () ;
Point NearestIntersection (const Line& l,
                           Boolean& intersect) ;
void Print (ostream& s) ;
// overloaded operator
Tetrahedra& operator = (const Tetrahedra& t) ;
// friend/overloaded operator
friend istream& operator >> (istream& s, Tetrahedra& t) ;
}; // class Tetrahedra

#endif // _TET_H

Point Tetrahedra::NearestIntersection (const Line& l,
                                       Boolean& intersect)
{
    Point ip ; // intersection point
    int int_count (0) ; // no. of intersections
    double dist (0.0) ;

    intersect = B_FALSE ;

    // for each triangular face
    for (int i=0; i<4; i++)
    {
        Boolean intersect_face (B_FALSE) ;

        Point int_p = faces[i].NearestIntersection (l,
                                                     intersect_face) ;
        if (intersect_face) // line intersects face
        {
            intersect = B_TRUE ;

            if (int_count == 0)
            {
                // dist. to line origin
                dist = l.P0() .
                    DistanceBetweenTwoPoints (int_p) ;
                int_count++ ;
                ip = int_p ;
                continue ; // next face
            }
            if (int_count > 0 &&

```

```

l.P0().DistanceBetweenTwoPoints(int_p) < dist)
{
    dist = l.P0().
        DistanceBetweenTwoPoints (int_p) ;
    int_count++ ;
    ip = int_p ;
}
}
return ip ;
} // NearestIntersection()

```

#### 20.4.4 Ray–Sphere Intersection

Spherical objects are very popular in raytracing because the ray–sphere intersection algorithm is computationally efficient. Figure 20.9(a) illustrates the ray–sphere configuration. The equation of a sphere of radius  $r$  and centre at point  $p_c(x_c, y_c, z_c)$  is:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

while the parametric equation of a line  $l(p_0(x_0, y_0, z_0), p_1(x_1, y_1, z_1))$  is:

$$x = x_0 + (x_1 - x_0)t = x_0 + x_d t$$

$$y = y_0 + (y_1 - y_0)t = y_0 + y_d t$$

$$z = z_0 + (z_1 - z_0)t = z_0 + z_d t$$

Equating the sphere and line equations we arrive at the following equation in terms of  $t$  for the intersection of a line and a sphere:

$$at^2 + bt + c = 0$$

where  $a, b$  and  $c$  are given by:

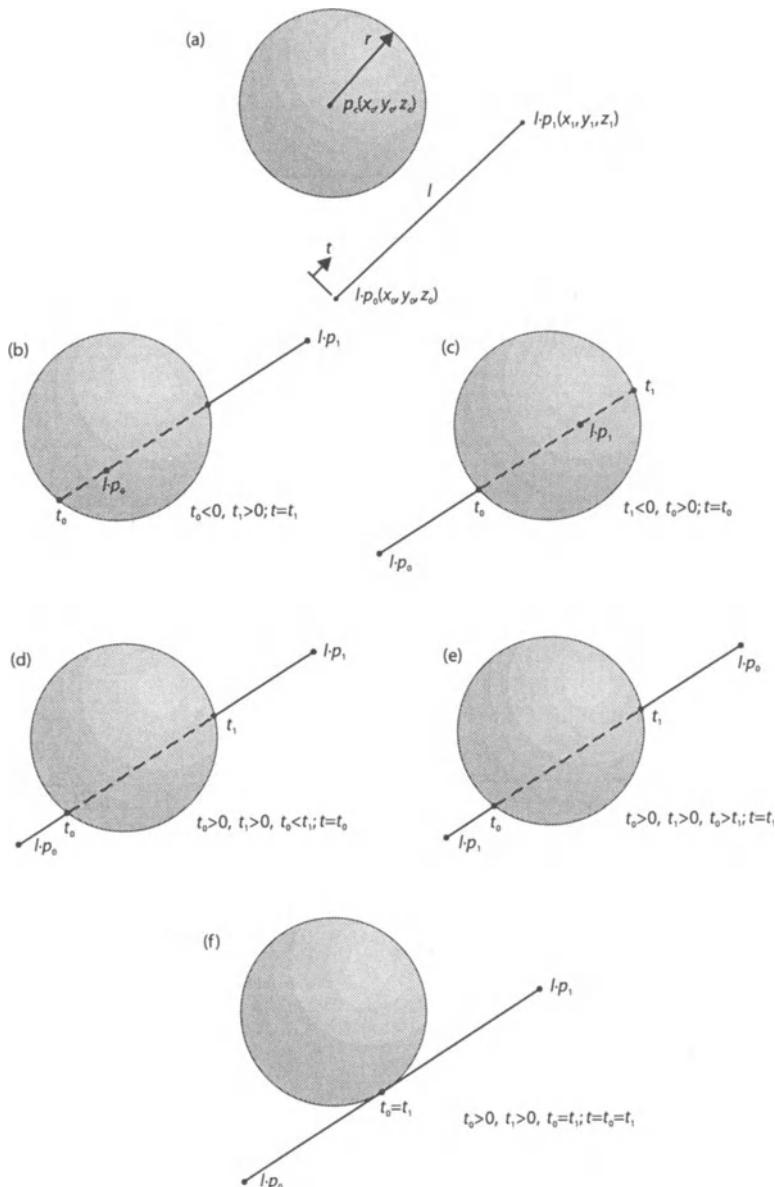
$$a = x_d^2 + y_d^2 + z_d^2 = \mathbf{v} \cdot \mathbf{v}$$

$$b = 2[x_d(x_0 - x_c) + y_d(y_0 - y_c) + z_d(z_0 - z_c)] = -2(\mathbf{p}_c - \mathbf{p}_0) \cdot \mathbf{v}$$

$$c = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2 = \mathbf{v} \cdot \mathbf{v} - r^2$$

where  $\mathbf{v} = \mathbf{p}_1 - \mathbf{p}_0$ .

Solving the above quadratic equation for  $t$  informs us of the six possible line–sphere intersections. If the determinant of the quadratic equation solution is negative then the line does not intersect the sphere; Fig. 20.9(a). Figures 20.9(b) and (c) illustrate a single intersection point with the line start- and end-points inside the sphere respectively. Figures 20.9(d) and (e) illustrate two intersection points with both the line start- and end-points being outside the sphere, and finally Fig. 20.9(f) illustrates a single intersection point when the line tangentially touches the sphere surface.



**Fig. 20.9** Line–sphere intersections. (a) No intersection points, with both the line end-points outside the sphere. (b) One intersection point with the line start point inside the sphere. (c) One intersection point with the line end point inside the sphere. (d) Two intersection points with both line end-points outside the sphere. (e) Two intersection points with both line end-points outside the sphere. (f) One intersection point with the line tangentially touching the surface of the sphere.

The declaration of **class Sphere** and definition of **Sphere::NearestIntersection()** are:

```
// sphere.h
// class Sphere
```

```

#ifndef _SPHERE_H // prevent multiple includes
#define _SPHERE_H

#include <math.h> // atan()

#include "tol.h" // tolerances
#include "pt.h" // class Point
#include "line.h" // class Line
#include "bbox.h" // class BoundingBox
#include "shape.h" // abstract class Shape

class Sphere : public Shape
{
protected:
    Point centre ;
    double radius ;
public:
    Sphere ()
        : Shape (), centre (), radius (0.0) {}
    Sphere (const Point& cent, double r, int n=0,
            const Surface& s=Surface())
        : Shape (n, s), centre (cent), radius (r) {}
    // copy constructor
    Sphere (const Sphere& s)
        : Shape (s.number, s.surface),
          centre (s.centre), radius (s.radius) {};
    // member functions
    double& Radius () { return radius ; }
    const double& Radius () const { return radius ; }
    Point& Centre () { return centre ; }
    const Point& Centre () const { return centre ; }
    // overridden member functions
    Vector3D Normal (const Point& p) ;
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    double Volume () ;
    BoundingBox BBox () ;
    Point NearestIntersection (const Line& l,
                               Boolean& intersect) ;
    void Print (ostream& s) ;
    // overloaded operator
    Sphere& operator = (const Sphere& s) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s, Sphere& sp) ;
};

#endif // _SPHERE_H

```

```
{  
Point ip ; // intersection point  
  
// translate sphere centre  
Vector3D tran = centre - l.P0 () ;  
  
double a = (l.P1()-l.P0()).DotProduct (l.P1()-l.P0()) ;  
double b = -2.0 * tran.DotProduct (l.P1()-l.P0()) ;  
double c = tran.DotProduct (tran) - (radius * radius) ;  
  
double deter = b * b - 4.0 * a * c ;  
  
// ray misses sphere  
if (deter < 0.0)  
{  
    intersect = B_FALSE ;  
    return ip ;  
}  
  
// intersection points  
deter = sqrt (deter) ;  
double t0 = (-b + deter) / (2.0 * a) ;  
double t1 = (-b - deter) / (2.0 * a) ;  
  
// ray misses sphere  
if (t0 < TOLERANCE || t0 > MAX)  
    t0 = -1.0 ;  
if (t1 < TOLERANCE || t1 > MAX)  
    t1 = -1.0 ;  
  
// find nearest intersection  
double t (0.0) ;  
if (t0 < 0.0 && t1 < 0.0) // ray misses  
{  
    intersect = B_FALSE ;  
    return ip ;  
}  
else if (t0 < 0.0 && t1 > 0.0)  
    t = t1 ;  
else if (t1 < 0.0 && t0 > 0.0)  
    t = t0 ;  
else if (t0 < t1)  
    t = t0 ;  
else // t0>0, t1>0, t1<t0  
    t = t1 ;  
  
intersect = B_TRUE ;  
ip.X() = l.P0().X() + ((l.P1().X()-l.P0().X()) * t) ;  
ip.Y() = l.P0().Y() + ((l.P1().Y()-l.P0().Y()) * t) ;  
ip.Z() = l.P0().Z() + ((l.P1().Z()-l.P0().Z()) * t) ;  
return ip ;
```

---

```
    } // NearestIntersection()
```

### 20.4.5 Ray–Circle Intersection

Consider the intersection of a line segment and a circle which lies within a plane; see Figs. 20.10(a) and (b). Because the circle lies within a plane the problem of determining the point or points of intersection essentially reduces to the two separate cases of the line passing through the circle plane or the line lying within the circle plane. If we consider the simpler case first, of the line passing through the circle plane, the line intersects the circle if the distance between the circle centre,  $p_c(x_c, y_c, z_c)$ , and the plane intersection point,  $p_i(x_i, y_i, z_i)$ , is less than the circle radius  $r$ . If the line lies within the circle plane we can determine the point or points of intersection by representing the circle in terms of plane coordinates  $p$  and  $q$ :

$$(p - x_c)^2 + (q - y_c)^2 = r^2$$

and the line segment with parametric variable  $t$  as:

$$p = x_0 + (x_1 - x_0)t = x_0 + x_d t$$

$$q = y_0 + (y_1 - y_0)t = y_0 + y_d t$$

Equating the circle and line equations and solving for  $t$ :

$$t = \frac{1}{x_d^2 + y_d^2} (x_d(x_c - x_0) + y_d(y_c - y_0) \pm \{r^2(x_d^2 + y_d^2) - [x_d(y_0 - y_c) - y_d(x_0 - x_c)]\}^{1/2})$$

This equation is greatly simplified if we translate the origin of plane coordinates to the line start-point and make the plane axis vector  $I_p$  coincide with the line segment;  $y_d=0$  and  $(x_0, y_0)=(0, 0)$ :

$$t = \frac{1}{x_d} [x_c \pm (r^2 - y_c^2)^{1/2}]$$

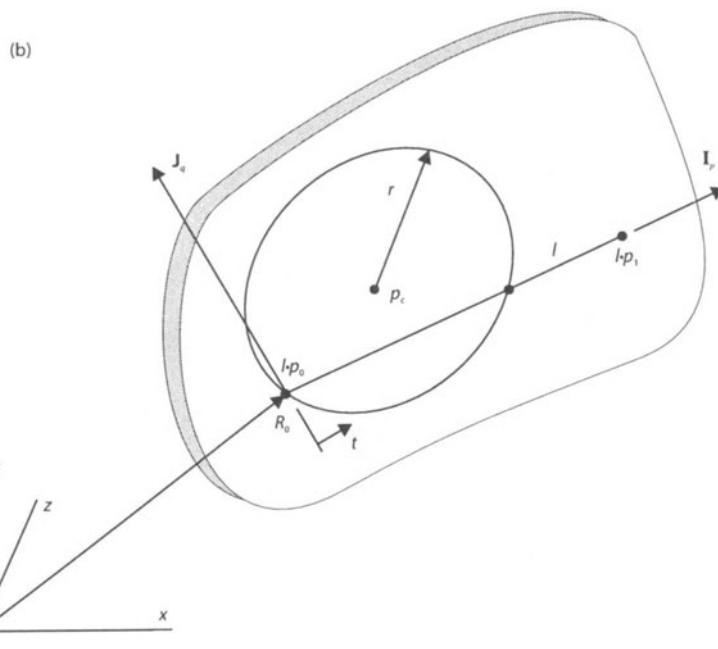
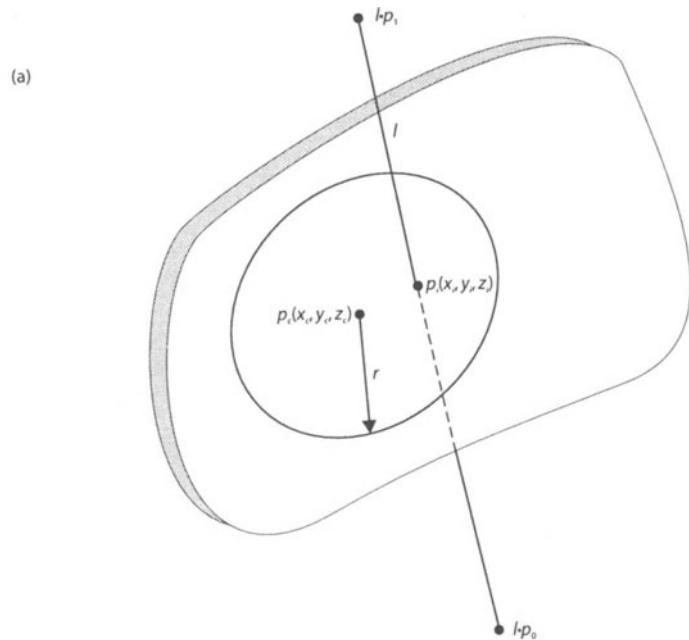
This expression can give no solutions if the determinant is negative, two values for  $t$  or a single value for  $t$  if the line tangentially touches the circle circumference. If two intersection points occur then the nearest, relative to the global origin, is chosen.

The declaration of `class Circle` (CIRC.H) and definition of `Circle::NearestIntersection()` (CIRC.CPP) are given below:

```
// circ.h
// header file for Circle class

#ifndef _CIRC_H // prevent multiple includes
#define _CIRC_H

#include <math.h> // atan()
#include "pt.h" // class Point
#include "shape.h" // abstract class Shape
```



**Fig. 20.10** Line–circle intersections. (a) Line  $l$  passes through the circle plane. (b) Line  $l$  lies within the circle plane.

```

#include "plane.h" // class Plane

class Circle : public Shape, public Plane
{
protected:
    Point centre ;
    double radius ;
public:
    Circle ()
        : Shape (), Plane (), centre (), radius (0.0) {}
    Circle (double a, double b, double c,
             const Point& cent, double r, int n=0,
             const Surface& s=Surface())
        : Shape (n, s),
          Plane (a, b, c, -(cent.X())*normal[0] +
                  cent.Y()*normal[1] +
                  cent.Z()*normal[2])),
          centre (cent), radius (r) {}
    Circle (const Vector3D& normal, const Point& cent,
             double r, int n=0, const Surface& s=Surface())
        : Shape (n, s),
          Plane (normal, -(cent.X())*normal[0] +
                  cent.Y()*normal[1] +
                  cent.Z()*normal[2])),
          centre (cent), radius (r) {}
    // copy constructor
    Circle (const Circle& c)
        : Shape (c.number, c.surface),
          Plane (c.normal, c.d),
          centre (c.centre), radius (c.radius) {} ;
    // member functions
    double& Radius () { return radius ; }
    const double& Radius () const { return radius ; }
    Point& Centre () { return centre ; }
    const Point& Centre () const { return centre ; }
    // overridden member functions
    Vector3D Normal (const Point& p) ;
    Point Centroid () ;
    double Perimeter () ;
    double SurfaceArea () ;
    double Volume () ;
    BoundingBox BBox () ;
    Point NearestIntersection (const Line& l,
                               Boolean& intersect) ;
    void Print (ostream& s) ;
    // overloaded operator
    Circle& operator = (const Circle& c) ;
    // friend/overloaded operator
    friend istream& operator >> (istream& s, Circle& c) ;
}; // class Circle

```

```
#endif // _CIRC_H

Point Circle::NearestIntersection (const Line& l,
                                    Boolean& intersect)
{
    Point ip ; // intersection point

    // first test if line intersects the circle plane
    int plane_intersect (2) ;
    Point plane_ip = Plane::Intersection (l, plane_intersect) ;

    // no intersection or line parallel to plane
    if (plane_intersect == 0 || plane_intersect == -1)
    {
        intersect = B_FALSE ;
    }
    // line within plane
    else if (plane_intersect == -2)
    {
        // use the line to define Ip since
        // we know the line lies in the plane!
        Point l_origin = l.P0 () ;
        Vector3D Ip = l.P1 () - l.P0 () ;
        Ip.Normalise () ; // normalise Ip
        Point plane_point ;

        // convert circle centre and line 3D coor. to plane coor.
        PlaneCoordinates (centre, l_origin, Ip, plane_point) ;
        Point circ_centre_plane_coor = plane_point ;
        Line line_plane_coor ;
        line_plane_coor.P0() = Point (0.0, 0.0, 0.0) ;
        line_plane_coor.P1() = Point (l.Length(), 0.0, 0.0) ;

        // find planar point/s of intersection
        double deter = radius*radius -
                      circ_centre_plane_coor.Y() *
                      circ_centre_plane_coor.Y() ;
        if (deter < 0.0) // no intersection
        {
            intersect = B_FALSE ;
        }
        // tangential intersection
        else if (fabs(deter) < TOLERANCE)
        {
            Point l_int_p ; // intersection point
            double t = circ_centre_plane_coor.X() /
                       line_plane_coor.P1().X() ;
            l_int_p.X () = line_plane_coor.P1().X() * t ;
            ip = GlobalCoordinates (l_int_p, Ip,
                                   Ip.CrossProduct(normal), l_origin) ;
            intersect = B_TRUE ;
        }
    }
}
```

```

        }
    else // two intersections
    {
        Point l_int_p ; // intersection point
        double root = sqrt (deter) ;
        double t1 = (circ_centre_plane_coor.X() + root) /
                    line_plane_coor.P1().X() ;
        double t2 = (circ_centre_plane_coor.X() - root) /
                    line_plane_coor.P1().X() ;

        if (fabs(t1) < fabs(t2)) // t1 nearest
        {
            l_int_p.X () = line_plane_coor.P1().X()*t1 ;
            ip = GlobalCoordinates (l_int_p, Ip,
                Ip.CrossProduct(normal), l_origin) ;
        }
        else // t2 nearest
        {
            l_int_p.X () = line_plane_coor.P1().X()*t2 ;
            ip = GlobalCoordinates (l_int_p, Ip,
                Ip.CrossProduct(normal), l_origin) ;
        }
        intersect = B_TRUE ;
    }
}
// line out-of-plane intersection (plane_intersect=1)
else
{
    // if norm between plane point and centre
    // is less than radius
    if (plane_ip.DistanceBetweenTwoPoints(centre) < radius)
    {
        intersect = B_TRUE ;
        ip = plane_ip ;
    }
    else
        intersect = B_FALSE ;
}
return ip ;
} // NearestIntersection()

```

#### 20.4.6 Other Ray–Object Intersections

The list of objects discussed above is far from exhaustive, and the majority of raytracing programs generally include an extensive library of object surfaces. Algebraic and quadric implicit surfaces are frequently modelled. Quadric surfaces are a specialised form of algebraic surface and include such surfaces as cylinders, cones, ellipsoids, paraboloids and hyperboloids. Typical algebraic surfaces are the torus and Steiner's surface. Apart from the sphere the only solid object discussed above was the tetrahedron. Other popular solid objects are a cube or hexahedron and general polyhedra. The polygon, triangle, quadrilateral, circle and tetrahedron objects discussed above were all based on a planar surface. A natural extension would be to

define such objects on non-planar implicit and explicit surfaces. Complex surfaces can also be formed by specifying a patch of primitive surfaces. Further, the restriction that the edges of polyhedra be straight could easily be relaxed by representing a line as a set of points or by a curve.

The raytracing algorithm integrates well with the *set-theoretic geometric* or *constructive solid geometric* modelling technique. Set-theoretic geometry allows complex objects to be generated by performing Boolean operations on primitive objects. Typical Boolean operations are union, intersection, difference and symmetric difference. For an excellent introduction to set-theoretic geometry and the SVLIS modeller refer to Bowyer (1995).

## 20.5 The Viewer

A viewer is modelled in terms of the position, or *at* point, relative to the global origin of coordinates and two vectors called the *looking at* and *up* vectors; see Fig. 20.11. In the present raytracing program a viewer is defined purely in terms of the *at* point with the *looking at* and *up* vectors being redundant. The *looking at* and *up* vectors enable a set of axes to be defined at the viewer, the orientation of the viewing plane to be defined, and ultimately the viewing frustum to be defined. However, it will be shown later that the present implementation explicitly defines the viewing plane with only the viewer's position being required.

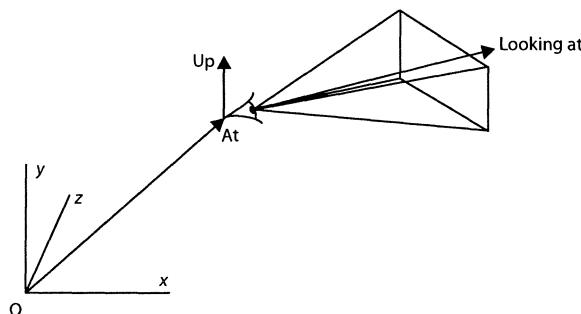
The declaration of **class** *Viewer* can be found in *VIEWER.H* with corresponding implementation *VIEWER.CPP*:

```
// viewer.h
// class Viewer

#ifndef _VIEWER_H // prevent multiple includes
#define _VIEWER_H

#include "pt.h"      // class Point
#include "vec3d.h"   // class Vector3D

class Viewer
{
private:
    Point    at ;
    Vector3D looking_at ;
```



**Fig. 20.11** A viewer.

```

    Vector3D up ;
public:
    // constructors
    Viewer ()
        : at (Point()), looking_at (Vector3D(0.0,0.0,1.0)),
          up (Vector3D(0.0,1.0,0.0)) {}
    Viewer (const Point& a, const Vector3D& l,
            const Vector3D& u)
        : at (a), looking_at (l), up (u) {}
    Viewer (const Viewer& v)
        : at (v.at), looking_at (v.looking_at),
          up (v.up) {}
    // member functions
    const Point& At () const { return at ; }
    Point& At () { return at ; }
    Vector3D& LookingAt () { return looking_at ; }
    Vector3D& Up () { return up ; }
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const Viewer& v) ;
    friend istream& operator >> (istream& s, Viewer& v) ;
}; // class Viewer

#endif // class Viewer

```

## 20.6 The View Plane and Screen

Figures 20.12(a) and (b) illustrate, respectively, the view plane and display screen coordinate systems. A view plane is characterised in terms of the plane normal  $\mathbf{n}$ , plane origin  $R_0$  relative to the global origin, and plane axes vectors  $\mathbf{I}_p$  and  $\mathbf{J}_q$ . The view plane contains a rectangular window defined by  $Vx_l, Vx_r, Vy_b$  and  $Vy_t$ . It is assumed that a point  $p_v(x_v, y_v)$  within the view plane is restricted to the view plane window area. Similarly, the display screen contains a rectangular window defined by  $Sx_l, Sx_r, Sy_b$  and  $Sy_t$  with a display screen point  $p_s(x_s, y_s)$  restricted to the display screen window.

A frequent requirement when dealing with view plane and display screen windows is the ability to map or transform a point from one window to another. Indeed, in the implementation of the `World::GenerateRayTracedImage()` member function discussed later the coordinates of a view plane point (through which a ray is traced from the viewer) are determined by transforming the known coordinates of a display screen pixel to the view plane window. A linear transformation is easily found if we equate the abscissa and ordinate lengths of the two points  $p_v$  and  $p_s$  within their respective windows:

$$\frac{x_s - Sx_l}{Sx_r - Sx_l} = \frac{x_v - Vx_l}{Vx_r - Vx_l}$$

$$\frac{y_s - Sy_b}{Sy_t - Sy_b} = \frac{y_v - Vy_b}{Vy_t - Vy_b}$$

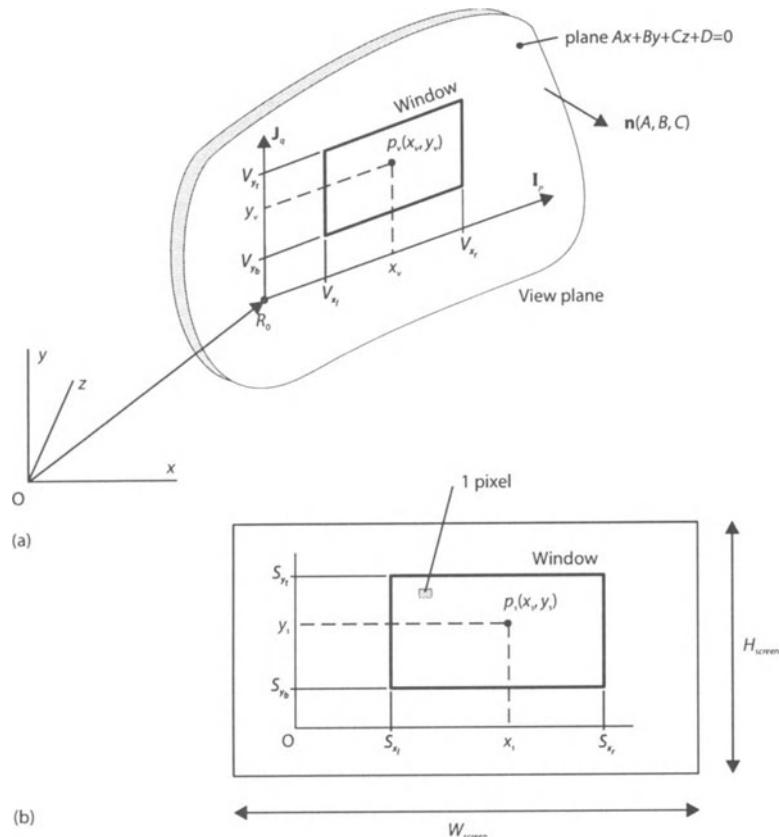


Fig. 20.12 (a) View plane.(b) Display screen.

Solving for  $x_s$  and  $y_s$ :

$$x_s = \frac{x_v - Vx_l}{Vx_r - Vx_l} (Sx_r - Sx_l) + Vx_l$$

$$y_s = \frac{y_v - Vy_b}{Vy_t - Vy_b} (Sy_t - Sy_b) + Vy_b$$

and similarly for  $x_v$  and  $y_v$ .

Observing that a rectangular window is common to both the view plane and display screen, let's declare a **template class** called `RectWindow` which encapsulates the left, right, top and bottom dimensions of a rectangular window and facilitates transforming from one window to another:

```
// rect_win.h
// template class RectWindow

#ifndef _RECT_WIN_H // prevent multiple includes
#define _RECT_WIN_H
```

```

#include "pt.h" // class Point

template <class T>
class RectWindow
{
private:
    T left, right, top, bottom ;
public:
    // constructor
    RectWindow ()
        : left (), right (), top (), bottom () {}
    RectWindow (T l, T r, T t, T b)
        : left (l), right (r), top (t), bottom (b) {}
    RectWindow (const Point& near_pt, const Point& far_pt)
        : left (near_pt.X()), right (far_pt.X()),
          top (far_pt.Y()), bottom (near_pt.Y()) {}
    RectWindow (const RectWindow& rw)
        : left (rw.left), right (rw.right),
          top (rw.top), bottom (rw.bottom) {}
    // member functions
    T Left () const { return left ; }
    T Right () const { return right ; }
    T Top () const { return top ; }
    T Bottom () const { return bottom ; }
    Point Nearest () { return Point (left, bottom) ; }
    Point Furthest () { return Point (right, top) ; }
    T Width () { return right - left ; }
    T Height () { return top - bottom ; }
    Point Origin () { return Point (left, bottom) ; }
    Point Transform (const RectWindow& rw, const Point& pt) ;
    T XScaleFactor (const RectWindow& rw) ;
    T YScaleFactor (const RectWindow& rw) ;
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const RectWindow<T>& rw) ;
    friend istream& operator >> (istream& s,
                                    RectWindow<T>& rw) ;
}; // class RectWindow

template <class T>
Point RectWindow<T>::Transform (const RectWindow<T>& rw,
                                const Point& pt)
{
    Point p ;
    p.X () = ((pt.X()-rw.Left())*(Right()-Left())) /
              (rw.Right()-rw.Left()) + Left() ;
    p.Y () = ((pt.Y()-rw.Bottom())*(Top()-Bottom())) /
              (rw.Top()-rw.Bottom()) + Bottom() ;
    return p ;
}

```

```

template <class T>
T RectWindow<T>::XScaleFactor (const RectWindow& rw)
{
return (rw.Right()-rw.Left()) / (Right()-Left()) ;
}

template <class T>
T RectWindow<T>::YScaleFactor (const RectWindow& rw)
{
return (rw.Top()-rw.Bottom()) / (Top()-Bottom()) ;
}

// friends/overloaded operators
template <class T>
ostream& operator << (ostream& s,
const RectWindow<T>& rw)
{
return s << "[" << rw.left << ", " << rw.right
    << ", " << rw.top << ", " << rw.bottom << "]" ;
}

template <class T>
istream& operator >> (istream& s, RectWindow<T>& rw)
{
char left ('('), right (')'), comma (',') ;
return s >> ws
    >> left
    >> rw.left >> comma >> rw.right >> comma
    >> rw.top >> comma >> rw.bottom
    >> right ;
}

#ifndef _RECT_WIN_H

```

RectWindow is declared a **template class** because view plane coordinates generally require floating-point accuracy whereas display screen coordinates (in terms of pixels) require integer coordinates. This is demonstrated below in the RectWindow **private** data members of classes ViewPlane and Screen.

The above-outlined procedure of transforming a point in one window to another window is implemented in RectWindow: *Transform()*. The member functions *XScaleFactor()* and *YScaleFactor()* return the following *x* and *y* scale factors:

$$x_{scale} = \frac{Sx_r - Sx_l}{Vx_r - Vx_l}$$

$$y_{scale} = \frac{Sy_t - Sy_b}{Vy_t - Vy_b}$$

It is important to note that an image of points will be distorted during transformation from one window to another if  $x_{scale}$  is not equal to  $y_{scale}$ .

Having described `RectWindow` we are now in a position to take a look at the view plane and display screen classes. First, `ViewPlane`:

```
// view_pl.h
// class ViewPlane

#ifndef _VIEW_PL_H // prevent multiple includes
#define _VIEW_PL_H

#include "pt.h"           // class Point
#include "vec3d.h"         // class Vector3D
#include "rect_win.h"      // template class RectWindow
#include "plane.h"         // class Plane

class ViewPlane : public Plane
{
private:
    Point      origin ;
    Vector3D   ip, jq ;
    // Kr given by inherited Plane::normal
    RectWindow<double> win ;    // view plane rect.
                                // window

public:
    // constructors
    ViewPlane ()
        : Plane (), origin (), ip (), jq (), win () {}
    ViewPlane (const Vector3D& n, const Point& o,
               const Vector3D& ip_arg,
               const Vector3D& jq_arg,
               const RectWindow<double>& w)
        : Plane (n, o.DistanceBetweenTwoPoints()),
          origin (o), ip (ip_arg), jq (jq_arg), win (w) {}
    ViewPlane (const ViewPlane& vp)
        : Plane (vp), origin (vp.origin), ip (vp.ip),
          jq (vp.jq), win (vp.win) {}

    // member functions
    Point& Origin () { return origin ; }
    Vector3D& Ip () { return ip ; }
    Vector3D& Jq () { return jq ; }
    Vector3D& Kr () { return normal ; }
    RectWindow<double> Window () const { return win ; }
    Point GlobalCoordinates (const Point& l_p) ;
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const ViewPlane& vp) ;
    friend istream& operator >> (istream& s,
                                    ViewPlane& vp) ;
}; // class ViewPlane

#endif // _VIEW_PL_H
```

with implementation file VIEW\_PL.CPP. Observe that ViewPlane is derived from Plane and therefore inherits the Plane::normal data member. The declaration of **class** Screen is:

```
// screen.h
// class Screen

#ifndef _SCREEN_H // prevent multiple includes
#define _SCREEN_H

#include "rect_win.h" // class RectWindow

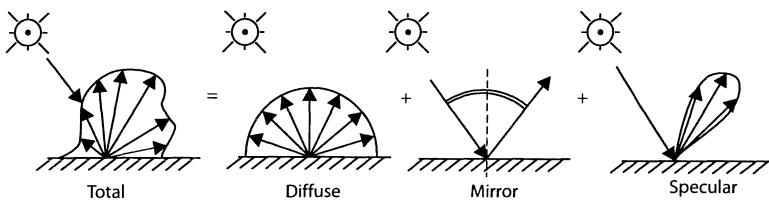
class Screen
{
private:
    RectWindow<int> win ;
    int width, height ;
public:
    // constructor
    Screen ()
        : win (0, 640, 480, 0),
          width (640), height (480) {}
    Screen (const RectWindow<int>& rw, int w=640, int h=480)
        : win (rw), width (w), height (h) {}
    // member functions
    RectWindow<int> Window () const { return win ; }
    int& Width () { return width ; }
    int& Height () { return height ; }
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const Screen& scr) ;
    friend istream& operator >> (istream& s, Screen& scr) ;
}; // class Screen

#endif // _SCREEN_H
```

and corresponding implementation file SCREEN.CPP. The default display screen width and height sizes are 640 and 480 respectively.

## 20.7 Object Surfaces, Illumination and Reflectance

In this section we shall examine five different components of lighting which are frequently used when generating a raytraced image to produce realistic object illumination effects. The models of light chosen are simplified empirical models which are easily implemented but can, nevertheless, produce realistic illumination effects. The different components of light modelled are *ambient*, *diffuse*, *specular*, *reflected* and *transmitted*, with each component having its respective coefficient, such as  $k_a$  or  $k_d$ , for a given object. Figure 20.13 illustrates the total reflected light from a surface separated into the three components of diffuse, perfect mirror and specular reflection. For a more detailed discussion of illumination effects refer to Watt and Watt (1992).



**Fig. 20.13** Diffuse, mirror and specular reflective components.

Each separate light component and its implementation is discussed in detail below and encapsulated within a **class** called **Surface**:

```
// surf.h
// class Surface

#ifndef _SURF_H // prevent multiple includes
#define _SURF_H

#include "cplx.h"      // class Complex
#include "pt.h"        // class Point
#include "vec3d.h"      // class Vector3D
#include "light.h"      // class PointLight
#include "viewer.h"     // class Viewer
#include "rgb.h"        // class RGBColour, ...

class Surface
{
private:
    NormalisedRGBColour colour ;
    float ka, kd, ks, kr, kt ; // coefficients
    float ns ; // specular reflection exponent
    float ri ; // index of refraction
public:
    Surface ()
        : colour (1.0, 1.0, 1.0),
          ka (0.0), kd (0.0), ks (0.0), kr (0.0), kt (0.0),
          ns (1.0), ri (1.0) {}
    Surface (const NormalisedRGBColour& c,
              float ka_arg, float kd_arg, float ks_arg,
              float kr_arg, float kt_arg,
              float ns_arg, float ri_arg)
        : colour (c),
          ka (ka_arg), kd (kd_arg), ks (ks_arg),
          kr (kr_arg), kt (kt_arg),
          ns (ns_arg), ri (ri_arg) {}
    Surface (const Surface& s)
        : colour (s.colour),
          ka (s.ka), kd (s.kd), ks (s.ks),
          kr (s.kr), kt (s.kt),
          ns (s.ns), ri (s.ri) {}
```

```

// member functions
const NormalisedRGBColour Colour () const
{ return colour ; }
NormalisedRGBColour Colour () { return colour ; }
float AmbientCoeff () const { return ka ; }
float DiffuseCoeff () const { return kd ; }
float SpecularCoeff () const { return ks ; }
float ReflectiveCoeff () const { return kr ; }
float TransmittedCoeff () const { return kt ; }
float SpecularExponent () const { return ns ; }
float RefractiveIndex () const { return ri ; }
NormalisedRGBColour AmbientColour ();
NormalisedRGBColour DiffuseColour (const Point& p,
const Vector3D& n,
const PointLight& pl, const Viewer& viewer) ;
NormalisedRGBColour SpecularColour (const Point& p,
const Vector3D& n,
const PointLight& pl, const Viewer& viewer) ;
Vector3D ReflectedRay (const Vector3D& i,
const Vector3D& n) ;
Vector3D TransmittedRay (const Vector3D& i,
const Vector3D& n,
float incident_ri=1.0) ;

// friends/overloaded operators
friend ostream& operator << (ostream& s,
const Surface& surf) ;
friend istream& operator >> (istream& s, Surface& surf) ;
}; // class Surface

#endif // _SURF_H

```

Before discussing the different components of object illumination and the corresponding data members and member functions of **class** Surface, we shall first take a look at the light source model used. The section concludes with a discussion of distance attenuation and shadows.

### 20.7.1 Light Source

Only a single point light source is considered in the present case, in which light rays from the point source are equally radially divergent in all directions. The **class** which encapsulates a single point light source is PointLight and is declared in LIGHT.H:

```

// light.h
// class PointLight

#ifndef _LIGHT_H // prevent multiple includes
#define _LIGHT_H

#include <math.h> // exp()

#include "pt.h" // class Point

```

```
#include "rgb.h" // class RGBColour, ...

class PointLight
{
protected:
    Point at ;
    NormalisedRGBColour colour ;
public:
    PointLight ()
        : at (), colour (1.0, 1.0, 1.0) {}
    PointLight (const Point& light_at,
                const NormalisedRGBColour& light_colour=
                    NormalisedRGBColour(1.0,1.0,1.0))
        : at (light_at), colour (light_colour) {}
    // member functions
    Point At () { return at ; }
    const Point At () const { return at ; }
    NormalisedRGBColour Colour () { return colour ; }
    const NormalisedRGBColour Colour () const
        { return colour ; }
    double DistanceAttenuation (double d, double k=0.01) ;
    // friends/overloaded operators
    friend ostream& operator << (ostream& s,
                                    const PointLight& pl) ;
    friend istream& operator >> (istream& s,
                                    PointLight& pl) ;
}; // class PointLight

#endif // _LIGHT_H
```

and implementation file LIGHT.CPP. A PointLight object is defined purely in terms of a light's location (relative to the global coordinate system) and RGB colour. By default, the light is positioned at the global origin and is white. The *DistanceAttenuation()* member function will be discussed in a later section.

Although only a single point light source is modelled in the present study there are other models of light sources, such as distributed (e.g. a rectangular strip of light such as a fluorescent tube) or directed (e.g. a cone of light, such as a spotlight). Equally, the raytracing program could cater for multiple point light sources.

## 20.7.2 Ambient Light

The ambient component of light approximates the diffusely reflected light from unknown objects in the scene as constant illumination from all directions. The ambient component is independent of the viewer and light source, is constant over the surface of an object and is given by:

$$k_a I_a$$

where  $k_a$  ( $0 \leq k_a \leq 1$ ) is the coefficient of ambient reflection and  $I_a$  is the intensity of ambient light. Each object has an associated  $k_a$  value to control the degree of object ambient illumination and  $I_a$  is a constant for the world domain.

The implementation of computing an object's ambient light intensity is:

```
NormalisedRGBColour
Surface::AmbientColour (const NormalisedRGBColour& ia)
{
    return colour * ka * ia ;
}
```

where colour and ka are data members of Surface and model an object's colour and coefficient of ambient reflection respectively. The argument, ia, will be assigned the value of the World::ambient data member which models the ambient light in the world domain.

Figure 20.14(b) illustrates the greyscale equivalent of a scene generated by the raytracing program, which consists of a sphere of colour red with  $k_a=0.7$  and white ambient light conditions (i.e.  $I_a=(1,1,1)$ ) for normalised RGB intensities. For comparison, Fig. 20.14(a) illustrates the same object with no ambient light, in which the  $k_a$  coefficient is zero.

### 20.7.3 Diffuse Reflection

Object illumination due to diffuse reflection is based on Lambert's cosine law, which states that the radiant energy from a surface in a direction  $\theta$  relative to the surface normal,  $\mathbf{n}$ , is proportional to  $\cos \theta$ ; see Fig. 20.15. Diffuse reflections are assumed to have equal intensity in all directions independent of the incident direction of the light source,  $\mathbf{l}$ , and are therefore independent of the viewer's position. If  $I_l$  denotes the intensity of the point light source and  $k_d$  ( $0 \leq k_d \leq 1$ ) denotes the object surface diffuse reflection coefficient, then the illumination component due to diffuse reflection is:

$$k_d I_l \cos\theta = k_d I_l (\mathbf{n} \cdot \mathbf{l})$$

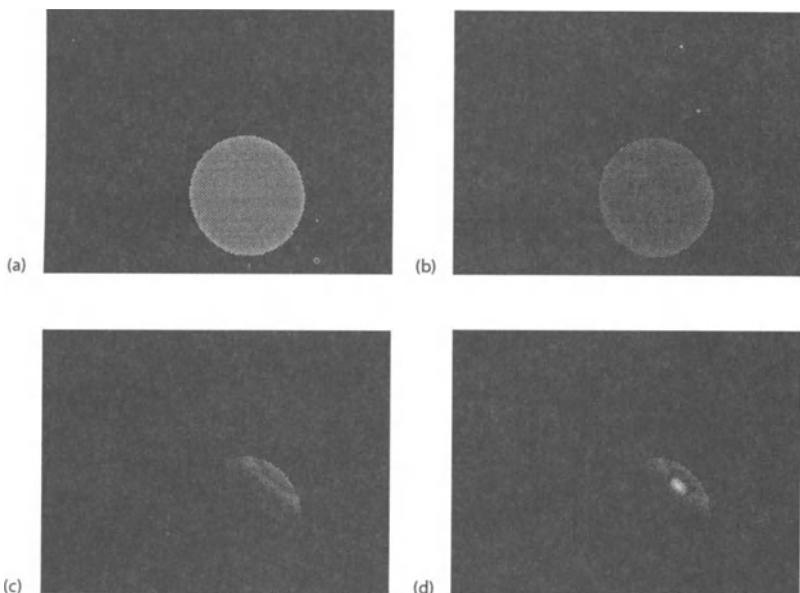
This equation informs us that if the direction of incident light coincides with the surface normal then the surface is fully illuminated. As  $\theta$  increases from zero to  $90^\circ$  the intensity of the diffuse reflection steadily decays to zero. If  $\cos \theta$  is negative then the light source is behind the surface.

The above model of diffuse reflection is implemented in the member function Surface::DiffuseColour():

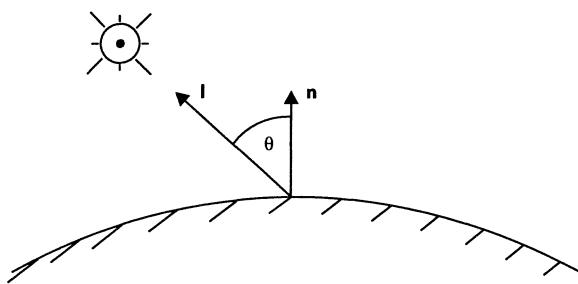
```
NormalisedRGBColour Surface::DiffuseColour (const Point& p,
                                             const Vector3D& n,
                                             const PointLight& pl,
                                             const Viewer& viewer)
{
    NormalisedRGBColour col ;

    if (fabs(kd) > TOLERANCE)
    {
        // intersection point to light source
        Vector3D l = pl.At () - p ;
        double d = l.Norm () ;
        l.Normalise () ;

        // if object normal is pointing away from viewer
        // then flip normal
```



**Fig. 20.14** A raytraced scene illustrating different object illumination and reflectance components. (a) Object colour only. (b) Ambient lighting with white ambient light,  $I_a = (1, 1, 1)$  and  $k_a = 0.7$ . With white ambient light ( $I_a = (1, 1, 1)$  and  $k_a = 0.7$ ) and a single white point light source,  $I_l = (1, 1, 1)$ , (c) illustrates ambient light and diffuse reflection for  $k_a = 0.7$  and  $k_d = 0.6$  and (d) illustrates ambient light and diffuse and specular reflection for  $k_a = 0.7$ ,  $k_d = 0.6$ ,  $k_s = 0.5$  and  $n_s = 25$ .



**Fig. 20.15** Incident angle  $\theta$  between the point light source vector  $\mathbf{l}$  and surface normal  $\mathbf{n}$ .

```

Vector3D normal = n ;
if (normal.DotProduct(viewer.At()) < 0.0)
    normal = -normal ;

// if object normal is pointing away
// from the light source then the diffuse component is 0
double ldn = l.DotProduct (normal) ;
if (ldn < 0.0)
    col = NormalisedRGBColour (0.0, 0.0, 0.0) ;
else
{
```

```

    // account for distance attenuation
    col = pl.Colour () * kd * ldn *
          pl.DistanceAttenuation (d) ;
    }
}
return col ;
}

```

*DiffuseColour()* is passed the surface point at which the diffuse component is to be computed, the surface normal, the point light source and the viewer. Although diffuse reflection is independent of the viewer's position, we still require the position of the viewer to determine the correct direction of the surface normal. If the point light source is behind the surface then the diffuse component is assumed to be zero. The attenuation due to the distance between the surface point and the point light source is discussed later.

Figure 20.14(c) illustrates the contribution of diffuse reflections to a raytraced scene.

#### 20.7.4 Specular Reflection

Specular reflection models the highlights observed on shiny surfaces, and the model adopted is the empirical specular reflection model of Phong (1975):

$$k_s I_l \cos^{n_s} \phi = k_s I_l (\mathbf{r} \cdot \mathbf{v})^{n_s}$$

where  $k_s$  ( $0 \leq k_s \leq 1$ ) is the coefficient of specular reflection (assumed to be constant) and  $I_l$  is the intensity of the point light source. It is worth noting that  $k_s$  is independent of wavelength, and the colour of the specular highlight is therefore that of the light source. Figure 20.16 illustrates the vectors used in the Phong model. Vector  $\mathbf{r}$  represents the direction of an ideal mirror reflection (angle of reflection  $\theta_r$  is equal to the angle of incidence  $\theta_i$ ) of the incident light source vector  $\mathbf{l}$ , while  $\mathbf{v}$  is the direction to the viewer's position from the surface point at which specular illumination is being determined. The angle between  $\mathbf{r}$  and  $\mathbf{v}$  is denoted by  $\phi$ .

The specular reflection index,  $n_s$  ( $1 \leq n_s \leq \infty$ ), models the type of specular reflection. A matt surface is modelled by using a small value for  $n_s$ , whereas a shiny surface is modelled by using a large value for  $n_s$ .

Phong's specular reflection model is implemented as follows:

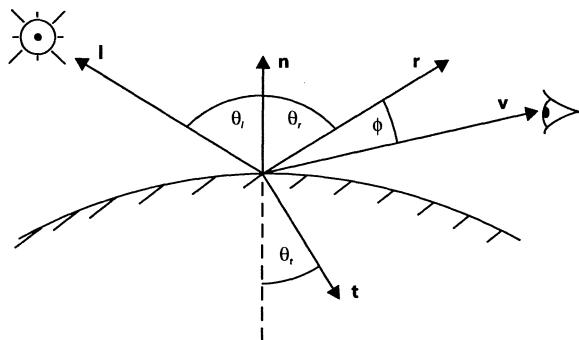


Fig. 20.16 Mirror reflection and refraction.

```

NormalisedRGBColour Surface::SpecularColour (const Point& p,
                                             const Vector3D& n,
                                             const PointLight& pl,
                                             const Viewer& viewer)
{
    NormalisedRGBColour col ;

    if (fabs(ks) > TOLERANCE)
    {
        // intersection point to light source
        Vector3D l = pl.At () - p;
        double d = l.Norm () ;
        l.Normalise () ;

        // if object normal is pointing away from viewer
        // then flip normal
        Vector3D normal = n ;
        if (normal.DotProduct(viewer.At()) < 0.0)
            normal = -normal ;

        // if object normal is pointing away
        // from the light source then the specular component
        // is 0
        double ld़n = l.DotProduct (normal) ;
        if (ld़n < 0.0)
            col = NormalisedRGBColour (0.0, 0.0, 0.0) ;
        else
        {
            Vector3D r = ReflectedRay (l, normal) ;

            Vector3D v = viewer.At () - p ;
            v.Normalise () ;

            // account for distance
            // attenuation
            col = pl.Colour () * ks * pow (r.DotProduct(v), ns)
                  * pl.DistanceAttenuation (d) ;
        }
    }
    return col ;
}

```

The `Surface::ReflectedRay()` member function, which returns a mirror-reflected vector, will be discussed in the next section. Figure 20.14(d) illustrates a raytraced scene which incorporates specular reflection.

Before discussing reflected and transmitted light components, let's combine the above outlined ambient light and diffuse and specular reflection components into a single expression for the light intensity,  $I$ , at a given point for a single light source:

$$I = k_a I_a + k_d I_l (\mathbf{n} \cdot \mathbf{l}) + k_s I_l (\mathbf{r} \cdot \mathbf{v})^{n_s}$$

### 20.7.5 Reflected and Transmitted Light

The recursive raytracing algorithm models illumination contributions due to reflected and transmitted rays by spawning both reflected and transmitted rays from the first object surface intersection point for a given ray from the viewer. This process of spawning rays from an object surface intersection point is recursively repeated for each successive reflected and transmitted ray. The light intensity of the original intersection point is determined by accumulating the light intensities due to each spawned ray. From an implementation point of view, an upper limit has to be specified for the number of spawned rays generated so as to guarantee program termination. The raytracing program controls the number of reflected and transmitted spawned rays by the integer constants REFLECTED\_DEPTH and TRANSMITTED\_DEPTH defined in TOL.H./CPP, which are both set to five.

If we consider first the light component due to a reflected ray, it will be assumed that ideal mirror reflection exists in which the angle of reflection is equal to the angle of incidence. The direction of reflection,  $r$ , is easily obtained in terms of the point light source vector,  $l$ , and the surface normal,  $n$ , by inspection of Fig. 20.16 as:

$$r = (2n \cdot l)n - l$$

and is implemented as:

```
Vector3D Surface::ReflectedRay (const Vector3D& i, const
Vector3D& n)
{
    Vector3D nln = n * 2.0 * i.DotProduct (n) ;
    return nln - i ;
}
```

For a reflected ray the reflected component of the illumination model is simply the product of the transmitted intensity,  $I_r$ , and the reflected coefficient,  $k_r$  ( $0 \leq k_r \leq 1$ ), of the object surface:

$$k_r I_r$$

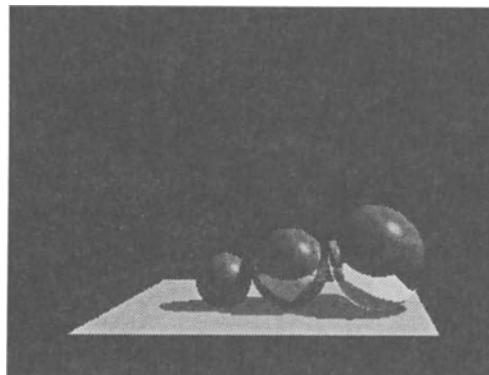
As an illustration of reflected rays consider Fig. 20.17, which shows a raytraced scene consisting of three spheres and a polygon in which  $k_a = 0.7$ ,  $k_d = 0.6$ ,  $k_s = 0.5$  and  $n_s = 25$  for all four objects. The smallest diameter sphere does not reflect rays ( $k_r = 0$ ), the intermediate diameter sphere is 50% reflective ( $k_r = 0.5$ ) and the largest diameter sphere is pure reflective ( $k_r = 1$ ).

The transmitted component of illumination is similar to the reflected component in that the surface light intensity is the product of the transmitted intensity,  $I_t$ , and the transparency coefficient,  $k_t$  ( $0 \leq k_t \leq 1$ ), of the surface:

$$k_t I_t$$

Highly transparent objects are modelled by using a  $k_t$  value approaching 1, whereas opaque objects have a  $k_t$  value approaching 0.

As with reflected light, a recursive process is required to determine the surface intensity due to transmitted light by repeatedly spawning rays from the points of object intersection and accumulating the surface intensity at each intersection point. The difference between the



**Fig. 20.17** A raytraced scene demonstrating different degrees of reflected rays. The smallest diameter sphere does not reflect rays ( $k_r = 0$ ), the intermediate diameter sphere is 50% reflective ( $k_r = 0.5$ ) and the largest diameter sphere is pure reflective ( $k_r = 1$ ). The ambient, diffuse and specular parameters for all four objects are  $k_a = 0.7$ ,  $k_d = 0.6$ ,  $k_s = 0.5$  and  $n_s = 25$ .

reflected and transmitted light components is the direction of the spawned ray. The direction of a transmitted or refracted ray is modelled using Snell's law:

$$\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_i}{\eta_t} = \eta_{it}$$

which relates the angles of incidence,  $\theta_i$ , and refraction,  $\theta_t$ , to the refractive indexes of the incident,  $\eta_i$ , and refractive,  $\eta_t$ , media. Although the incident medium is usually taken to be either air or vacuum ( $\eta_i=1$ ), the raytracing program does allow an alternative value to be defined via the `World::refractive_index` data member. The data member `Surface::ri` allows a refractive index to be set for each object.

With reference to Fig. 20.16 we find that the transmitted vector,  $t$ , is given by:

$$\mathbf{t} = (\eta_{it} \cos\theta_i - \cos\theta_t) \mathbf{n} - \eta_{it} \mathbf{l}$$

Using the identity  $\cos\theta_t = (1 - \sin^2\theta_t)^{1/2}$  and Snell's law we can express  $\mathbf{t}$  purely in terms of  $\mathbf{n}$ ,  $\mathbf{l}$  and  $\eta_{it}$ :

$$\mathbf{t} = (\eta_{it}(\mathbf{n} \cdot \mathbf{l}) - [1 + \eta_{it}^2((\mathbf{n} \cdot \mathbf{l})^2 - 1)]^{1/2}) \mathbf{n} - \eta_{it} \mathbf{l}$$

If the radical term is negative this physically equates to internal reflection, and in this case the raytracing program assumes zero refracted light.

`Surface::TransmittedRay()` implements the above-outlined procedure for determining vector  $\mathbf{t}$ :

```
Vector3D Surface::TransmittedRay (const Vector3D& i,
                                    const Vector3D& n,
                                    float incident_ri)
{
    // ratio of incident to refracted indexes of refraction
    double nit (1.0) ;
```

```

if (ri > TOLERANCE)
    nit = incident_ri / ri ;

double ndi = n.DotProduct (i) ;

// account for complex radical
double sq_root =
    Complex(1.0+nit*nit*(ndi*ndi-1.0),0.0).Sqrt().Real() ;

return n*(nit*ndi-sq_root) - i*nit ;
}

```

The possibility of a negative radical in the above expression is dealt with by defining the radical as a complex number.

### 20.7.6 Distance Attenuation

When light travels through a non-vacuum space its amplitude is attenuated by a factor of  $1/d^2$  where  $d$  is the distance that the light has travelled. From a computational perspective, however, a distance attenuation factor of  $1/d^2$  generates little variation in light intensity when  $d$  is large and significant variations in light intensity when  $d$  is small for a single point light source. A solution to this problem is to replace the  $1/d^2$  attenuation factor by a general quadratic function,  $f(d)$ , but limit the attenuation function to the value of one for small values of  $d$ ; see Hearn and Baker (1994):

$$f(d) = \min \left[ 1, \frac{1}{a_0 + a_1 d + a_2 d^2} \right]$$

The coefficients  $a_0$ ,  $a_1$  and  $a_2$  can be chosen so as to produce realistic distance attenuation effects. However, in the present study an alternative approach is adopted by choosing an exponential decay function, which will produce a gradual decay from one to zero as  $d$  increases from zero to infinity:

$$f(d) = e^{-kd}$$

The exponential decay function is chosen in preference to the quadratic function simply because it is easier to set the value of  $k$  rather than the three coefficients  $a_0$ ,  $a_1$  and  $a_2$ . The default value of  $k$  is 0.01.

The definition of `PointLight::DistanceAttenuation()` is:

```

double PointLight::DistanceAttenuation (double d, double k)
{
    return exp (-k*d) ;
}

```

### 20.7.7 Shadows

Incorporating shadowing in a raytracing program for point light sources is a straightforward procedure that can produce visually effective images. Figure 20.18 illustrates a raytraced scene

of a tetrahedron and a polygon and the associated shadow for a single point light source. Observe that a point light source produces hard-edged shadows.

The method used to determine whether an object intersection point is in shadow or not is to cast a *shadow ray* from the intersection point to the light source. If the shadow ray intersects any other object which lies between the intersection point and the light source then the intersection point is in shadow. If an intersection point is found to be in shadow then only the ambient component of light is incorporated into the illumination model.

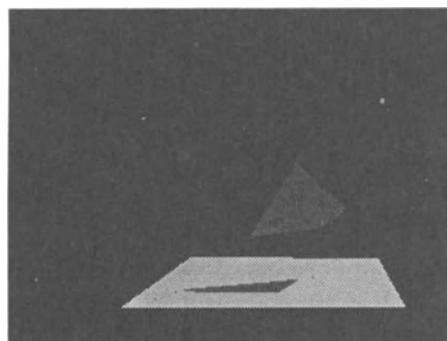
The shadow algorithm is implemented in the `World::AmbDiffSpecColour()` member function, which computes light intensity due to ambient light and diffuse and specular reflections:

```
NormalisedRGBColour
World::AmbDiffSpecColour (const int& obj_index,
                           const Point& int_pt)
{
    NormalisedRGBColour rgb ;

    // ambient light
    rgb = l_obj[obj_index]->Surf().AmbientColour (ambient) ;

    // test if int_pt is in shadow
    Boolean in_shadow (B_FALSE) ;
    for (int i=0; i<l_obj.NumberNodes(); i++)
    {
        if (i != obj_index) // don't test self
        {
            l_obj[i]->NearestIntersection
                (Line(int_pt, light.At()), in_shadow) ;
            if (in_shadow)
                break ;
        }
    }

    // if int_pt not in shadow
    if (!in_shadow)
    {
        Vector3D n = l_obj[obj_index]->Normal (int_pt) ;
```



**Fig. 20.18** A raytraced scene illustrating shadowing.

```

    // diffuse light
    rgb += l_obj[obj_index]->Surf().DiffuseColour
        (int_pt, n, light, viewer) ;

    // specular light
    rgb += l_obj[obj_index]->Surf().SpecularColour
        (int_pt, n, light, viewer) ;
}
return rgb ;
} // AmbDiffSpecColour()

```

The shadow ray is tested for intersection against every object in the world domain (excluding the object in which the intersection point occurs) using the *NearestIntersection()* member function. The algorithm could be speeded up by developing another function for each object, called, say, *Intersection()*, which simply returns logical-true or logical-false if an intersection exists without computing the nearest point of intersection.

### 20.7.8 Simplifications and Extensions to the Surface Illumination and Reflectance Model

The illumination and reflectance models discussed above are simplifications of actual lighting conditions, and one natural extension is to incorporate more physically based models into the raytracing program. Simplified illumination and reflectance models are frequently sought because the raytracing method is inherently a computationally expensive process. For instance, a common simplification to the Phong specular reflection model is the use of the so-called *halfway* vector, which defines a plane that produces maximum specular reflection in the direction of the viewer. Another popular simplification when computing the direction of a refracted ray through a transparent material is simply to assume that the direction of the refracted ray is equivalent to that of the incident ray rather than determining the refracted ray direction exactly using Snell's law.

In the present raytracing program only a single point light source has been modelled. It is not a difficult process to incorporate multiple point light sources and even distributed light sources. A distributed light source also allows us to model soft shadows, which produce a gradual transition from umbra to penumbra to a non-shadow region.

The objects modelled have been assumed to have ideal smooth surfaces of constant colour. Several raytracing programs support *texture* and *bump* mapping. Texture mapping consists of mapping a two-dimensional texture pattern onto the surface of an object, whereas bump mapping attempts to model the stochastic nature of rough surfaces as a dimpled surface.

## 20.8 The World

The world domain comprises all of the objects discussed above, such as a set of world objects, a viewer, a view plane and a point light source; refer to Fig. 20.1. All of these objects are encapsulated in a **class** called **World**, which is the key **class** in the raytracing program and is an extended version of the **World** **class** presented in Chapter 17 in connection with reading from and writing to a disk file a set of objects.

The declaration of **class** **World** in the present chapter is:

```

// world.h
// class World

#ifndef _WORLD_H // prevent multiple includes
#define _WORLD_H

#include <typeinfo.h> // RTTI
#include <cstring.h> // C++ class string

#include "rw_file.h" // classes ReadFile, ...

#include "bool.h" // enum Boolean
#include "ll.h" // template class LinkedList

#include "shape.h" // class Shape
#include "plane.h" // class Plane
#include "poly.h" // class Polygon
#include "tri.h" // class Triangle
#include "quad.h" // class Quadrilateral
#include "circ.h" // class Circle
#include "sphere.h" // class Sphere
#include "tet.h" // class Tetrahedra

#include "dib.h" // class DIBitmap
#include "bbox.h" // class BoundingBox
#include "rgb.h" // classes RGBColour,
                // NormalisedRGBColour
#include "surf.h" // class Surface
#include "light.h" // class PointLight
#include "viewer.h" // class Viewer
#include "view_pl.h" // class ViewPlane
#include "screen.h" // class Screen
#include "rect_win.h" // template class RectWindow

class World
{
private:
    // nested class
    class SortKeyword
    {
        //...
    }; // nested class SortKeyword
    // data members
    // obj and world in files
    ReadFile obj_infile, wor_infile ;
    // obj and world out files
    WriteFile obj_outfile, wor_outfile ;
    LinkedList<Shape*> l_obj ; // list of world objects

    Viewer viewer ; // viewer
    ViewPlane view_plane ; // viewplane in world space
}

```

```
Screen      display_screen ; // display
PointLight   light ;           // point light source
NormalisedRGBColour background ; // background colour
NormalisedRGBColour ambient ;    // ambient colour
double      refractive_index ; // index of refraction

// member functions
LinkedList<SortKeyword> KeywordPositions
    (const string& str, LinkedList<string>& l_kw) ;
Boolean      CheckBracesNoNesting
    (const streampos& fp) ;
Shape*       ReadParticularObject
    (const string& obj_type) ;
LinkedList<Shape*> ReadListOfObjects
    (const string& obj_type, const streampos& fp_arg) ;
Boolean WorldCheckBracesNoNesting (const streampos& fp) ;
void        ReadListOfWorldObjects
    (const string& obj_type, const streampos& fp_arg) ;
NormalisedRGBColour AmbDiffSpecColour
    (const int& obj_index, const Point& int_pt) ;
NormalisedRGBColour ComputeColour
    (Line ray, const Point& int_pt,
     const int& obj_index, int depth) ;
public:
// constructor
World (string o_infile="OBJ_IN.DAT",
        string w_infile="WOR_IN.DAT",
        string o_outfile="OBJ_OUT.DAT",
        string w_outfile="WOR_OUT.DAT",
        const Viewer& v=Viewer(),
        const ViewPlane& v_pl=ViewPlane(),
        const Screen& scr=Screen(),
        const PointLight& l=PointLight(),
        const NormalisedRGBColour& bkgr=
        NormalisedRGBColour(),
        const NormalisedRGBColour& amb=
        NormalisedRGBColour(1.0,1.0,1.0),
        double refrac_index=1.0) ;

// member functions
ReadFile& ObjectsInFile () { return obj_infile ; }
ReadFile& WorldInFile   () { return wor_infile ; }
WriteFile& ObjectsOutFile () { return obj_outfile ; }
WriteFile& WorldOutFile  () { return wor_outfile ; }
Viewer WorldViewer () const { return viewer ; }
ViewPlane VPlane () const { return view_plane ; }
Screen DisplayScreen () const { return display_screen ; }
PointLight Light () const { return light ; }
NormalisedRGBColour Background () const
    { return background ; }
NormalisedRGBColour Ambient () const { return ambient ; }
double RefractiveIndex () const
```

```

    { return refractive_index ; }
LinkedList<string> ReadKeywords
    (const string& kw_fname="OBJ_KW.DAT") ;
void ReadObjects (const string& kw_fname="OBJ_KW.DAT") ;
void ReadWorld   (const string& kw_fname="WOR_KW.DAT") ;
void WriteObjects () ;
void WriteWorld () ;
BoundingBox ObjectsBBox () ;
void GenerateRayTracedImage (const char* filename) ;
}; // class World

#endif // _WORLD_H

```

The set of world objects is encapsulated via the `l_obj` data member which is a linked list of pointers to `class Shape`. The viewer, view plane and point light source are represented by the `viewer`, `view_plane` and `light` data members. `World` also encapsulates `display_screen`, `background`, `ambient` and `refractive_index` data members, which, respectively, model the display screen, background colour, ambient light intensity and refractive index of the world domain medium.

Apart from the addition of appropriate access member functions to `World`, the main additions with respect to generating a raytraced image are `ObjectsBBox()`, `AmbDiffSpecColour()`, `ComputeColour()` and `GenerateRayTracedImage()`. The `World::ObjectsBBox()` member function returns an axis-aligned bounding box object of `class BoundingBox` which encompasses all of the world shape objects. `AmbDiffSpecColour()` computes the light intensity for a given intersection point on a shape object due to ambient light and diffuse and specular reflections, and has previously been discussed in connection with object shadows. The ambient, diffuse and specular components of light have been separated from reflected and transmitted light components because they can be determined directly, whereas a recursive process is required to determine the reflected and transmitted components.

`ComputeColour()` is a recursive member function to determine the intensity of light at a point on a shape object for all light components modelled, and is defined as:

```

NormalisedRGBColour
World::ComputeColour (Line ray, const Point& int_pt,
                     const int& obj_index, int depth)
{
    NormalisedRGBColour rgb ;

    // ambient, diffuse and specular light
    rgb = AmbDiffSpecColour (obj_index, int_pt) ;

    // reflected
    if (depth+1 < REFLECTED_DEPTH &&
        l_obj[obj_index]->Surf().ReflectiveCoeff() > TOLERANCE)
    {
        // if object normal is pointing away from viewer
        // then flip normal
        Vector3D normal = l_obj[obj_index]->Normal(int_pt) ;
        if (normal.DotProduct(ray) < 0.0)
            normal = -normal ;
    }
}

```

```

// ray from intersection point
Vector3D v = l_obj[obj_index]->Surf();
    ReflectedRay (-ray, normal) ;

// ray start point
ray.P0() = int_pt ;

// extend line
double c_alpha (0.0), c_beta (0.0), c_gamma (0.0) ;
v.DirectionCosines (c_alpha, c_beta, c_gamma) ;
double norm_newv = v.Norm () +
    (2.0 * ObjectsBBox().EnclosingRadius()) ;

// new ray terminal point
ray.P1().X() = norm_newv * c_alpha ;
ray.P1().Y() = norm_newv * c_beta ;
ray.P1().Z() = norm_newv * c_gamma ;

// for all world objects (excluding current)-find
// nearest intersection
Boolean intersection (B_FALSE) ;
Point nearest_int_pt (MAX, MAX, MAX) ;
int nearest_obj_index (0) ;

for (int i=0; i<l_obj.NumberNodes(); i++)
{
    if (i != obj_index) // don't test self
    {
        Boolean obj_int (B_FALSE) ;
        Point int_pt = l_obj[i]->NearestIntersection
            (ray, obj_int) ;
        // if an intersection and nearer than
        // previous intersection
        if (obj_int &&
            (int_pt.DistanceBetweenTwoPoints(ray.P0())
             < nearest_int_pt.

                DistanceBetweenTwoPoints(ray.P0())))
        {
            intersection = obj_int ;
            nearest_int_pt = int_pt ;
            nearest_obj_index = i ;
        }
    }
    // determine the colour at the intersection point
    if (intersection)
    {
        // compute colour (note: previous object kr)
        rgb += ComputeColour (ray, nearest_int_pt,

```

```

        nearest_obj_index, depth+1) *
    l_obj[obj_index]->Surf().ReflectiveCoeff() ;
}
else // set colour to the background colour
{
    rgb += background ;
}
}

// transmitted
if (depth+1 < TRANSMITTED_DEPTH &&
    l_obj[obj_index]->Surf().TransmittedCoeff() > TOLERANCE)
{
//...
}
return rgb ;
} // ComputeColour()

```

The member function begins by determining the light intensity due to ambient light and diffuse and specular reflections with a call to *AmbDiffSpecColour()*. The remainder of the function is then divided into two parts which recursively compute additional light intensity contributions due, firstly, to reflected rays and, secondly, to transmitted rays. Considering reflected rays, we observe that *ComputeColour()* proceeds to spawn a reflected ray if the depth parameter is less than REFLECTED\_DEPTH (5) and  $k_r$  for the object is greater than zero. The constant REFLECTED\_DEPTH specifies an upper limit on the number of recursive calls to *ComputeColour()* for reflected rays. If a spawned ray is to be generated its direction is determined from *Surface::ReflectedRay()*. The start-point of the spawned ray is the object intersection point, and the ray terminal point is determined by converting the *Vector3D* ray object returned from *ReflectedRay()* to a *Line* object and extending the ray to ensure that the ray terminal point lies outside the world shape objects' bounding box. This conversion is performed by adding the diameter of the world shape objects' bounding sphere to the length of the ray. The nearest object intersection point is then determined, if one exists, and *ComputeColour()* is again called with the new spawned ray, object, intersection point and depth value. When the light intensity has been accumulated for all reflected rays a similar procedure is followed for transmitted rays.

*World::GenerateRayTracedImage()* generates a Windows format device-independent bitmap 24-bit RGB raytraced bitmap image and writes the image to the specified path and filename, *filename*:

```

void World::GenerateRayTracedImage (const char* filename)
{
// determine the world objects bounding box,
// its centre and enclosing radius
BoundingBox objs_bbox = ObjectsBBox () ;
Point      objs_bbox_centre = objs_bbox.Centre () ;
double     objs_bbox_radius = objs_bbox.EnclosingRadius () ;

// pixel colour and bitmap object
NormalisedRGBColour pixel_rgb ;
DIBitmap dib (display_screen.Window().Width(),
              display_screen.Window().Height(), 24) ;

```

```
unsigned char huge* dib_bits = dib.Bits () ;  
  
// for all pixels in display screen (row by row)  
for (int h=0; h<display_screen.Window().Height(); h++)  
{  
    for (int w=0; w<display_screen.Window().Width(); w++)  
    {  
        // find corresponding point in view plane  
        Point view_pl_win_pt =  
            view_plane.Window().Transform  
            (RectWindow<double>  
            (display_screen.Window().Nearest(),  
            display_screen.Window().Furthest()),  
            Point(w,h)) ;  
        Point view_pl_pt = view_plane.Window().Origin() +  
            view_pl_win_pt ;  
  
        // convert point in view plane coor.  
        // to global coor.  
        Point gl_view_pl_pt =  
            view_plane.GlobalCoordinates (view_pl_pt) ;  
  
        // ray from viewer to view plane point  
        Line ray (viewer.At(), gl_view_pl_pt) ;  
  
        // extend line to back of world objects  
        // bounding box  
        Vector3D v = gl_view_pl_pt - viewer.At () ;  
        double c_alpha (0.0), c_beta (0.0),  
            c_gamma (0.0) ;  
        v.DirectionCosines (c_alpha, c_beta, c_gamma) ;  
  
        double norm_newv = v.Norm () +  
            gl_view_pl_pt.DistanceBetweenTwoPoints  
            (objs_bbox_centre) + objs_bbox_radius ;  
  
        // new ray terminal point  
        ray.P1().X() = norm_newv * c_alpha ;  
        ray.P1().Y() = norm_newv * c_beta ;  
        ray.P1().Z() = norm_newv * c_gamma ;  
  
        // for all world objects-find nearest  
        // intersection from viewer  
        Boolean intersection (B_FALSE) ;  
        Point nearest_int_pt (MAX, MAX, MAX) ;  
        int nearest_obj_index (0) ;  
  
        for (int i=0; i<l_obj.NumberNodes(); i++)  
        {  
            Boolean obj_int (B_FALSE) ;  
            Point int_pt = l_obj[i]->NearestIntersection
```

```

        (ray, obj_int) ;
    // if an intersection and nearer to viewer
    // than previous intersection
    if (obj_int &&
        (int_pt.DistanceBetweenTwoPoints(ray.P0()) )
        < nearest_int_pt.
        DistanceBetweenTwoPoints(ray.P0())))
    {
        intersection = obj_int ;
        nearest_int_pt = int_pt ;
        nearest_obj_index = i ;
    }
}
// determine the colour at the intersection point
if (intersection)
{
    // compute colour
    pixel_rgb = ComputeColour (ray,
                               nearest_int_pt,
                               nearest_obj_index,
                               0) ;
}
else // set colour to the background colour
{
    pixel_rgb = background ;
}

// de-normalise RGB colour
RGBColour de_rgb = RGBColour (pixel_rgb) ;
// set memory with rgb colours
dib_bits[3*((long)h*display_screen.
             Window().Width()+w)] = de_rgb.Red () ;
dib_bits[3*((long)h*display_screen.
             Window().Width()+w)+1] = de_rgb.Green () ;
dib_bits[3*((long)h*display_screen.
             Window().Width()+w)+2] = de_rgb.Blue () ;
}
}
// write dib object to disk
dib.Write (filename) ;
} // GenerateRayTracedImage()

```

*GenerateRayTracedImage()* first determines the dimensions of the bounding box into which the rays from the viewer are cast. A 24-bit RGB device-independent bitmap object of **class DIBitmap** is then defined to encapsulate a raytraced image of width and height set to the display screen rectangular window. The raytraced image is scanned from bottom to top and from left to right for each pixel in the display screen window. For each display screen window pixel the corresponding view plane window point is determined with the aid of the *RectWindow: :Transform()* member function. A ray is then traced from the viewer's position through the view plane point and appropriately extended beyond the world shape objects' bounding box so as to ensure that the ray terminal point does not lie within a shape

object. For each ray, the nearest object intersection point from the viewer is determined, assuming one exists, and the colour calculated. If an object intersection point does not exist then the background colour is chosen. The RGB intensities of the display screen window pixel corresponding to the object-ray intersection point are then set. Finally, the `DIBitmap` object is written to disk.

### 20.8.1 Object and World Data Files

The `World` **class** presented in Chapter 17 encapsulated data members called `obj_infile` and `obj_outfile` of classes `ReadFile` and `WriteFile`, respectively, which similarly encapsulate the path and filenames of the shape objects' *in* and *out* data files. The **public** member functions `ReadObjects()` and `WriteObjects()` and **private** support member functions such as `KeywordPositions()` read from and write to a set of arbitrary objects to a disk file. Similarly, `wor_infile` and `wor_outfile` data members and `ReadWorld()` and `WriteWorld()` member functions are defined in the present chapter to enable the manipulation of world objects. The valid types of shape objects and world objects are specified in the `OBJ_KW.DAT` and `WOR_KW.DAT` keyword data files. `OBJ_KW.DAT` is typically:

```
Triangle
QUADRILATERAL
CIRCLE
SPHERE
TETRAHEDRA
POLYGON
```

and `WOR_KW.DAT` is:

```
VIEWER
VIEWPLANE
SCREEN
POINTLIGHT
BACKGROUNDCOLOUR
REFRACTIVEINDEX
```

A typical shape object *in* data file, `OBJ_IN.DAT`, read by `World::ReadObjects()` is of the form:

```
Sphere
{
  [(2,1,4),1,0,((1,0,0),0.7,0.6,0.5,0,0,25,1)]
}
Sphere
{
  [(4.5,1.5,4),1.5,1,((1,0,0),0.7,0.6,0.5,0.5,0,25,1)]
}
Sphere
{
  [(8,2,4),2,2,((1,0,0),0.7,0.6,0.5,1,0,25,1)]
}
Polygon
```

```

{
[4, (-2, 0, 0), (10, 0, 0), (10, 0, 10), (-2, 0, 10), 3,
 ((0, 1, 0), 0.7, 0.6, 0.5, 0, 0, 25, 1)]
}

```

and a typical world objects *in* data file, WOR\_IN.DAT, read by World::ReadWorld() is:

```

Viewer // viewer: at, looking_at, up
{
[((7, 5, -20), (0, 0, 1), (0, 1, 0))]
}
ViewPlane
{
[((0, 0, 1), (0, 0, 0), (1, 0, 0), (0, 1, 0), (-3, 21, 17, -1))]
}
Screen
{
[((0, 320, 240, 0), 640, 480)]
}
PointLight
{
[((20, 20, 0), (1.0, 1.0, 1.0))]
}
BackgroundColour
{
[(0.0, 0.0, 0.0)]
}
AmbientColour
{
[(1.0, 1.0, 1.0)]
}
RefractiveIndex
{
[1]
}

```

The above OBJ\_IN.DAT, and WOR\_IN.DAT data files were used to generate the raytraced scene shown in Fig. 20.17. The use of two files rather than one helps the user to separate the shape objects from the world objects, although a single *in* data file can be used if preferred.

The corresponding *out* shape objects data file, OBJ\_OUT.DAT, generated by World::WriteObjects() is:

```

Sphere
number: 0, centre: (2, 1, 4), radius: 1,
surface: ((1, 0, 0), 0.7, 0.6, 0.5, 0, 0, 25, 1)
Sphere
number: 1, centre: (4.5, 1.5, 4), radius: 1.5, surface: ((1,
0, 0), 0.7, 0.6, 0.5, 0.5, 0, 25, 1)
Sphere
number: 2, centre: (8, 2, 4), radius: 2, surface: ((1, 0,
0), 0.7, 0.6, 0.5, 1, 0, 25, 1)

```

```
Polygon
number: 3
vertices: (-2, 0, 0), (10, 0, 0), (10, 0, 10), (-2, 0, 10)
edges: [(-2, 0, 0), (10, 0, 0)], [(10, 0, 0), (10, 0, 10)],
       [(10, 0, 10), (-2, 0, 10)], [(-2, 0, 10), (-2, 0, 0)]
surface: ((0, 1, 0), 0.7, 0.6, 0.5, 0, 0, 25, 1)
```

and the *out* world objects data file, WOR\_OUT.DAT, generated by `World::WriteWorld()` is:

```
Viewer
((7, 5, -20), [0.00, 0.00, 1.00], [0.00, 1.00, 0.00])
ViewPlane
normal: [0.00, 0.00, 1.00], origin: (0.00, 0.00, 0.00),
Ip: [1.00, 0.00, 0.00], Jq: [0.00, 1.00, 0.00],
window: [-3.00, 21.00, 17.00, -1.00]
Screen
([0, 160, 120, 0], 640, 480)
PointLight
((20.00, 20.00, 0.00), (1.00, 1.00, 1.00))
BackgroundColour
(0.00, 0.00, 0.00)
AmbientColour
(1.00, 1.00, 1.00)
RefractiveIndex
1.00
```

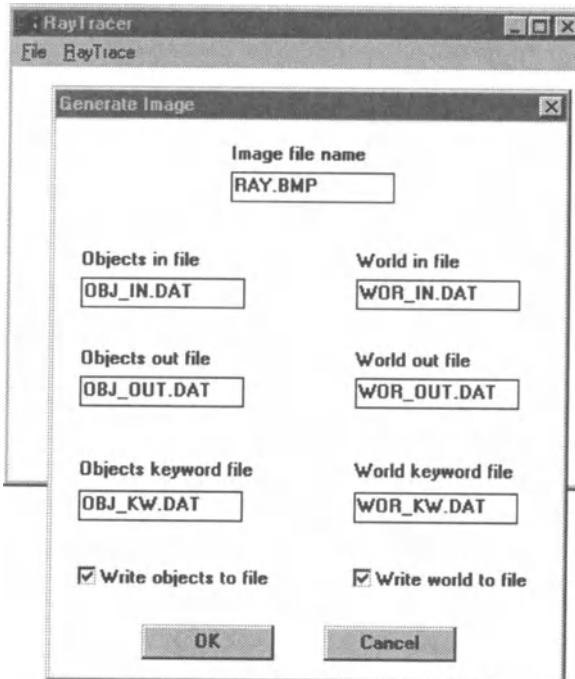
## 20.9 Windows Program

This section briefly discusses a minimal C++ Windows program called RayTracer which illustrates the raytracing method outlined in the previous sections. The main application window and *Generate* dialog box of RayTracer are shown in Fig. 20.19.

RayTracer consists of a single document interface and two pull-down menus, File and RayTrace. File | Exit exits the RayTracer program. RayTrace | Generate displays the *Generate* dialog box and upon a user clicking the OK button proceeds to generate a raytraced image and store the image as a Windows format device-independent bitmap, but does not display the image. RayTrace | Generate.and.Display is the same as RayTrace | Generate but also displays the raytraced image in the client area of RayTracer's main window. RayTrace | Display displays a previously generated and stored raytraced image and calls the *File Open* dialog box from the Windows common dialog box library to extract the path and filename of the bitmap image.

The *Generate* dialog box consists of several edit controls to allow a user to enter a path and filename for the raytraced image and path and filenames for the *in*, *out* and *keyword* shape and world object data files. Two checkboxes allow a user to specify whether or not the shape and world objects are to be written to the files entered in the *out* data file edit controls. After the OK button is clicked an hourglass cursor will be displayed while the program is generating the raytraced image.

The Windows main program function, `WinMain()`, window procedure call-back function, `WndProc()`, and dialog procedure call-back function, `DlgProc()`, are listed in RAYTRACE.CPP:



**Fig. 20.19** The RayTracer application's main window and *Generate* dialog box.

```
WNDCLASS wndclass ;
if (!hPrevInstance)
{
    wndclass.style      = CS_HREDRAW|CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance   = hInstance ;
    wndclass.hIcon       = LoadIcon (hInstance, szAppName) ;
    wndclass.hCursor     = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground =
        (HBRUSH)GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName = (char*)MAIN_WINDOW_MENU ;
    wndclass.lpszClassName = szAppName ;

    RegisterClass (&wndclass) ;
}
HWND hwnd = CreateWindow (szAppName, szAppName,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL) ;

ShowWindow (hwnd, nCmdShow) ;
UpdateWindow (hwnd) ;

MSG msg ;
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
} // WinMain()

// window procedure
long FAR PASCAL _export WndProc (HWND hwnd, UINT message,
                                     UINT wParam, LONG lParam)
{
    static char szFileName  [_MAX_PATH],
                szTitleName [_MAX_FNAME+_MAX_EXT] ;
    static char szFilter[2][21] = { "Bitmap files (*.bmp)",
                                    "*.bmp" } ;
    static OPENFILENAME ofn ;

    static DIBitmap* dib (NULL) ;
    static short      cxClient, cyClient ;
    HDC              hdc ;
    HMENU            hMenu ;
    HINSTANCE        hInstance ;
    DLGPROC          lpDlgProc ;
```

```

PAINTSTRUCT          ps ;

```

```

switch (message)
{
case WM_SIZE:
cxClient = LOWORD (lParam) ;
cyClient = HIWORD (lParam) ;
return 0 ;

case WM_COMMAND:
hMenu = GetMenu (hwnd) ;

switch (wParam)
{
case CM_TRACEGEN:
// generate ray traced image
hInstance = (HINSTANCE)GetWindowWord
(hwnd, GWW_HINSTANCE) ;
lpDlgProc = (DLGPROC)MakeProcInstance
((FARPROC)DlgProc, hInstance) ;
DialogBox (hInstance, "Generate",
hwnd, lpDlgProc) ;
break ;
case CM_TRACEGENDISP:
// generate and display ray traced image
hInstance = (HINSTANCE)GetWindowWord
(hwnd, GWW_HINSTANCE) ;
lpDlgProc = (DLGPROC)MakeProcInstance
((FARPROC)DlgProc, hInstance) ;
if ((DialogBox (hInstance, "Generate",
hwnd, lpDlgProc)) == -1)
break ;

if (dib)
    delete dib ;
dib = new DIBitmap (szBMPFileName) ;
if (dib == NULL)
    MessageBox (hwnd, szAppName,
                "can't create DIB object",
                MB_ICONEXCLAMATION|MB_OK) ;
InvalidateRect (hwnd, NULL, TRUE) ;
break ;
case CM_TRACEDISP:
// display ray traced image
ofn.lStructSize      = sizeof (OPENFILENAME) ;
ofn.hwndOwner        = hwnd ;
ofn.lpstrFilter       = szFilter[0] ;
ofn.lpstrFile         = szFileName ;
ofn.nMaxFile          = _MAX_PATH ;
ofn.lpstrFileTitle    = szTitleName ;
ofn.nMaxFileTitle     = _MAX_FNAME + _MAX_EXT ;

```

```
ofn.lpstrDefExt      = "bmp" ;

if (GetOpenFileName(&ofn))
{
if (dib)
    delete dib ;
dib = new DIBitmap (szFileName) ;
if (dib == NULL)
    MessageBox (hwnd, szAppName,
                "can't create DIB object",
                MB_ICONEXCLAMATION|MB_OK) ;
InvalidateRect (hwnd, NULL, TRUE) ;
}

return 0 ;
case CM_FILEEXIT:
SendMessage (hwnd, WM_CLOSE, 0, 0L) ;
return 0 ;
}
break ;
case WM_PAINT:
hdc = BeginPaint (hwnd, &ps) ;
// display ray traced image
if (dib)
    dib->DisplayBitmap (hdc) ;
EndPaint (hwnd, &ps) ;
return 0 ;
case WM_DESTROY:
if (dib)
    delete dib ;

PostQuitMessage (0) ;
return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
} // WndProc()

// dialog procedure
BOOL CALLBACK _export DlgProc (HWND hDlg, UINT msg,
                               WPARAM wParam, LPARAM lParam)
{
// World object
World world ;

char obj_infile[80], obj_outfile[80],
     wor_infile[80], wor_outfile[80] = "" ;
static char obj_kwfile[80] = "OBJ_KW.DAT" ;
static char wor_kwfile[80] = "WOR_KW.DAT" ;

switch (msg)
{
case WM_INITDIALOG:
```

```

// initialise dialog box
// image file
SendDlgItemMessage (hDlg, IDC_BMPFILE,
                     EM_LIMITTEXT, 80, 0L) ;
SetDlgItemText (hDlg, IDC_BMPFILE, szBMPfileName) ;
// objects files
SendDlgItemMessage (hDlg, IDC_OBJECTSINFILE,
                     EM_LIMITTEXT, 80, 0L) ;
SetDlgItemText (hDlg, IDC_OBJECTSINFILE,
    world.ObjectsInFile().PathAndFileName().c_str()) ;
//...
// world files
SendDlgItemMessage (hDlg, IDC_WORLDINFILE,
                     EM_LIMITTEXT, 80, 0L) ;
SetDlgItemText (hDlg, IDC_WORLDINFILE,
    world.WorldInFile().PathAndFileName().c_str()) ;
//...
// check boxes
CheckDlgButton (hDlg, IDC_OBJECTSCHECK, MF_CHECKED) ;
CheckDlgButton (hDlg, IDC_WORLDCHECK, MF_CHECKED) ;
break ;

case WM_COMMAND:
switch (wParam)
{
    case IDOK:
        // get text fields
        GetDlgItemText (hDlg, IDC_BMPFILE,
                        szBMPfileName, 80) ;
        GetDlgItemText (hDlg, IDC_OBJECTSINFILE,
                        obj_infile, 80) ;
        //...
        // set world data members
        world.ObjectsInFile().
            SetPathAndFileName (obj_infile) ;
        world.ObjectsOutFile().
            SetPathAndFileName (obj_outfile) ;
        world.WorldInFile().
            SetPathAndFileName (wor_infile) ;
        world.WorldOutFile().
            SetPathAndFileName (wor_outfile) ;

        // generate image
        SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
        world.ReadObjects (obj_kwfile) ; // read objects
        world.ReadWorld (wor_kwfile) ; // read world
        world.GenerateRayTracedImage (szBMPfileName) ;
        SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

        // write if requested
        if (IsDlgButtonChecked(hDlg, IDC_OBJECTSCHECK))

```

```

        world.WriteObjects () ;
    if (IsDlgButtonChecked(hDlg, IDC_WORLDCHECK))
        world.WriteWorld () ;
    EndDialog (hDlg, NULL) ;
    break ;

case IDCANCEL:
    EndDialog (hDlg, -1) ;
    break ;
}
break ;

default:
    return FALSE ;
}
return TRUE ;
} // DlgProc()

```

The *Generate* dialog box procedure call-back function *DlgProc()* defines a *World* object called *world* and then, assuming the OK button is clicked, proceeds to generate a raytraced image with path and filename *szBMPFileName* by first reading the shape and world data files using *World::ReadObjects()* and *World::ReadWorld()* and then calling the *World::GenerateRayTracedImage()* member function:

```

World world ;
//...
world.ReadObjects (obj_kwfile) ; // read objects
world.ReadWorld (wor_kwfile) ; // read world
world.GenerateRayTracedImage (szBMPFileName) ;
//...

```

The main menu and *Generate* dialog box of RayTracer are implemented in the resource script file RAYTRACE.RC with identifiers listed in RAYTRACE.RH:

```

// raytrace.rc
// resource script file

#include "raytrace.rh"

RayTracer ICON "raytrace.ico"

MAIN_WINDOW_MENU MENU
{
    POPUP "&File"
    {
        MENUITEM "&Exit", CM_FILEEXIT
    }
    POPUP "&RayTrace"
    {
        MENUITEM "&Generate...", CM_TRACEGEN
        MENUITEM "Generate and Display...", CM_TRACEGENDISP
    }
}

```

```
MENUITEM "&Display...", CM_TRACEDISP
}

Generate DIALOG 35, 48, 194, 226
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Generate Image"
FONT 8, "MS Sans Serif"
{
    DEFPUSHBUTTON "OK", IDOK, 35, 204, 50, 14
    PUSHBUTTON "Cancel", IDCANCEL, 104, 204, 50, 14
    EDITTEXT IDC_BMPFILE, 67, 21, 62, 12
    CHECKBOX "Write world to file", IDC_WORLDCHECK, 115, 180,
              75, 10,
    BS_AUTOCHECKBOX | WS_TABSTOP
    CONTROL "Write objects to file", IDC_OBJECTSCHECK, "BUTTON",
            BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
            10, 180, 78, 10
    LTEXT "Image file name", -1, 67, 9, 60, 8
    EDITTEXT IDC_WORLDINFILE, 115, 64, 62, 12
    EDITTEXT IDC_WORLDOUTFILE, 115, 104, 62, 12
    EDITTEXT IDC_WORLDKWFILE, 115, 150, 62, 12
    EDITTEXT IDC_OBJECTSINFILE, 10, 64, 62, 12
    EDITTEXT IDC_OBJECTSOUTFILE, 10, 104, 62, 12
    EDITTEXT IDC_OBJECTSKWFILE, 10, 150, 62, 12
    LTEXT "World in file", -1, 115, 52, 60, 8
    LTEXT "World out file", -1, 115, 92, 60, 8
    LTEXT "World keyword file", -1, 115, 137, 71, 8
    LTEXT "Objects in file", -1, 10, 52, 60, 8
    LTEXT "Objects out file", -1, 10, 92, 60, 8
    LTEXT "Objects keyword file", -1, 10, 137, 71, 8
}

// raytrace.rh
// resource script header file

#define IDC_BMPFILE      10
#define IDC_OBJECTSINFILE 11
#define IDC_OBJECTSOUTFILE 12
#define IDC_OBJECTSKWFILE 13
#define IDC_WORLDINFILE   14
#define IDC_WORLDOUTFILE  15
#define IDC_WORLDKWFILE   16
#define IDC_OBJECTSCHECK  17
#define IDC_WORLDCHECK     18

#define MAIN_WINDOW_MENU 100
#define CM_TRACEDISP    101
#define CM_TRACEGENDISP 102
#define CM_TRACEGEN      103
```

```
#define CM_FILEEXIT    200
```

The accompanying Windows module definition file for RayTracer is RAYTRACE.DEF:

```
; RAYTRACE.DEF
; module definition file

NAME          RAYTRACER

DESCRIPTION   'Ray tracer for Windows by G. M. Seed, 1996'
EXETYPE       WINDOWS
STUB          'C:\BC5\BIN\WINSTUB.EXE'
CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE MULTIPLE
HEAPSIZE      1024
STACKSIZE     8192
```

## 20.10 Summary

The present chapter should have demonstrated just how good an object-oriented programming language such as C++ is in supporting a programmer involved in a large programming project – an object-oriented raytracing program in the present chapter. The raytracing program is built from approximately 50 files, which can be categorised into geometric and non-geometric tool classes, such as Point, Line, Vector3D, DIBitmap and LinkedList; geometric shape classes, such as Sphere and Polygon; world domain classes, such as Viewer, PointLight and World; and the Windows program files. The majority of the tool classes were developed in previous chapters to illustrate key language features of C++, such as the **class**, operator overloading, **friends**, **templates**, inheritance and object-oriented stream manipulation.

One of the key objectives of all software development is to develop efficient reusable code. Every **class** presented in the present chapter could equally be applied to other geometric modelling projects. The only **class** member function which is hard-wired to the raytracing method is the `World::GenerateRayTracedImage()` function. The raytracing program is reasonably well structured and efficient, but, most importantly, ready to be extended. If you are not happy with the underlying design of the classes or program then do modify them. When it comes to **class** and program design there is no exact solution – just a working solution. One obvious extension to the classes is to make more use of the **template** and exception handling features of C++, which have been purposely neglected in the present chapter so as to help emphasise **class** design and the raytracing method. Alternatively, consider adding other geometric shape classes, designing more efficient ray-object intersection algorithms, incorporating acceleration techniques or extending the Windows program user interface.

## Exercises

- 20.1 Develop a Ray **class** which encapsulates a start-point and direction vector of a line, as illustrated in Fig. 20.4(a). What are the advantages and disadvantages of using Ray rather than Line?

- 20.2 Modify the ray–sphere intersection algorithm of member function `Sphere::NearestIntersection()` using the Ray **class** developed in Exercise 20.1.
- 20.3 Specifically develop a `NearestIntersection()` member function for **class** Triangle. You may want to refer to the article by Synder and Barr (1987).
- 20.4 Use the method of projecting the vertices of a polygon onto a two-dimensional coordinate plane to implement the ray–polygon intersection algorithm in `Polygon::NearestIntersection()`.
- 20.5 Derive a ConvexPolygon **class** from `Polygon` to specifically model convex polygons.
- 20.6 Develop a Hexahedra **class** which models a hexahedron object.
- 20.7 Consider the development of classes to model algebraic and quadric surfaces.

# Conclusion

*computers – they'll never catch on*  
A. Broome, 1995

Although we are concluding our tour of the C++ programming language, I feel as though I've just begun describing C++ because of its wealth of features. Do remember that this text has been an introductory text to C++, with applications specifically oriented to computer graphics. C++ is a big programming language and has numerous applications other than graphical.

If you enjoy the C++ programming language then do continue to develop your understanding of the language and programming skills. A first step in moving from an introductory text is to read slightly more technical books on C++ and books on object-oriented programming, such as Coplien (1992). With reasonable object-oriented programming skills the next step is to begin developing better object-oriented design and analysis skills. A good first text for the C++ programmer which examines the design and analysis issues of object-oriented programming is the book *Object-Oriented Analysis and Design with Applications* (Booch, 1994). As your confidence in the object-oriented approach increases be prepared to learn other object-oriented programming languages. It is often very informative to see how programmers tackle a given problem slightly differently with a different programming language and environment. If you do intend learning other object-oriented programming languages then can I recommend Smalltalk and Java because of their similarity to C++. *Object-Oriented Programming* (Coad and Nicola, 1993) presents a fun-to-read introduction and comparison of object-oriented programming in Smalltalk and C++. Budd (1987) and Anuff (1996) provide a very good introduction to programming in Smalltalk and Java respectively. The Java language is derived from C++, but is simpler due to the removal of several C++ features, such as pointers, multiple inheritance and operator overloading. Java adds a slightly different syntax to C++ and the capabilities of automatic memory management, multithreading and the generation of architecture-neutral object files, which makes Java ideal for a range of environments. The Eiffel object-oriented programming language is derived from Simula and Ada and supports the majority of object-oriented features. Eiffel is developed by ISE and supports strong static typing and an excellent run-time graphical environment (Meyer, 1991; Thomas and Weedon, 1995). Oberon-2 is aimed at the educational community and offers a good introduction to object-oriented programming and executes on a number of platforms and operating systems. The Oberon-2 object-oriented language and Oberon operating system can be obtained free; for details of how to obtain Oberon-2 refer to Mössenböck (1993). Finally, conference proceedings, journals and the Internet are an excellent source of current thinking and development – a few of these sources are listed in Chapter 1.

Several programmers move to programming in a graphical user interface environment, such as Windows, after learning C++. Windows offers an ideal and challenging environment in which to apply your newly learned programming language. If you are interested in programming for Windows I can recommend the book *Programming Windows 3.1* (Petzold, 1992),

which although it is written in C rather than C++ is nevertheless an excellent introduction to Windows programming for the C/C++ programmer. Also, *Borland C++ 4.0 Programming for Windows* (Yao, 1994) is a very enjoyable introduction to Windows programming using the Borland ObjectWindows library. Compilers such as Borland C++ and Microsoft Visual C++ for Windows both include comprehensive class libraries (Borland's ObjectWindows library and Microsoft's foundation class library) specifically designed for Windows programming.

All of the programs developed throughout the book have used sequential programming techniques. Parallel programming languages exist which are based on the C++ programming language. For example, the CC++ language adds object-oriented parallel programming capabilities to the C++ language and is developed by the California Institute of Technology. Information concerning the CC++ compiler and run-time environment can be found on the Internet.

At present C++ is in a period of transition as the ANSI/ISO standard for the C++ programming language is being developed. C++ programmers are used to the language continually developing – it's one of the language's key features. However, this period of standardisation of the language can be very disconcerting for new C++ programmers. If you are new to the C++ language then I certainly recommend referring to the latest ANSI/ISO C++ draft standard, Standard (1996), and not waiting until the standard is approved. In addition, I recommend reading *The Draft Standard C++ Library* (Plauger, 1995), which provides a great insight into the developing C++ standard library.

Several classes have been presented throughout the book which will prove useful to C++ programmers, particularly to programmers involved in graphical and geometric modelling. The classes are far from perfect, but should provide a starting block from which to develop your own data structures and geometric **class** libraries.

The Point **class** has played an important role throughout the latter half of the book and is an important **class** because of the fundamental importance of a point in three-dimensional space. A possible extension to consider when examining **class** Point is that inherent in a Point object is that the object exists in a Cartesian coordinate system. Alternatively, a separate coordinate system **class** hierarchy could be declared and an appropriate coordinate system object encapsulated in **class** Point. This would then allow a Point object to be defined in terms of position and respective coordinate system, which would greatly assist in the manipulation of Point objects in different coordinate systems. Similarly, lines and curves are essential to computational geometric modelling, and the Line **class** could be expanded to a Line **class** hierarchy which models both explicit and implicit straight and curved line objects. Shape objects, such as polygons or polyhedra, could then be defined with straight or curved edges defined explicitly in terms of a number of points or an equation. In addition, the shapes **class** hierarchy could be extended and generalised further to accommodate additional geometric shapes, such as general polyhedra and algebraic surfaces. Recent works which discuss computational geometry and computer graphics in C++ are O'Rourke (1994) and Laszlo (1996). However, if you consider developing your own C++ geometric **class** library to be too fundamental an exercise, then an alternative approach would be to consider integrating your program code with an existing geometric and solid modeller kernel such as SVLIS (Bowyer, 1995), or ACIS (Corney, 1996), which are both written in C++.

# C++ Keywords

The following lists the keywords and reserved words of the C++ language:

<b>asm</b>	To declare that a block of code is to be passed to the assembler
<b>auto</b>	A storage class specifier that is used to define objects in a block
<b>bool</b>	Boolean false-true type that can hold either the <b>false</b> or <b>true</b> literals
<b>break</b>	Terminates a <b>switch</b> statement or a loop
<b>case</b>	Used specifically within a <b>switch</b> statement to specify a match for the statement's expression
<b>catch</b>	Specifies actions taken when an exception occurs
<b>char</b>	Fundamental data type that defines character objects
<b>class</b>	To declare a user-defined type that encapsulates data members and operations or member functions
<b>const</b>	To define objects whose value will not alter throughout the lifetime of program execution
<b>continue</b>	Transfers control to the start of a loop
<b>default</b>	Handles expression values in a <b>switch</b> statement that are not handled by <b>case</b>
<b>delete</b>	Memory deallocation operator
<b>do</b>	Indicates the start of a <b>do-while</b> statement in which the sub-statement is executed repeatedly until the value of the expression is logical-false
<b>double</b>	Fundamental data type used to define a floating-point number
<b>else</b>	Used specifically in <b>if-else</b> statement
<b>enum</b>	To declare a user-defined enumeration data type
<b>explicit</b>	To declare an explicit constructor
<b>extern</b>	An identifier specified as <b>extern</b> has external linkage to the block
<b>false</b>	Boolean literal of value zero
<b>float</b>	Fundamental data type used to define a floating-point number
<b>for</b>	Indicates the start of a <b>for</b> statement to achieve repetitive control
<b>friend</b>	A <b>class</b> or operation whose implementation can access the <b>private</b> data members of a <b>class</b>
<b>goto</b>	Transfer control to a specified label
<b>if</b>	Indicate start of an <b>if</b> statement to achieve selective control
<b>inline</b>	A function specifier that indicates to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation
<b>int</b>	Fundamental data type used to define integer objects

<b>long</b>	A data type modifier that defines a 32-bit <b>int</b> or an extended <b>double</b>
<b>mutable</b>	Allows an object member to override <b>constness</b>
<b>namespace</b>	Defines a scope
<b>new</b>	Memory allocation operator
<b>operator</b>	Overloads a C++ operator with a new declaration
<b>private</b>	Declares <b>class</b> members which are not visible outside the <b>class</b>
<b>protected</b>	Declares <b>class</b> members which are <b>private</b> except to derived classes
<b>public</b>	Declares <b>class</b> members which are visible outside the <b>class</b>
<b>register</b>	A storage class specifier that is an <b>auto</b> specifier, but which also indicates to the compiler that an object will be frequently used and should therefore be kept in a register
<b>return</b>	Returns an object to a function's caller
<b>short</b>	A data type modifier that defines a 16-bit <b>int</b> number
<b>signed</b>	A data type modifier that indicates an object's sign is to be stored in the high-order bit
<b>sizeof</b>	Returns the size of an object in bytes
<b>static</b>	The lifetime of an object defined <b>static</b> exists throughout the lifetime of program execution
<b>struct</b>	To declare new types that encapsulate both data and member functions
<b>switch</b>	Switch statement
<b>template</b>	Parametrised or generic type
<b>this</b>	A <b>class</b> pointer which points to an object or instance of the <b>class</b>
<b>throw</b>	Generate an exception
<b>true</b>	Boolean literal of value one
<b>try</b>	Indicates start of a block of exception handlers
<b>typedef</b>	Synonym for another integral or user-defined type
<b>typename</b>	Within a <b>template</b> <b>typename</b> indicates that a qualified name denotes a type
<b>union</b>	Similar to a structure, <b>struct</b> , in that it can hold different types of data, but a union can hold only one of its members at a given time.
<b>unsigned</b>	A data type modifier that indicates the high-order bit is to be used for an object
<b>using</b>	<b>using</b> declaration and <b>using</b> directive
<b>virtual</b>	A function specifier that declares a member function of a <b>class</b> which will be redefined by a derived <b>class</b>
<b>void</b>	Absent of a type or function parameter list
<b>volatile</b>	Define an object which may vary in value in a way that is undetectable to the compiler
<b>wchar_t</b>	Wide character type
<b>while</b>	Start of a <b>while</b> statement and end of a <b>do-while</b> statement

**ASCII Character Set**

It appears customary to include the ASCII character set in an Appendix of a book on computer programming, so here it is<sup>1</sup>:

Decimal	Hexadecimal	Octal	Binary	Key	ASCII character
0	00	0	00000000	CTRL+2	null
1	01	1	00000001	CTRL+A	☺
2	02	2	00000010	CTRL+B	☻
3	03	3	00000011	CTRL+C	♥
4	04	4	00000100	CTRL+D	♦
5	05	5	00000101	CTRL+E	♣
6	06	6	00000110	CTRL+F	♠
7	07	7	00000111	beep	.
8	08	10	00001000	backspace	█
9	09	11	00001001	tab	■
10	0a	12	00001010	newline	■
11	0b	13	00001011	CTRL+K	♂
12	0c	14	00001100	CTRL+L	♀
13	0d	15	00001101	enter	♪
14	0e	16	00001110	CTRL+N	♫
15	0f	17	00001111	CTRL+O	¤
16	10	20	00010000	CTRL+P	.
17	11	21	00010001	CTRL+Q	,
18	12	22	00010010	CTRL+R	↑
19	13	23	00010011	CTRL+S	!!
20	14	24	00010100	CTRL+T	¶
21	15	25	00010101	CTRL+U	§
22	16	26	00010110	CTRL+V	■
23	17	27	00010111	CTRL+W	↑
24	18	30	00011000	CTRL+X	↑
25	19	31	00011001	CTRL+Y	↓
26	1a	32	00011010	CTRL+Z	→
27	1b	33	00011011	esc	←
28	1c	34	00011100	CTRL+\	@
29	1d	35	00011101	CTRL+]	↔
30	1e	36	00011110	CTRL+6	▲
31	1f	37	00011111	CTRL+-	▼
32	20	40	00100000	spacebar	!
33	21	41	00100001	!	!

<sup>1</sup> Note that characters 0–31 (inclusive) and 127 are control characters, 32–126 represent keys on the keyboard and 127–255 are the IBM extended ASCII character set, which can be displayed by pressing the ALT key and typing the decimal code of the character on the numeric keypad. The IBM extended character set is not part of the ASCII character set and as a result may not be fully portable.

Decimal	Hexadecimal	Octal	Binary	Key	ASCII character
34	22	42	00100010	"	"
35	23	43	00100011	#	#
36	24	44	00100100	\$	\$
37	25	45	00100101	%	%
38	26	46	00100110	&	&
39	27	47	00100111	,	,
40	28	50	00101000	(	(
41	29	51	00101001	)	)
42	2a	52	00101010	*	*
43	2b	53	00101011	+	+
44	2c	54	00101100	,	,
45	2d	55	00101101	-	-
46	2e	56	00101110	.	.
47	2f	57	00101111	/	/
48	30	60	00110000	0	0
49	31	61	00110001	1	1
50	32	62	00110010	2	2
51	33	63	00110011	3	3
52	34	64	00110100	4	4
53	35	65	00110101	5	5
54	36	66	00110110	6	6
55	37	67	00110111	7	7
56	38	70	00111000	8	8
57	39	71	00111001	9	9
58	3a	72	00111010	:	:
59	3b	73	00111011	;	;
60	3c	74	00111100	<	<
61	3d	75	00111101	=	=
62	3e	76	00111110	>	>
63	3f	77	00111111	?	?
64	40	100	01000000	@	@
65	41	101	01000001	A	A
66	42	102	01000010	B	B
67	43	103	01000011	C	C
68	44	104	01000100	D	D
69	45	105	01000101	E	E
70	46	106	01000110	F	F
71	47	107	01000111	G	G
72	48	110	01001000	H	H
73	49	111	01001001	I	I
74	4a	112	01001010	J	J
75	4b	113	01001011	K	K
76	4c	114	01001100	L	L
77	4d	115	01001101	M	M
78	4e	116	01001110	N	N
79	4f	117	01001111	O	O
80	50	120	01010000	P	P
81	51	121	01010001	Q	Q
82	52	122	01010010	R	R
83	53	123	01010011	S	S
84	54	124	01010100	T	T
85	55	125	01010101	U	U
86	56	126	01010110	V	V
87	57	127	01010111	W	W
88	58	130	01011000	X	X
89	59	131	01011001	Y	Y
90	5a	132	01011010	Z	Z
91	5b	133	01011011	[	[
92	5c	134	01011100	\	\
93	5d	135	01011101	]	]
94	5e	136	01011110	^	^
95	5f	137	01011111	_	_
96	60	140	01100000	,	,
97	61	141	01100001	a	a

Decimal	Hexadecimal	Octal	Binary	Key	ASCII character
98	62	142	01100010	b	b
99	63	143	01100011	c	c
100	64	144	01100100	d	d
101	65	145	01100101	e	e
102	66	146	01100110	f	f
103	67	147	01100111	g	g
104	68	150	01101000	h	h
105	69	151	01101001	i	i
106	6a	152	01101010	j	j
107	6b	153	01101011	k	k
108	6c	154	01101100	l	l
109	6d	155	01101101	m	m
110	6e	156	01101110	n	n
111	6f	157	01101111	o	o
112	70	160	01110000	p	p
113	71	161	01110001	q	q
114	72	162	01110010	r	r
115	73	163	01110011	s	s
116	74	164	01110100	t	t
117	75	165	01110101	u	u
118	76	166	01110110	v	v
119	77	167	01110111	w	w
120	78	170	01111000	x	x
121	79	171	01111001	y	y
122	7a	172	01111010	z	z
123	7b	173	01111011	{	{
124	7c	174	01111100	:	:
125	7d	175	01111101	}	}
126	7e	176	01111110	~	~
127	7f	177	01111111	CTRL+←	Δ
128	80	200	10000000	ALT+128	ç
129	81	201	10000001	ALT+129	ü
130	82	202	10000010	ALT+130	é
131	83	203	10000011	ALT+131	â
132	84	204	10000100	ALT+132	ä
133	85	205	10000101	ALT+133	à
134	86	206	10000110	ALT+134	å
135	87	207	10000111	ALT+135	ç
136	88	210	10001000	ALT+136	é
137	89	211	10001001	ALT+137	ë
138	8a	212	10001010	ALT+138	è
139	8b	213	10001011	ALT+139	í
140	8c	214	10001100	ALT+140	î
141	8d	215	10001101	ALT+141	ì
142	8e	216	10001110	ALT+142	Ä
143	8f	217	10001111	ALT+143	Å
144	90	220	10010000	ALT+144	É
145	91	221	10010001	ALT+145	æ
146	92	222	10010010	ALT+146	Æ
147	93	223	10010011	ALT+147	ô
148	94	224	10010100	ALT+148	ö
149	95	225	10010101	ALT+149	ò
150	96	226	10010110	ALT+150	ú
151	97	227	10010111	ALT+151	ù
152	98	230	10011000	ALT+152	ÿ
153	99	231	10011001	ALT+153	Ö
154	9a	232	10011010	ALT+154	Ü
155	9b	233	10011011	ALT+155	ç
156	9c	234	10011100	ALT+156	£
157	9d	235	10011101	ALT+157	¥
158	9e	236	10011110	ALT+158	₱
159	9f	237	10011111	ALT+159	f
160	a0	240	10100000	ALT+160	á
161	a1	241	10100001	ALT+161	í

Decimal	Hexadecimal	Octal	Binary	Key	ASCII character
162	a2	242	10100010	ALT+162	ó
163	a3	243	10100011	ALT+163	ú
164	a4	244	10100100	ALT+164	ñ
165	a5	245	10100101	ALT+165	Ñ
166	a6	246	10100110	ALT+166	ä
167	a7	247	10100111	ALT+167	ö
168	a8	250	10101000	ALT+168	ï
169	a9	251	10101001	ALT+169	߱
170	aa	252	10101010	ALT+170	߲
171	ab	253	10101011	ALT+171	߳
172	ac	254	10101100	ALT+172	ߴ
173	ad	255	10101101	ALT+173	ߵ
174	ae	256	10101110	ALT+174	߶
175	af	257	10101111	ALT+175	߷
176	b0	260	10110000	ALT+176	߸
177	b1	261	10110001	ALT+177	߹
178	b2	262	10110010	ALT+178	߻
179	b3	263	10110011	ALT+179	߻
180	b4	264	10110100	ALT+180	߻
181	b5	265	10110101	ALT+181	߻
182	b6	266	10110110	ALT+182	߻
183	b7	267	10110111	ALT+183	߻
184	b8	270	10111000	ALT+184	߻
185	b9	271	10111001	ALT+185	߻
186	ba	272	10111010	ALT+186	߻
187	bb	273	10111011	ALT+187	߻
188	bc	274	10111100	ALT+188	߻
189	bd	275	10111101	ALT+189	߻
190	be	276	10111110	ALT+190	߻
191	bf	277	10111111	ALT+191	߻
192	c0	300	11000000	ALT+192	߻
193	c1	301	11000001	ALT+193	߻
194	c2	302	11000010	ALT+194	߻
195	c3	303	11000011	ALT+195	߻
196	c4	304	11000100	ALT+196	߻
197	c5	305	11000101	ALT+197	߻
198	c6	306	11000110	ALT+198	߻
199	c7	307	11000111	ALT+199	߻
200	c8	310	11001000	ALT+200	߻
201	c9	311	11001001	ALT+201	߻
202	ca	312	11001010	ALT+202	߻
203	cb	313	11001011	ALT+203	߻
204	cc	314	11001100	ALT+204	߻
205	cd	315	11001101	ALT+205	=
206	ce	316	11001110	ALT+206	#
207	cf	317	11001111	ALT+207	±
208	d0	320	11010000	ALT+208	߻
209	d1	321	11010001	ALT+209	߻
210	d2	322	11010010	ALT+210	߻
211	d3	323	11010011	ALT+211	߻
212	d4	324	11010100	ALT+212	߻
213	d5	325	11010101	ALT+213	߻
214	d6	326	11010110	ALT+214	߻
215	d7	327	11010111	ALT+215	#
216	d8	330	11011000	ALT+216	+
217	d9	331	11011001	ALT+217	]
218	da	332	11011010	ALT+218	߻
219	db	333	11011011	ALT+219	߻
220	dc	334	11011100	ALT+220	߻
221	dd	335	11011101	ALT+221	߻
222	de	336	11011110	ALT+222	߻
223	df	337	11011111	ALT+223	߻
224	e0	340	11100000	ALT+224	߻
225	e1	341	11100001	ALT+225	߻

Decimal	Hexadecimal	Octal	Binary	Key	ASCII character
226	e2	342	11100010	ALT+226	Γ
227	e3	343	11100011	ALT+227	π
228	e4	344	11100100	ALT+228	Σ
229	e5	345	11100101	ALT+229	σ
230	e6	346	11100110	ALT+230	μ
231	e7	347	11100111	ALT+231	τ
232	e8	350	11101000	ALT+232	Φ
233	e9	351	11101001	ALT+233	Θ
234	ea	351	11101010	ALT+234	Ω
235	eb	353	11101011	ALT+235	δ
236	ec	354	11101100	ALT+236	∞
237	ed	355	11101101	ALT+237	φ
238	ee	356	11101110	ALT+238	ε
239	ef	357	11101111	ALT+239	∩
240	f0	360	11110000	ALT+240	≡
241	f1	361	11110001	ALT+241	±
242	f2	362	11110010	ALT+242	≥
243	f3	363	11110011	ALT+243	≤
244	f4	364	11110100	ALT+244	{
245	f5	365	11110101	ALT+245	j
246	f6	366	11110110	ALT+246	÷
247	f7	367	11110111	ALT+247	≈
248	f8	370	11111000	ALT+248	°
249	f9	371	11111001	ALT+249	•
250	fa	372	11111010	ALT+250	.
251	fb	373	11111011	ALT+251	√
252	fc	374	11111100	ALT+252	η
253	fd	375	11111101	ALT+253	²
254	fe	376	11111110	ALT+254	■
255	ff	377	11111111	ALT+255	blank

# Operators: Precedence, Associativity and Arity

The following list shows all of the operators in C++. The precedence, associativity and arity of each operator is given. Note that **sizeof**, **new** and **delete** are operators. All of the operators listed can be overloaded except for `::`, `..`, **sizeof**, `.*` and `?::`.

**Key:** L=Left, R=Right, U=Unary, B=Binary, T=Ternary, N/A=Not applicable

Operator	Precedence	Associativity	Arity	Description
<code>::</code>	17	R	U	global scope
<code>::</code>		L	B	class scope
<code>()</code>	16	L	N/A	function call
<code>()</code>		L	N/A	type construction
<code>[]</code>		L	B	array index
<code>-&gt;</code>		L	B	indirect member access
<code>.</code>		L	B	direct member access
<b>sizeof</b>	15	R	U	object size in bytes
<b>new</b>		R	U	memory allocator
<b>delete</b>		R	U	memory deallocator
<code>()</code>		R	B	cast
<code>-</code>		R	U	minus sign
<code>+</code>		R	U	plus sign
<code>*</code>		R	U	pointer dereference
<code>&amp;</code>		R	U	reference
<code>!</code>		R	U	logical NOT
<code>~</code>		R	U	bitwise NOT
<code>++</code>		R	U	increment
<code>--</code>		R	U	decrement
<code>.*</code>	14	L	B	direct member pointer access
<code>-&gt;*</code>		L	B	indirect member pointer access
<code>/</code>	13	L	B	division
<code>*</code>		L	B	multiplication
<code>%</code>		L	B	modulus
<code>+</code>	12	L	B	addition
<code>-</code>		L	B	subtraction
<code>&lt;&lt;</code>	11	L	B	bit-shift left
<code>&gt;&gt;</code>		L	B	bit-shift right
<code>&lt;</code>	10	L	B	less than
<code>&gt;</code>		L	B	greater than
<code>&lt;=</code>		L	B	less than or equal to
<code>&gt;=</code>		L	B	greater than or equal to
<code>==</code>	9	L	B	equality
<code>!=</code>		L	B	inequality

---

Operator	Precedence	Associativity	Arity	Description
&	8	L	B	bitwise AND
^	7	L	B	bitwise XOR
	6	L	B	bitwise OR
&&	5	L	B	logical AND
	4	L	B	logical OR
? :	3	L	T	conditional
=	2	R	B	assignment
+=		R	B	addition and assignment
-=		R	B	subtraction and assignment
/=		R	B	division and assignment
*=		R	B	multiplication and assignment
%=		R	B	modulus and assignment
&=		R	B	bitwise AND and assignment
=		R	B	bitwise OR and assignment
^=		R	B	bitwise XOR and assignment
<<=		R	B	bit-shift left and assignment
>>=		R	B	bit-shift right and assignment
,	1	L	B	separator

## Glossary

<i>abstraction</i>	The essential characteristics of an object which distinguish it from other objects.
<i>array</i>	Group of identical data types or elements.
<i>base class</i>	A <b>class</b> from which another class can inherit. Also known as a superclass.
<i>character</i>	Such as /, *, ?, a, A, b, .... Refer to Appendix B for a complete listing of the ASCII character set.
<b>class</b>	A <b>class</b> encapsulates data and functions which operate on the <b>class</b> 's data, defining a given structure and behaviour. A <b>class</b> is a type and a scope with an exact specification specified in the <b>class</b> declaration.
<i>comment</i>	A user-defined string of information placed within program code that does not influence compilation.
<i>compile</i>	Convert a program source code file in to an object file which contains machine language instructions.
<i>concrete class</i>	A <b>class</b> that possesses the functionality of C++'s built-in data types, such as <b>int</b> and <b>double</b> .
<i>concurrency</i>	The action of different objects operating simultaneously.
<i>constant</i>	Something that does not change.
<i>declaration</i>	Specifies an identifier's name and type.
<i>definition</i>	A declaration that allocates memory.
<i>derived</i>	A <b>class</b> that has inherited from one or more alternative classes. Also known as a subclass.
<i>encapsulation</i>	The formulation of separating the structure and behaviour of an abstraction into cells.
<b>enum</b>	An enumeration data type which provides mnemonic identifiers to a set of integer constants.

<i>exception</i>	A situation in which a program encounters an abnormal situation for which it was not designed. In C++ you can transfer control ( <b>throw</b> an exception) to another part of a program that is designed to deal explicitly with exceptions.
<i>execution</i>	The process of running a program.
<b>friend</b>	A <b>class</b> or function that has access to the <b>private</b> sections of another <b>class</b> .
<i>function</i>	A standalone module that can have input or output or both.
<i>identifier</i>	A name given to either a constant or a variable object.
<i>inheritance</i>	A hierarchy among classes.
<i>instance</i>	An instance does things, and we can do things to an instance by sending messages. Messages are sent via member functions. An instance is also referred to as an object.
<i>interface</i>	A set of functions within a module.
<i>keyword</i>	Such as <b>int</b> , <b>double</b> or <b>new</b> . A word that the compiler already has a definition for. A keyword (or reserved word) cannot be used as the name of a data object. Keywords in C++ are in lowercase. For a list of keywords refer to Appendix A.
<i>link</i>	Linking is the process of combining object files into a single executable program.
<i>make</i>	Combined process of compiling and linking to generate an executable program.
<i>member function</i>	An operation on an object. Member functions are called methods or operations in other object-oriented programming languages.
<i>module</i>	A piece of program code which forms part of a larger software system. A module has a clearly defined interface. A source file is an example of a module.
<i>object</i>	An object does things, and we can do things to an object by sending messages. Messages are sent via member functions. An object is also referred to as an instance.
<i>persistence</i>	An object is referred to as persistent if it outlives its creator or program execution and/or an object survives a move from its original memory address.
<i>polymorphism</i>	A name that is attached to a variable which denotes objects of several different classes that are related through a common base <b>class</b> . Polymorphous – assuming many forms.
<i>pragmatics</i>	Rules for the good and proper use of a language.
<i>preprocessor</i>	The compiler's preprocessor. Preprocessor directives are prefixed with the number (or hash) sign, #.
<i>primitives</i>	Primary subsystems of a complex system such as base classes.
<b>private</b>	If a member of <b>class</b> is <b>private</b> it can only be accessed by member functions and <b>friends</b> of the <b>class</b> in which it is declared.

---

<b>protected</b>	If a member of a <b>class</b> is <b>protected</b> it can only be accessed by member functions and <b>friends</b> of the <b>class</b> in which it is declared or a <b>class</b> that is derived from this <b>class</b> .
<b>public</b>	If a member of a <b>class</b> is <b>public</b> it can be accessed by any function, member or not.
<i>run-time type information</i>	A mechanism which allows information to be obtained about an object at run-time.
<i>scope</i>	The visibility, availability and life of an object within a program. There are generally six different types of scope: global, local, <b>class</b> , function, <b>namespace</b> and file.
<i>semantics</i>	The meaning of syntactically correct constructs of a language.
<i>source file</i>	A file containing declarations, functions and objects. Frequently referred to as a translation unit.
<b>structure</b>	A user-defined declaration that encapsulates data members and member functions or operations.
<i>syntax</i>	The set of rules for the correct use of a language.
<b>template</b>	A generic <b>class</b> . Often referred to as a parametrised type.
<i>translation unit</i>	A module.
<i>type</i>	A <b>class</b> implements a type. A type determines what values and operations can be performed on an object. All names or objects must have a type in C++.
<i>type checking</i>	All objects are created with a specific type, such as <b>int</b> , <b>double</b> and <b>Complex</b> *. Type checking involves checking that the use of an object is consistent with its type. C++ is often referred to as strongly typed. If violations of type are detected at compilation, a language is referred to as strongly statically typed. Alternatively, if violations of type are allowed to pass compilation and type checking is postponed until run-time then a language is referred to as dynamically typed.
<b>union</b>	Similar to a structure, with the restriction that a <b>union</b> can hold only one of its members at any given time.
<i>variable</i>	As the name suggests, something that changes. The name of a storage location that depends on the type and size of the data that it contains.
<i>virtual function</i>	A base <b>class</b> member function that can be redefined by each derived <b>class</b> .

## References

---

- Ada Reference Manual (1983) *Reference Manual for the Ada Programming Language*, February 1983, Washington DC: Department of Defence, Ada Joint Program Office.
- Adams, J., Leestma, S. and Nyhoff, L. (1995) *C++: An Introduction to Computing*, Prentice-Hall, Englewood Cliffs NJ.
- Anton, H. (1991) *Elementary Linear Algebra*, 6th edn, John Wiley & Sons, New York.
- Anuff, E. (1996) *The Java Sourcebook*, John Wiley & Sons, New York.
- Barton, J. J. and Nackman, L. R. (1994) *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, Reading MA.
- Bashein, G. and Detmer, P. R. (1994) Centroid of a polygon, in *Graphics Gems IV* (ed. Heckbert, P. S.), Academic Press, London.
- Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G. and Moon, D. (1988) *Common Lisp Object System Specification*, X3J13 Document 88-002R, September 1988, SIGPLAN Notices 23.
- Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*, 2nd edn, Benjamin/Cummings, Redwood City CA.
- Bowyer, A. (1995) *SVLIS Set-Theoretic Kernel Modeller: Introduction and User Manual*, 2nd edn, Information Geometers, Winchester.
- Bowyer, A. and Woodwark, J. (1983) *A Programmer's Geometry*, Butterworths, London.
- Bowyer, A. and Woodwark, J. (1993) *Introduction to Computing with Geometry*, Information Geometers, Winchester.
- Budd, T. A. (1987) *A Little Smalltalk*, Addison-Wesley, Reading MA.
- Budd, T. A. (1994) *Classic Data Structures in C++*, Addison-Wesley, Reading MA.
- Coad, P. and Nicola, J. (1993) *Object-Oriented Programming*, Yourdon Press, New Jersey.
- Coplien, J. O. (1992) *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading MA.
- Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990) *Introduction to Algorithms*, MIT Press and McGraw-Hill, New York.
- Corney, J. (1996) *ACIS by Example*, John Wiley & Sons, in preparation.
- Dahl, O.-J., Myrhaug, B. and Nygaard, K. (1970) *SIMULA Common Base Language*, Norwegian Computing Centre S-22, Oslo.
- Ellis, M. A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading MA.
- Ford, W. and Topp, W. (1996) *Data Structures in C++*, Prentice-Hall, Englewood Cliffs NJ.
- Gifford, B. (1990) *Wild at Heart*, Grafton, London.
- Glaeser, G. (1994) *Fast Algorithms for 3D-Graphics*, Springer-Verlag, London.

- Goldberg, A. and Robson, D. (1983) *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading MA .
- Golub, G. H. and Van Loan, C. F. (1989) *Matrix Computations*, 2nd edn, Johns Hopkins University Press, Baltimore MD.
- Graham, R. L. (1972) An efficient algorithm for determining the convex hull of a finite planar set, *Info. Proc. Lett.*, 1, 132–3.
- Hearn, D. and Baker, M. P. (1994) *Computer Graphics*, 2nd edn, Prentice-Hall, Englewood Cliffs NJ.
- Horowitz, E., Sahni, S. and Mehta, D. (1995) *Fundamentals of Data Structures in C++*, Computer Science Press, New York.
- Jennings, A. and McKeown, J. J. (1992) *Matrix Computation*, 2nd edn, John Wiley & Sons, New York.
- Kay, A. C. (1993) *The Early History of Smalltalk*, ACM SIGPLAN Second History of Programming Languages Conference, 20–23 April 1993, ACM, pp. 69–95.
- Kernighan B. and Ritchie D. (1978) *The C Programming Language*, 1st edn, Prentice-Hall, Englewood Cliffs NJ.
- Kernighan B. and Ritchie D. (1988) *The C Programming Language*, 2nd edn, Prentice-Hall, Englewood Cliffs NJ.
- Kirkerud, B. (1989) *Object-Oriented Programming with Simula*, Addison-Wesley, Reading MA.
- Lafore, R. (1991) *Object-Oriented Programming in Turbo C++*, Waite Group Press, Mill Valley CA.
- Lafore, R. (1993) *Lafore's Windows Programming Made Easy*, Waite Group Press, Mill Valley CA.
- Laszlo, M. J. (1996) *Computational Geometry and Computer Graphics in C++*, Prentice-Hall, Englewood Cliffs NJ.
- Lippman, S. B. (1991) *C++ Primer*, 2nd edn, Addison-Wesley, Reading MA.
- Luse, M. (1993) *Bitmapped Graphics Programming in C++*, Addison-Wesley, Reading MA.
- Masters, T. (1993) *Practical Neural Network Recipes in C++*, Academic Press, London.
- Meyer, B. (1991) *Eiffel: The Language*, Prentice-Hall, Englewood Cliffs NJ.
- Meyer, B. (1995) *Object Success: A Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process*, Prentice-Hall, Englewood Cliffs NJ.
- Mortenson, M. E. (1989) *Computer Graphics: An Introduction to the Mathematics and Geometry*, Industrial Press, New York.
- Mössenböck, H. (1993) *Object-Oriented Programming in Oberon-2*, Springer-Verlag, London.
- Negroponte, N. (1995) *being digital*, Hodder & Stoughton, London.
- O'Rourke, J. (1994) *Computational Geometry in C*, Cambridge University Press, Cambridge.
- Peitgen, H.-O., Jurgens, H. and Saupe, D. (1992) *Chaos and Fractals, New Fractals of Science*, Springer-Verlag, London.
- Petzold, C. (1992) *Programming Windows 3.1*, 3rd edn, Microsoft Press, Redmond WA.
- Phong, B. T. (1975) Illumination for Computer-Generated Pictures, *Commun. ACM*, 18(6), 311–17.
- Pinson, L. J. and Wiener, R. S. (1988) *An Introduction to Object-Oriented Programming and Smalltalk*, Addison-Wesley, Reading MA.
- Rock, R. A. and Kalley, G. (1986) *Theory and Problems of Computer Graphics*, Schaum's Outline Series, McGraw-Hill, New York.
- Plauger, P. J. (1995) *The Draft Standard C++ Library*, Prentice-Hall, Englewood Cliffs NJ.
- Porter, A. (1993) *C++ Programming for Windows*, Osborne McGraw-Hill, Maidenhead.
- Preparata, F. P. and Shamos, M. I. (1985) *Computational Geometry: An Introduction*, Springer-Verlag, London.
- Pugh, W. (1989) Skip lists: a probabilistic alternative to balanced trees, in *Proc. Workshop Algorithms and Data Structures*, Ottawa Canada, August 1989.

- Rao, V. B. and Rao, H. V. (1993) *C++ Neural Networks and Fuzzy Logic*, MIS Press, New York.
- Reiser, M. and Wirth, N. (1992) *Programming in Oberon – Steps Beyond Pascal and Modula-2*, Addison-Wesley, Reading MA.
- Richards, M. and Whitney-Strevens, C. (1979) *BCPL: The Language and its Compiler*, Cambridge University Press, Cambridge.
- Ritchie, D. M. (1993) *The Development of the C Language*, ACM SIGPLAN Second History of Programming Languages Conference, 20–23 April 1993, ACM, pp. 201–8.
- Saunders, J. (1989) A Survey of Object-Oriented Programming Languages, *Journal of Object-Oriented Programming*, 1 (6).
- Schildt, H. (1994) *C++ from the Ground Up*, Osborne McGraw-Hill, Maidenhead.
- Schildt, H. (1995) *C++: The Complete Reference*, 2nd edn, Osborne McGraw-Hill, Maidenhead.
- Sedgewick, R. (1988) *Algorithms*, 2nd edn, Addison-Wesley, Reading MA.
- Sedgewick, R. (1992) *Algorithms in C++*, Addison-Wesley, Reading MA.
- Snyder, J. M. and Barr, A. H. (1987) Raytracing Complex Models Containing Surface Tessellations, SIGGRAPH 87, pp. 119–28.
- Standard (1996) *Working Paper for Draft Proposed International Standard for Information Systems-Programming Language C++*. A copy of the latest draft standard can be obtained by writing to X3 Secretariat, Information Technology Councils (ITI), 1250 Eye Street, N.W., Suite 200, Washington, DC 20005-3922, USA. Alternatively, the C++ draft standard can be obtained by ftp from `research.att.com` in directory `/dist/c++/std/WP` and further information can be found at the URL `http://www.cygnus.com/misc/wp`.
- Stevens, R. T. (1992) *Fractal Programming and Ray Tracing with C++*, Prentice-Hall, Englewood Cliffs NJ.
- Stevens, R. T. (1994) *Object-Oriented Graphics Programming in C++*, Academic Press, London.
- Stroustrup, B. (1982) *Classes: An Abstract Data Type Facility for the C Language*, ACM SIGPLAN Notices.
- Stroustrup, B. (1986) *The C++ Programming Language*, Addison-Wesley, Reading MA.
- Stroustrup, B. (1991) *The C++ Programming Language*, 2nd edn, Addison-Wesley, Reading MA.
- Stroustrup, B. (1993a) *A History of C++: 1979–1991*, ACM SIGPLAN Second History of Programming Languages Conference, 20–23 April 1993, ACM, pp. 271–7.
- Stroustrup, B. (1993b) Why consider language extensions?, in *Proc. European C++ User Group Technical Conference*, 7 March–9 July 1993, pp. 50–63.
- Stroustrup, B. (1994) *The Design and Evolution of C++*, Addison-Wesley, Reading MA.
- Sykes, C. (1994) *No Ordinary Genius, The Illustrated Richard Feynman*, Weidenfeld & Nicolson, London.
- Thomas, P. and Weedon, R. (1995) *Object-Oriented Programming in Eiffel*, Addison-Wesley, Reading MA.
- USENIX (1987) Sante Fe NM, C++ Workshop, November.
- USENIX (1988) Denver CO, C++ Conference, October.
- USENIX (1990) San Francisco CA, C++ Conference, April.
- USENIX (1991) Washington DC, C++ Conference, April.
- USENIX (1992) Portland OR, C++ Technical Conference, August.
- USENIX (1993) Munich, European C++ User Group Technical Conference, July.
- Watt, A. and Watt, M. (1992) *Advanced Animation and Rendering Techniques: Theory and Practice*, ACM Press, New York.
- Weiss, M. A. (1996) *Algorithms, Data Structures and Problem Solving with C++*, Addison-Wesley, Reading MA.
- Wells, D. and Young, C. (1993) *Raytracing Creations: Generate 3D Photo-Realistic Images on the PC* (with a chapter by Farmer, D.), Waite Group Press, Mill Valley CA.
- Wilt, N. (1994) *Object-Oriented Ray Tracing in C++*, John Wiley & Sons, New York.
- Yao, P. (1994) *Borland C++ 4.0 Programming for Windows*, Borland Press, Scotts Valley CA.

# Index

*The order of the operators coincides with the precedence levels of Appendix C.*

- ::
  - class scope 263
  - global scope 136
- ()
  - function call 37, 126
  - type conversion 270
- [] (array index) 187
  - overloading 332
- > (indirect member access) 422
  - . (direct member access) 219, 422
- ( ) (cast) 76, 770
  - overloading 337
- \* (pointer dereference) 384
  - & (reference) 139
  - ! (logical NOT) 98
  - ~ (one's complement) 120
- ++ (increment) 68
  - overloading 325
- (decrement) 68
  - overloading 325
- . \* (direct member pointer access) 422
  - >\* (indirect member pointer access) 422
- / (division) 67
  - overloading 324
- \* (multiplication) 67
  - overloading 324
- % (modulus) 67
  - overloading 349
- + (addition) 67
  - overloading 320
- (subtraction) 67
  - overloading 320
- << (bit-shift left) 39, 120
  - overloading 343, 372, 785
- >> (bit-shift right) 120
  - overloading 343, 372, 785
- < (less than) 94
  - overloading 329
- > (greater than) 94
  - overloading 329
- <= (less than or equal to) 94
  - overloading 329
- >= (greater than or equal to) 94
  - overloading 329
- == (equal to) 94
  - overloading 329
- != (not equal to) 94
  - overloading 329
- & (bitwise AND) 116
  - ^ (bitwise exclusive OR) 119
  - | (bitwise OR) 118
- && (logical AND) 96
  - || (logical OR) 97
- ? : (conditional) 102
  - = (assignment) 67
  - overloading 329, 633
- += (addition and assignment) 68
  - overloading 324
- = (subtraction and assignment) 68
  - overloading 324
- /= (division and assignment) 68
  - overloading 324
- \*= (multiplication and assignment) 68
  - overloading 324
- %= (modulus and assignment) 68
  - &= (bitwise AND and assignment) 121
  - | = (bitwise OR and assignment) 121
  - ^= (bitwise exclusive OR and assignment) 121
  - <<= (bit-shift left and assignment) 121
  - >>= (bit-shift right and assignment) 121
- , (comma) 48
  - /\* . . . \*/ (C-style comments) 32
  - // (C++-style comments) 32
- abort () 113, 429, 579
  - abstract **class** see **class**, abstract
  - abstraction 14
- access declaration see inheritance, access declaration
  - access modifier
  - const** 59
  - volatile** 61
- access permission see file, access permission
  - access specifier
  - inheritance 615
  - private** 251, 615
  - protected** 622
  - public** 251, 615
- Algol 9

---

**ambient light** 986, 989  
**ANSI**  
  standard library 163  
    C header files 166  
    C++ header files 167  
**append()** 827  
**argc** 852  
**argv** 852  
**arithmetic assignment**  
  overloading *see operator overloading, arithmetic assignment*  
**array** 183  
  data member 304  
  function arguments 200  
    one-dimensional 200  
    two-dimensional 202  
  higher order 194  
  indexing 187  
  of objects 303  
  one-dimensional 189  
    initialisation 189  
  of pointers 405  
  pointers and 398  
  size 185  
  of structures 223  
  two-dimensional 190  
  initialisation 192  
**array subscript operator**  
  overloading *see operator overloading, array subscript operator*  
**ASCII character set** 46  
**asm** 81  
**assert()** 429, 914  
**Assert()** 591  
**assignment** 66  
  operator overloading *see operator overloading, assignment operator*  
  structure objects 221  
**assignment suppression character** 861  
**atof()** 853  
**atoi()** 853  
**atol()** 853  
**auto** 77  
  **class** *see class, data member, auto*  
**automatic memory management** 18  
  
**bad()** 807  
**Bad\_cast** 768  
**Bad\_typeid** 768  
**binary number** 116  
**binary operator**  
  overloading *see operator overloading, binary operators*  
**bit fields**  
  **class** *see class, bit fields*  
  structure *see structure, bit fields*  
**bitmap** 881  
  BITMAPFILEHEADER 881  
  BITMAPINFO 882  
  BITMAPINFOHEADER 881  
**bitwise operators** 115  
  &, AND 116  
  ^, XOR 119  
  |, OR 118  
  
  ~, one's complement 120  
**bool** 56  
**Boolean** 241  
**Borland**  
  IDE 26  
  ObjectWindows 250, 303  
**bounding box** 172, 952  
**BoundingBox** 951  
**break** 100  
  loop 111  
**switch** 100  
  
**C++**  
  applications 20  
  future 4  
  history 3  
**c\_str()** 828  
**calloc()** 419  
  cascading 39  
**case** 99  
  casting 76, 770  
  pointer 416  
**catch** 567  
  all exceptions 573  
  multiple statements 572  
**ceil()** 817  
**cerr** *see stream, predefined (C++)*  
**char** 46  
**character**  
  constant 46  
  string constant 49  
**cin** *see stream, predefined (C++)*  
**Circle** 353, 713, 975  
**circle-line intersection** *see ray-object intersections*  
**class** 253  
  abstract 696  
  base 612  
    constructor 623  
    destructor 623  
  bit fields 269  
  constructor 270  
    calling 272, 301  
    **const** 284  
    default 273, 275  
    default arguments 293  
    derived **class** *see class, derived, constructor*  
    **inline** 289  
    multiple inheritance 682  
    overloaded 274  
    **volatile** 284  
  container 29  
  copy constructor 278  
    function calls 282  
    overloading 280  
      inheritance 630  
  data member 253  
    **auto** 267  
    **const** 267  
    **extern** 267  
    initialisation 276  
    **mutable** 267  
    **register** 267  
    **static** 265  
    **volatile** 269

---

declaration 253,308  
 derived 612  
     constructor 623  
         default 623  
     destructor 623  
         default 623  
 destructor 285,623  
     **const** 287  
     derived **class** *see class*, derived, destructor  
     multiple inheritance 682  
     **volatile** 287  
 local 304  
 member function 256,287  
     automatic **inline** 288  
     calling 260,301  
     **const** 290  
     default arguments 293  
     **inline** 287  
     naming 260  
     object arguments 295,452  
     out-of-line definition 261  
     returning  
         a local object 448  
         object 295  
         by reference 299  
     **static** 265  
     **volatile** 293  
 method 256  
 nested 305  
 polymorphic 689,765  
**private**  
     data member 251  
**protected**  
     data member 621  
**public**  
     data member 251  
     member function 257  
**template** *see template, class*  
**virtual** base 686  
**virtual** member function 689  
**clear()** 807  
**clearerr()** 879  
 clockwise-anticlockwise 173,309  
**clog** *see stream, predefined (C++)*  
**close()** 809  
**clreol()** 848  
**clrscr()** 848  
 code reusability 644  
 coefficient of ambient reflection 989  
 coefficient of diffuse reflection 990  
 coefficient of reflection 994  
 coefficient of specular reflection 992  
 coefficient of transparency 994  
 colour table 881  
 command line arguments 160,851,872  
 comments 32  
     /\* . . . \*/ *see / \* . . . \*/*  
     // *see //*  
     nested 33  
 compile 40  
 complex number 364,948  
**Complex** 366,947  
 composed type 529  
 composite operators 334  
 compound statement 88  
     **conbuf** 778,847  
     **concurrency** 18  
     **conditional compilation statement** 148  
     **conditional operator** 102  
     **console streams** *see stream, console*  
**const** 59  
     data member *see class*, data member, **const**  
     pointer 389  
     reference arguments 140  
     return value 140  
**const\_cast<>()** 773  
 constant 62  
     decimal 62  
     enumeration 63  
     floating-point 62  
     hexadecimal 62  
     integer 62  
     octal 62  
**constream** 781,847  
 constructive solid geometry 980  
 constructor *see class*, constructor  
 containment 655,660  
     **hasa** relationship 660  
**continue** 113  
 conversion  
     characters *see printf()*, conversion characters  
     function 337  
 convex hull 554  
 copy constructor, *see class*, copy constructor  
**cout** *see stream, predefined (C++)*  
  
     data abstraction 8  
     data-hiding 11  
     data member *see class*, data member  
     Date 225  
     dec 788  
     declaration 64  
         **class** *see class*, declaration  
     decrement operator *see operator, decrement*  
**default** 100  
 default arguments  
     member function *see class*, member function,  
         default arguments  
 definition 64  
**delete** 424  
     array 431  
     global 441  
     object 424  
     overloading 438  
         inheritance 637  
**delline()** 848  
 dereferencing 384  
 destructor *see class*, destructor  
 device independent bitmap 880  
     file format 881  
**DIBitmap** 880,883,949  
 diffuse light 990  
 diffuse reflection 986  
 digraph 50  
 direction angles 655  
 direction cosines 655  
 direct member access operator 219  
 disk file *see file, disk file*  
 display screen 981

distance attenuation 996  
**do-while** 109  
**double** 53  
double-dispatch 761  
down-cast 761  
**dynamic\_cast<>()**  
    pointer 763  
    reference 765  
dynamic memory allocation 419

Eiffel 5  
**else** *see if-else*  
EmployedPerson 226  
encapsulation 15  
end of file *see file, end of*  
    endl 39,788  
    ends 788  
**enum** 55  
enumerated type 239  
**eof()** 807  
epsilon error 54  
escape sequence 49  
exception handling 567  
    constructor 591  
    exception classes 590  
    exception specification 576  
    functions 574  
    inheritance *see inheritance, exception handling*  
    **new** operator 606  
        re-throwing an exception 578  
exception specification 751  
**exit()** 113,579  
**explicit** 339  
expression 66  
**extern** 78,161  
    **class** *see class, data member, extern*  
external linkage 161  
extraction operator 785

**fabs()** 147  
**fail()** 807  
**false** 56  
**fclose()** 868  
**feof()** 879  
**ferror()** 879  
**fflush()** 879  
**fgetc()** 869  
**fgetpos()** 877  
**fgets()** 871  
FILE 867  
file  
    access permission 806  
    disk file 798  
    end of 808  
    header 144  
    implementation 144  
    objects and 817  
    opening mode 806  
    pointer 813,876  
    random access  
        (C) 876  
        (C++) 813  
    read

    (C) 869  
    (C++) 74,798,821  
    read and write 821  
    stream classes 780  
    write  
        (C) 872  
        (C++) 74,803,821  
file pointer *see file, pointer*  
**filebuf** 778  
**fill()** 797  
**find()** 827  
fixed 795  
**flags()** 796  
**float** 52  
**floor()** 817  
flush 788  
**flushall()** 880  
**fopen()** 868  
**for** 103  
    nested 110  
format flags *see ios, format flags*  
format string 856  
formatting  
    input 70,788  
    output 70,788  
**fprintf()** 876  
**fputc()** 872  
**fputs()** 873  
**fread()** 874  
**free()** 418  
**freopen()** 880  
**friend** 353  
    **class** 358  
    function 353  
    inheritance 643  
operator overloading *see operator overloading, friend*  
**virtual** functions 693  
**fscanf()** 876  
**fseek()** 877  
**fsetpos()** 877  
**fstream** 781,805  
**fstreambase** 778  
**ftell()** 877  
function 126  
    arguments 37,136  
        default 155  
        pointer 390  
        reference 138  
            **const** 140  
        value 137  
array 200  
    one-dimensional 200  
    string 205  
    two-dimensional 202  
body 37,127  
calling 130  
declaration 132  
definition 131  
**inline** 158  
libraries 162  
member *see class, member function*  
name 37,127  
overloading 150  
pointer to 409

---

**prototype** *see* function, declaration  
**return** 129  
  **const** 140  
  multiple values 141  
  reference 142  
**return type** 37, 128  
  default 129  
  **void** 128  
**reusability** 144  
**scope** 133  
  global 134  
  local 133  
  precedence 135  
**signature** *see* function, declaration  
**template** *see* **template**, function  
  use of 143  
**fundamental data type** 45  
**FuzzyShape** 10, 698  
**fwrite()** 874  
  
**Gaussian elimination** 477  
**gcount()** 812  
**get()** 197, 800  
**getche()** 108  
**getline()** 801  
**gets()** 870  
**GlobalMemory** 444, 513, 734  
**global variable** 267  
**good()** 807  
**goto** 114  
  
**header file** 34, 263  
**hello, object** 31  
**hello, world** 856  
**hex** 788  
**Hexahedron** 240  
**highvideo()** 848  
  
**if** 86  
  nested 92  
  scope 93  
**if-else** 87  
**ifstream** 780, 799  
**illumination** 986  
**imomanip** 793  
**implementation file** 263  
**include** 34  
  " " 35  
  <> 35  
**increment operator** *see* operator, increment  
**Index** 598  
**indirection** 384  
**indirect member access operator** 219, 422  
**inheritance** 15, 612, 675  
  access declaration 617  
  access specifier 615  
    default 616  
  **private** 615  
  **protected** 622  
  **public** 615  
**declaration** 614  
**exception handling** 748  
  
**isa relationship** 675  
**member function**  
  overloading 639  
  overriding 640  
  multiple 680  
  operator overloading 643  
**template** 722  
**initialisation** 66  
**inline** 158  
  constructor *see* **class**, constructor, **inline**  
  function *see* function, **inline**  
  member function *see* **class**, member function,  
    **inline**  
  insertion operator 785  
**insline()** 848  
**instance** 254  
**int** 50  
**ios** 778  
  format flags  
    dec 795  
    fixed 795  
    hex 795  
    internal 795  
    left 795  
    oct 795  
    right 795  
    scientific 795  
    showbase 795  
    showpoint 795  
    showpos 795  
    skipws 795  
    stdio 795  
    unitbuf 795  
    uppercase 795  
**iostream** 780  
**is\_open()** 811  
**isspace()** 784  
**istream** 780  
**istream\_withassign** 780  
**istrstream** 780, 846  
**iterator** 530  
  **class** 535  
  function 530  
**Iterator** 745  
**itoa()** 853  
  
**Java** 5, 28  
  
**keyword** 36, 64  
  
**Line** 232, 295, 312, 658, 939  
**line intersection** 168, 309, 943  
**link** 40  
**LinkedList** 538, 741, 820, 945  
**LinkedListIterator** 547, 745, 945  
**literal** 62  
**local class** *see* **class**, local  
**LocalMemory** 734  
**local plane coordinates** 959  
**logical**  
  false 94  
  true 94

---

logical operators 95  
   !, NOT 98  
   &&, AND 96  
   ||, OR 97  
**long** 59  
 loop 103  
   **do-while** 109  
   **for** 103  
   forever 110  
   nested 110  
   scope 115  
   **while** 107  
**lowvideo()** 848  
**ltoa()** 853

**macro** 904  
**predefined**  
   \_\_cplusplus 912  
   \_\_DATE\_\_ 912  
   \_\_FILE\_\_ 912  
   \_\_LINE\_\_ 912  
   \_\_STDC\_\_ 912  
   \_\_TIME\_\_ 912  
**main()** 36, 159, 851  
**malloc()** 418  
**manipulators** 70  
   console 848  
   dec 73  
   endl 72  
   flush 72  
   hex 73  
   non-parametrised 788  
   oct 73  
   parametrised 788  
   **resetiosflags()** 73  
   **setiosflags()** 73  
   **setprecision()** 73  
   **setw()** 72  
   user-defined  
     non-parametrised 790  
     parametrised 791  
**Matrix** 462, 515, 738  
**Max()** 487, 937  
**member function** *see class, member function*  
**memcpy()** 416  
**memory** 209  
**Memory** 734  
**memset()** 435  
**Mesh** 722  
**message** 257  
**method** 257  
**Min()** 937  
**modifiers** 58  
**modularity** 15  
**multiple inheritance** *see inheritance, multiple*  
**mutable** 267  
   data member *see class, data member, mutable*

**namespace** 918  
   alias 920  
   nameless 924  
**nested class** *see class, nested*  
**new** 424

array 431  
 exception handling *see exception handling, new*  
   operator  
   global 441  
   object 424  
   overloading 438  
     inheritance 637  
     placement syntax 447  
 Newton-Raphson method 410  
 NormalisedRGBColour 949  
**normvideo()** 849  
 NULL 385  
 null termination character 196  
 NumberedPoint 646, 817

**Oberon-2** 5  
**object** 19, 254  
   arrays of 303  
   **const** 292  
   default initialisation 273  
   definition 272  
   temporary 301  
   **volatile** 293  
**object data file** 1006  
**object-oriented programming** 9  
**ObjectWindows** *see Borland, ObjectWindows*  
**oct** 788  
**ofstream** 781, 803  
**omanip** 794  
**open()** 809  
**opening mode** *see file, opening mode*  
**operation** 257  
**operator** 67  
   arithmetic 67  
   arity 69  
   associativity 69  
   chaining 67  
   decrement 68  
   increment 68  
   precedence 69  
**operator** 322  
**operator overloading** 18  
   << 372  
   >> 372  
   arithmetic assignment 324  
     inheritance 633  
   array subscript operator 332  
   assignment operator 329  
   binary operators 319  
   composite operators 324  
   extraction operator *see >>, overloading*  
   **delete** 438  
   **friend** 363  
   inheritance *see inheritance, operator overloading*  
   insertion operator *see <<, overloading*  
   **new** 438  
   non-member functions 331  
   operators that cannot be overloaded 349  
   physical meaning 328  
   relational operators 329  
   unary operators 325  
**ostream** 780  
**ostream\_withassign** 780  
**ostrstream** 780, 846

---

overloading  
     function *see* function, overloading  
     operator *see* operator overloading  
 overriding member functions *see* inheritance, member  
     function, overriding  
*OwlMain()* 161

palette 881  
*PathAndFile* 821  
*pcount()* 846  
*peek()* 803  
 perfect mirror reflection 986  
 perpendicular distance to a plane 959  
 persistence 18  
 Person 215  
*PlanarPolygon* 721  
 plane 702  
 Plane 700, 956  
 plane coordinates 959  
 plane-line intersection *see* ray-object intersections  
 Point 227, 249, 493, 645, 655, 785, 934, 937  
 POINT 244, 250  
 point light source 988  
 point on a line 942  
 point on a plane 959  
 pointer 377  
     addition 393  
     arithmetic 393  
     to an array 405  
     array of 405  
     casting 416  
     **const** 389  
     to data member 422  
     decrement 395  
     to function 409  
     function argument 390  
     increment 395  
     initialisation 385  
     to member function 422  
     to member operators 422  
 NULL 385  
 to objects 419  
 to pointer 404  
 relational operators 397  
 returning 415  
 smart 512  
 to a string constant 409  
 subtraction 394  
**this** 454  
**void** 387  
*PointLight* 988  
 polygon  
     area 709  
     concave 548  
     convex 548  
     point in 552  
 Polygon 548, 703, 962  
 polygon-line intersection *see* ray-object intersections  
*Polyhedra* 722  
 polymorphism 16, 689  
*POV-Ray raytracer* 934  
 pragmatics 20  
*precision()* 797  
 preprocessor 903

directive 34, 903  
     # 904  
     #define 904  
     #elif 910  
     #else 910  
     #endif 910  
     #error 907  
     #if 910  
     #ifndef 910  
     #endif 910  
     #include 908  
     #line 908  
     #pragma 909  
     #undef 909  
     defined 911  
*printf()* 856  
     conversion characters 857

**private**  
     data member *see* **class, private**, data member  
 procedural programming 6  
 programming style 40  
 programming in the large 264  
**protected** 622  
*PtrVector* 727

**public**  
     data member *see* **class, public**, data member  
     member function *see* **class, public**, member  
         function  
*put()* 804  
*putback()* 803  
*putchar()* 870

*qsort()* 388  
 Quadrilateral 711, 966  
 quadrilateral-line intersection *see* ray-object  
     intersections  
 queue 743  
*Queue* 741

*rand()* 112  
 random file access *see* file, random access  
 Range 597  
 Ray 954  
 ray-object intersections 953, 979  
     ray-circle intersection 975  
     ray-plane intersection 955  
     ray-polygon intersection 962  
     ray-quadrilateral intersection 962  
     ray-sphere intersection 971  
     ray-tetrahedra intersection 969  
     ray-triangle intersection 962  
 raytracing 934  
     image 1003  
     program 1008  
*rdbuf()* 799, 848  
*rdstate()* 807  
*read()* 812  
*ReadAndWriteFile* 821  
*ReadFile* 821, 949  
 reading objects 830  
*realloc()* 419  
 RECT 244  
*RectWindow* 982

redirection 849  
 reflectance 986  
 reflected light 986, 994  
 refractive index 995  
**register** 79  
 class *see* **class**, data member, **register**  
**reinterpret\_cast<>()** 772  
 relational operators 94  
 overloading *see* operator overloading, relational operators  
**remove()** 880  
**rename()** 880  
**resetiosflags()** 788  
**return** 129  
 returning by reference  
 member function *see* **class**, member function, returning, by reference  
**rewind()** 879  
**rfind()** 827  
 RGBColour 949  
 RGBQUAD 882  
 run-time type information 763  
 RVector 723

**scanf()** 860  
 conversion characters 861  
 scanset format specifier 864  
 scientific 795  
 scope 133  
 function *see* function, scope  
 loop 115  
 scope resolution operator *see* **:** :  
 nested **class** 308  
**seekg()** 813  
**seekp()** 816  
 semantics 19  
**set\_new\_handler()** 607  
**set\_terminate()** 579  
**set\_unexpected()** 581  
**setattr()** 849  
**setbase()** 788  
**setbk()** 849  
**setclr()** 849  
**setcrsrtype()** 849  
**setf()** 796  
**setfill()** 788  
**setiosflags()** 788  
**setprecision()** 788  
**setw()** 788  
**setxy()** 849  
 shadowing 996  
 Shape 700, 953  
 shift operators 115  
 <<, left 120  
 >>, right 120  
**short** 59  
 showbase 795  
 showpoint 795  
 showpos 795  
**signed** 59  
 Simula 9  
**size\_t** 827  
**sizeof** 64, 186  
 Smalltalk 5

smanip 794  
 smart pointers 512  
 SortKeyword 833  
 specular reflection 986, 992  
 Sphere 972  
 sphere-line intersection *see* ray-object intersections  
**sprintf()** 867  
 square root 145  
**srand()** 112  
**sscanf()** 866  
**stack** 741  
 Stack 741  
 standard  
 input 783  
 output 782  
 statement  
 compound 88  
 null 39  
 program 38  
**static** 80  
 data member *see* **class**, data member, **static**  
 local variables 156  
 member function *see* **class**, member function, **static**  
**static\_cast<>()** 770  
 stdio 795  
 storage class specifier  
**auto** 77  
**extern** 78  
**register** 79  
**static** 80  
**str()** 846  
**strcmp()** 389  
**strcpy()** 415  
 stream 35, 778  
 C approach 855  
 C++ approach 778  
 console 847  
 input  
 (C) 860  
 (C++) 778, 783  
 output  
 (C) 856  
 (C++) 39, 778, 782  
 predefined (C)  
 stdaux 855  
 stderr 855  
 stdin 855  
 stdout 855  
 stdprn 855  
 predefined (C++)  
 cerr 781  
 cin 781  
 clog 781  
 cout 39, 781  
 status flags 807  
 string 846  
 streambuf 778  
 string 195  
 constant 39, 198  
 formatting 846  
 variable 195  
 string 629  
 String 340, 427, 506  
 strstream 780, 846

---

**strstreambase** 778, 846  
**strstreambuf** 778, 846  
**strtod()** 853  
**strtol()** 853  
**strtoul()** 853  
**struct** 215  
**structure** 215  
     accessing 219  
     anonymous *see* structure, nameless  
     array of 223  
     bit fields 235  
     data member 217  
         **public** 231  
     declaration 217  
     default assignment operator 221  
     function argument 224  
     initialiser list 218  
     member function 230  
     name 218  
     nameless 220  
     nested 225  
     object of 218  
     operator overloading 222  
     **private** 231  
     **public** 231  
     size of 219  
     tag 218  
**SuperQuadrics** 722  
**Surface** 954, 987  
**SVLIS** geometric modeller 980  
**switch** 99  
**Switch** 241  
**syntax** 19

**tellg()** 813  
**tellp()** 816  
**template** 488  
     **class** 493  
         multiple type arguments 502  
         overloading 505  
     function 487  
         argument 490  
         multiple type arguments 490  
         overloading 492  
     inheritance *see* inheritance, **template**  
**terminate()** 579  
**Tetrahedra** 667, 715, 969  
**tetrahedra-line intersection** *see* ray-object intersections  
**tetrahedron** 671  
     centroid 671  
     normal 672  
     perimeter 672  
     surface area 672  
     volume 672  
**textmode()** 848  
**texture mapping** 998  
**this** 454  
**throw** 567  
**time()** 112  
**to\_lower()** 803  
**to\_upper()** 803  
**tolower()** 803  
**toupper()** 803  
**TPoint** 250

**transmitted light** 986, 994  
**triangle**  
     centroid 665  
     normal 666  
     perimeter 665  
     surface area 175  
**Triangle** 232, 660, 709, 966  
**triangle-line intersection** *see* ray-object intersections  
**trigraph** 50  
**true** 56  
**try** 567  
**two\_omanip** 848  
**type** 16, 45  
**type\_info** 768  
**typedef** 242  
     Windows 244  
**typeid()** 767  
**typename** 511  
**type specifier** 38  
**typing**  
     dynamic 16  
     static 16

**unexpected()** 581  
**union** 236  
     size of 238  
**unitbuf** 795  
**unsetf()** 796  
**unsigned** 59  
**uppercase** 795  
**using** 922  
     declaration 922  
     directive 923

**vector** 362, 431  
     cross product 175, 654  
     dot product 653  
     norm 175, 652  
     normalised 653  
     unit 653  
**Vector** 362, 431, 502, 515, 584, 649, 652, 819, 943  
**Vector3D** 649, 934, 943  
**VectorIterator** 535, 745  
**vfprintf()** 876  
**vfscanf()** 876  
**view plane** 981  
**view plane-screen transformations** 981  
**viewer** 980  
**Viewer** 980  
**ViewPlane** 985  
**virtual** 689  
     base **class** *see* **class, virtual base**  
     functions 689  
         constructor 693  
         destructor 693  
         **friend** 693  
         **inline** 693  
         pure 696  
         **static** 693  
**void** 128  
**volatile** 61  
     data member *see* **class, data member, volatile**  
**vprintf()** 867

*vscanf()* 867  
*vsprintf()* 867  
*vsscanf()* 867  
  
**wchar\_t** 54  
WeekDays 240  
**while** 107  
white space 784  
*width()* 798  
*window()* 848  
Windows 734, 880  
*WinMain()* 160, 896, 1008  
  
World 831, 998  
world data file 1006  
world domain 934  
world objects 831  
*write()* 812  
WriteFile 821, 949  
ws 788  
  
X(X&) 282  
*xalloc* 582, 629  
*xmsg* 582, 629